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
When using today's productivity applications, people rely heavily on graphical controls (GUI widgets) as the way to invoke application functions and to obtain feedback. Yet we all know that certain controls can be difficult or tedious to find and use. As an alternative, a *customizable physical interface* lets an end-user easily bind a modest number of physical controls to similar graphical counterparts. The user can then use the physical control to invoke the corresponding graphical control's function, or to display its graphical state in a physical form. To show how customizable physical interfaces work, we present examples that illustrate how our combined *phidgets*® and *widget tap* packages are used to link existing application widgets to physical controls. While promising, our implementation prompts a number of issues relevant to others pursuing interface customization.

INTRODUCTION
Typical productivity applications, such as those included in Microsoft Office suite, present its users with hundreds of different functions. All make the tacit assumption that users should access these functions through on-screen GUI controls (e.g., buttons, menus, sliders) by using the mouse or keyboard shortcuts. Yet GUI controls have known problems.

1. Because space is at a premium, not all controls fit on the top-level display; many end up hierarchically nested in menus and dialog boxes. A person may find it difficult or tedious to discover and navigate to these controls.
2. The controls that are visible in the top-most window (usually clustered into tool palettes) may not be the ones that the person needs. Yet these controls compete with the application itself not only for display space, but also for the user's attention.
3. While nearly all GUI controls rely on the mouse and display for input and output, a pointing device is not necessarily the best input device for any given control task [2,3,20]. For example, because graphical controls provide only visual cues to its behavior, they demand considerable attention. Because only a single mouse is usually available, almost all interaction is constrained to one-handed input.

Greenberg, S. and Boyle, M. (2002) Customizable physical interfaces for interacting with conventional applications. Proceedings of the UIST 2002 15th Annual ACM Symposium on User Interface Software and Technology, ACM Press.

We challenge the assumption that all interaction must be through the mouse acting over GUI controls. Similar to other researchers who advocate tangible computing [2,8,9,12,15,17,19,20,21,22,24,25], we advocate supplementing interaction with physical interface controls. Input could be through physical push buttons, toggles, rheostats (dials and sliders), RFID tags, and light or pressure sensors. Output could be through LEDs, servo motors, or even off-the-shelf powered devices such as lamps and fans.

Physical controls offer a number of advantages over their graphical counterparts.

- *Screen real estate is saved*, leaving more room on the display for applications and diminishing competition for the user's attention.
- *Physical controls are usually top-level.* Controls are always visible, and are thus easier to locate and acquire.
- *More efficient input is possible*, since a physical control's form factor can more closely match the needs of the interaction [19,20,24]. For example, adjusting sound volume is easier through a rheostat slider *vs* a GUI slider because it constrains the user's actions along just one dimension and because it provides tactile feedback [2].
- *Two-handed input* is possible. For example, the dominant hand can control a mouse while the other hand controls the physical device [2,8,20].
- *Controls can be positioned 'ready to hand'* by bringing them close by when needed, but pushed to the periphery when not needed [26].
- *Spatial memory is better used*. Physical controls do not move about the workspace of their own accord [18,19], and thus people can quickly remember where they are.
- *All of a person's abilities are used* [9]. Consider an electric fan instead of a GUI progress bar to illustrate the progress made on a lengthy operation, where 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: seeing the moving fan blade, hearing its whir, and feeling its wind.

Given these advantages, why aren't physical controls more prevalent in modern interfaces? Some problems are obvious. Physical controls consume desk space, different ones take time to learn, and they are costly [3]. They also 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.

Thus we expect that we could give people only a modest number of physical controls, where they would be mapped onto the few functions that the person deems important or that one uses frequently. Yet this mapping is difficult to do. Although it has been repeatedly shown that people use only a small subset of the large number of functions available in most productivity applications, this subset (excepting a few universal functions) differs considerably from user to user [5,13]. 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 interfaces*, the main idea of which is:

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

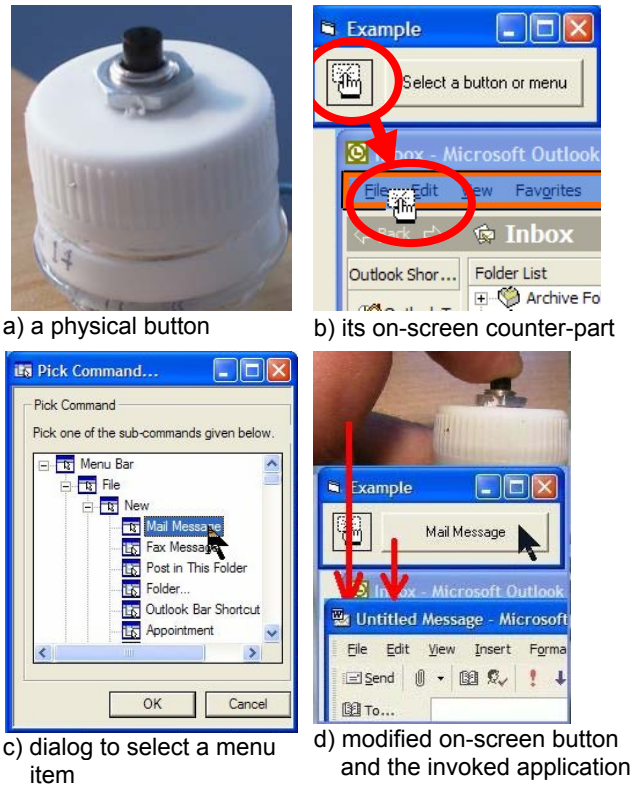
We also believe that customizable physical user interfaces will be realistic only if they work with existing unaltered applications. These could include not only well known productivity applications (e.g., Microsoft Office) but also niche software. In either case, customizing applications with physical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

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 GUI control 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 how they are built from the end-programmer's perspective. Next, we give a small representative sample of the interface design possibilities afforded by our architecture. We then raise several issues arising from our implementation, and conclude with a brief historical overview of related work.

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 button on the left triggers a *widget picker* operation, 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 clicks the picker button on the left of Figure 1b, which initializes the widget picker operation. The cursor changes its shape to indicate that a graphical widget should be selected, and the user moves it over the Outlook menu bar (see Figure 1b annotation). This particular picker recognizes 'command' widgets that invoke a single function, such as buttons, menu items and toolbar buttons. As the picker passes over a widget of this type, it indicates it is selectable by highlighting it in an orange box.
2. Selecting the Outlook menu bar raises a dialog box listing the many menu items it contains (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 and middle), a new Outlook mail message window appears (Figure 1d, bottom).

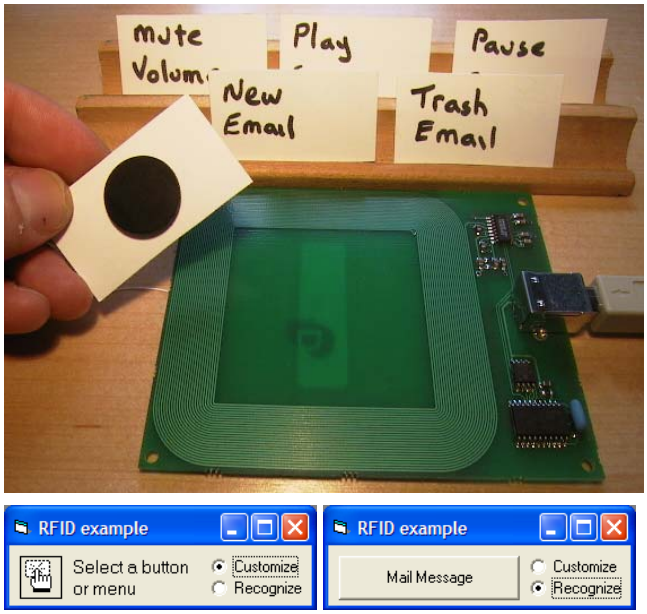


Figure 2. The RFID example.

Pressing 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 the tag near the reader, and by then selecting a menu or button as in the first example. To differentiate among the many tags, the user then simply writes the action (in his or her words) on the stiff paper, as shown in the figure. When in the 'recognize' mode, the 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 that the end-user has assigned RFID tags to invoke several e-mail and media player functions, and has organized them on wooden trays.

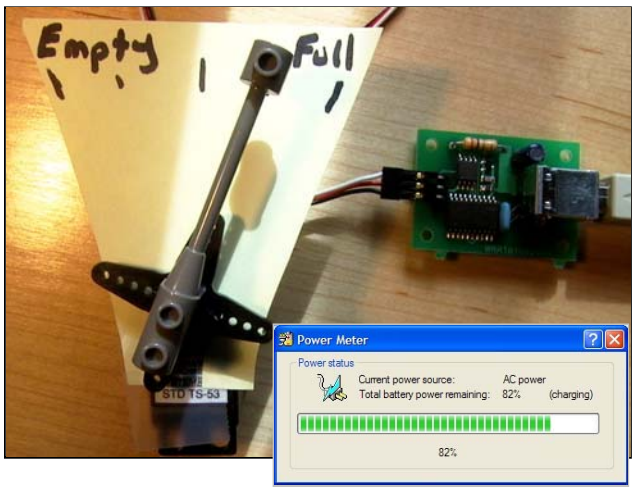


Figure 3. A gauge made with a servo motor connected to a laptop's power meter.

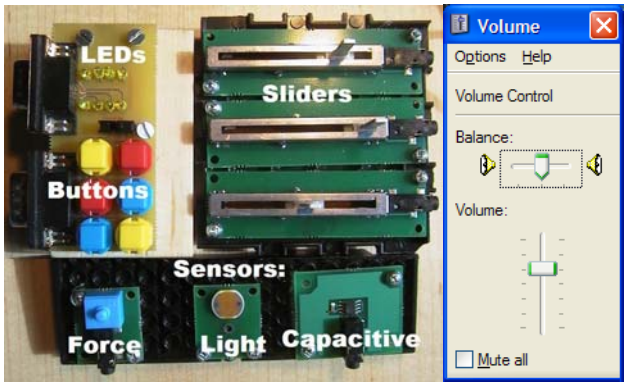


Figure 4. A console made up of many inputs and outputs used to adjust a software volume control.

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, some paper, and a Lego™ piece. The end-user has picked and linked this physical gauge to 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 this value.

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 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 one can attach to any widget that recognizes a continuous range of values, for example, a slider or scrollbar

For example, we can use this console to create a customized physical interface linked to the volume control pictured on the right of Figure 4. 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.

AN END-PROGRAMMER'S PERSPECTIVE

Our customizable physical interfaces architecture contains two main parts. First, *phidgets*® are *physical widgets* used

by programmers to easily access a myriad of physical controls (buttons, dials, sliders, switches) and displays (gauges, lights) [6]. Second, *widget taps* are programmer objects that 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, the widget tap 'taps in' to the functionality exposed by a GUI widget. End programmers write software that connects the phidgets and widget taps together, where the exposed application functionality is bound to physical devices. The four examples provided in the previous section are all examples of this type of software.

Physical widgets

Constructing and programming physical hardware is usually onerous and requires a great deal of specialized knowledge. To get around this, our physical interfaces are made with *phidgets*™, as reported in a previous UIST conference [6]. In this section, we briefly summarize what phidgets are and how they work, and show how they relate to this project.

Phidgets are input and output hardware devices packaged in a way that makes them very easy to program. With phidgets, an average programmer with no hardware knowledge can quickly prototype various customizations without spending effort developing special hardware, firmware, or software. Several phidgets are listed below, and are illustrated in the previous examples and figures.

- *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 eight digital output devices (e.g., LEDs and solenoids); retrieve the state of eight digital input devices (e.g., push buttons and switches); and inspect the state of four analog sensors (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 of the unique identity of an RFID tag passing over the reader's antenna.
- *PhidgetServo* comprises one or more servo motors (one is illustrated in Figure 3) where a motor's position is easily set through software.

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 and its callback exececuted. As shown below in our Visual Basic example, we check which digital input signaled the change (1 for the first button) 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.

```
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
```

Example 2 from above would extend this sample code by dynamically tracking these tag identifiers in an array, and by searching the array whenever it sees a tag.

A programmer can also 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 a 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 taps

An elusive goal of many research projects has been *external attachment*, where one can robustly attach external code to an application so that it can be remotely controlled [1,15,16,17,19,22]. Because of operating system limitations, no one has solved this problem completely, an issue we will explore later. Our own partial solution is our new *widget tap* library, which provides the programmer with objects that access the semantics of GUI widgets in existing applications. In turn, these expose the application function controlled by the selected widget which we can then link to a phidget. We illustrate this library by showing code fragments that drive the previous examples.

Our widget tap library abstracts analogous widgets into meta-classes. For example, the `CommandTap` class represents different GUI widgets that invoke a single unparameterized function (a command); these include push buttons, toolbar buttons, and menu items. This class has several important methods and properties. The `PickCommandTap` class method begins a modal cycle asking the user to select a push button, toolbar button, or menu item with the mouse. It then returns a `CommandTap` instance linked to the selected GUI control. The programmer can then use the `Text` property of this `CommandTap` instance to retrieve any text label associated

with that button or menu item, and the `Click` method to execute the application functionality triggered by the GUI widget i.e., as if the actual GUI widget were selected.

The widget tap library has other classes as well. The `ToggleTap` class abstracts GUI widgets— checkboxes and radioboxes—that contain boolean state. While it shares the same methods and properties found in `CommandTap`, it also contains a property called `Checked` that sets or gets the current Boolean state. As another example, the `RangeTap` class abstracts GUI widgets that let a user choose a single value in a minimum/maximum bounded range. This includes scrollbars, sliders, up/down (a.k.a., spinner) controls, and progress bars. While there are differences between these widgets, the `RangeTap` class exposes their common important semantics through its `Minimum`, `Maximum` and `Value` properties.

To show how the widget tap and phidget library are used together, the Visual Basic program in Figure 5 contains the key code fragments that sit behind the customizable button of Example 1.

1. Using the Visual Basic interface builder, the programmer constructs the window in Figure 1b by dropping in: a picture named `wPicker`, and a button named `GUIButton` with its text set to the string "Select a button or a menu".
2. The programmer declares a `CommandTap` object called `wTap`, which will eventually be linked to a command-type GUI widget.
3. After the user clicks the `wPicker` picture, the program executes the `wPicker_MouseDown` callback. Its code first calls the `CommandTap.PickCommandTap` class method, which asks the user to interactively select a command widget. In the example, he chose Outlook's New Mail Message menu item (Figures 1b+c). This returns a `CommandTap` instance that is stored in the `wTap` global variable. This instance is now linked to the New Mail Message menu item of Microsoft Outlook.
4. The programmer retrieves the text label from the linked widget (i.e., "Mail Message") via the `wTap.Text` property and displays it in the GUI button's label property. (Figure 1d).
5. When the end-user presses the GUI button, its `GUIButton_Click` callback is executed, which in turn calls the `wTap.Click` method. This invokes the equivalent semantic operation on the linked widget. In this case, a new Outlook mail message will appear (Figure 1d).
6. Similarly, pressing the physical button automatically invokes the `phidgetIK_OnInputChange` callback. This also calls the `wTap.Click` method with the same results.

The code behind the RFID example in Figure 2 is very similar except that it maintains an array of `CommandTap` objects, one for each RFID tag encountered.

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

'The user clicked on the wPicker icon (Figure 1b left side).
Sub wPicker_MouseDown(…)
   'Let the user select a command widget,
   'and return an instance linked to this widget
   wTap = CommandTap.PickCommandTap()
   'Display the selected widget's label in the GUI button
   'visible on the right of Figure 1b
   GUIButton.Text = wTap.Text
End Sub

'When the GUI button is pressed, invoke the widget's action
Sub GUIButton_Click(…)
   wTap.Click()
End Sub

'When the physical button is pressed, invoke the widget's action
Sub phidgetIK_OnInputChange(Index, State)
   If Index = 1 and State = True Then _
      wTap.Click()
End Sub
```

Figure 5. The code behind Example 1's Button. Error handling is omitted for display purposes.

The third example provided a physical customization of the "battery charge" progress bar, where output is displayed on a physical control. To implement this, the `RangeTap` is attached to the progress bar of the battery meter. The program then periodically polls the `Value` property of the `RangeTap` instance, and updates the position of the phidget servo motor to reflect the current value. This is illustrated in the subroutine below; its calculations just normalize the range value to fit between the 0 – 180 degree positions of the servo motor.

```
Private Sub timerPoll_Tick(…)
   Dim ratio As Double
   ratio = (wTap.Value    - wTap.Minimum) /
           (wTap.Maximum - wTap.Minimum)
   Servo.MotorPosition(1) = 180 * ratio
End Sub
```

The fourth example includes a physical slider as input. In this case, the physical slider manipulates the `RangeTap` instance and its linked GUI control, again normalizing for differences. For example, when the first physical slider is moved, the code to adjust the linked GUI slider would look something like:

```
Private Sub phidgetIK_OnSensorChange(…)
  If Index = 1 Then
    WTap.Value = SensorValue / 1000 *
    (WTap.Maximum - WTap.Minimum) +
    WTap.Minimum
End Sub
```

We should add that the tap library is reasonably robust. If the original application were closed in the interim (e.g., the user linked to Microsoft Outlook and then closed Outlook), a call to the `wTap.Click` method would automatically restart the application, reconnect the widget tap to the new target widget instance, and invoke the application function.

The examples above are simple, and we can construct far more interesting ones. What is important is that our widget

tap library gives the programmer access to the semantics of *any* recognized GUI 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.

DESIGN POSSIBILITIES

While the basic idea of a customizable physical interface is 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.

Interfaces for people with special needs. While many people suggest that computers should help those with special needs (and this is even legally required in some instances [7]), 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. For example, if a progress bar is mapped to a fan, those with visual and/or aural impairments will benefit.

It is important to mention that assistive technology product makers often use approaches similar to what we describe here. However, our phidget and widget tap library is much more general purpose: it does not focused on one or a few particular kinds of ability impairments, nor is it wholly restricted to the assistive technology domain.

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.

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.

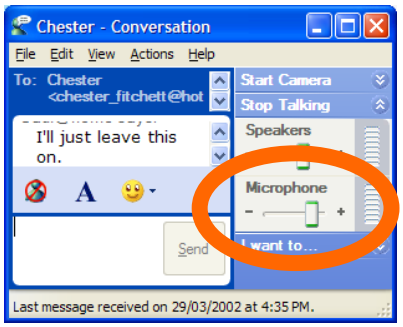


Figure 6. Controlling Windows™ Messenger.

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 linking a capacitive sensor to the microphone volume level (circled in Figure 6), and then embedding that sensor 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 those 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.

ISSUES

There is much promise in customizable physical interfaces, yet we have discovered many roadblocks to its robust implementation. Most are associated with problems related to external attachment i.e., how we can reliably tap into graphical widgets. While external attachment is only a secondary theme in this paper, it is the bottleneck to customizable physical interfaces and deserves further discussion. After first describing several approaches to external attachment, this section presents the major implementation issues as well as our partial solutions.

Approaches to External Attachment

In our wish to provide physical customizations of existing applications, we share an implementation goal with other researchers: the ability to invoke existing application

functionality without having to see or modify its source code. While there are several approaches, many do not fit well with our needs.

One approach invokes application functionality through hooks, APIs, scriptable object models and automation features predefined by the application itself. This is problematic. First, we can only access those functions that the application programmer decided to supply *a priori*. Second, these functions are hidden from the end user: they do not have a graphical, on-screen presence. Even if we exposed the function names, they are not in the end-user's language. This makes it difficult for an end-user to identify which function he or she wishes to invoke.

A second approach uses function aliases. For example, Siio and Mima [22] bind paper stickers holding a barcode to Apple Macintosh's icon aliases. Scanning the barcode invokes the alias, which starts the application. This is too simplistic, as it does not provide access to the internal functions of an application.

A third approach captures the user's syntactic input, and replays it on the interface [1]. An advantage of this 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 trigger the same action sequence'. The problem is that syntactic access is difficult to implement reliably. It fails in GUIs when things do not appear in constant locations or when the interface is rearranged from one invocation to the next. It is ignorant of the modes that the application may be in, and provides little opportunity to assess an application's feedback.

Our own approach remotely controls another application by accessing the semantics of its widgets, an approach shared by the Mercator [17] and parts of the Pebbles project [15,16]. The details as to how we tap into widgets is too involved to include here and we will simply summarize that our basic technique is common to various screen reader and accessibility utilities and to those used in Myers et al.'s *semantic snarfing* for Pebbles [16]. We synthesize notifications (messages) that are exchanged between the widget and the application. To invoke an application function, we masquerade as the GUI widget and send the application a message that appears as if it came from the widget. To query or modify the GUI widget state, we masquerade as the application and send the widget the appropriate message. However, our widget tap implementation goes one step further as it attempts to handle the following issues.

Finding and Re-finding Widgets

It is easy to get a 'handle' identifying an existing GUI widget. However, if the application is stopped and restarted, or if the widget is destroyed and recreated, we face serious problems locating the new instance of the widget. If we cannot find the widget in these cases, then the user will have to manually re-link the physical control to the GUI widgets between invocations.

This problem arises because widgets have no instance-invariant names in current windowing systems such as X11R6 and Microsoft Windows. That is, they do not have unique identifiers that remain the same when the widget is destroyed and recreated, or when it is moved around. While windowing systems do provide unique window handles at runtime, these are meaningless and change each time a widget is recreated.

We solved this problem by inventing an instance-invariant naming scheme, which is comprised of a path of three colon-delimited parts: the first is the file name of the process that provides the widget; the middle part is the path to a heavyweight widget or widget container; the last optional part specifies a lightweight widget relative to its heavyweight container. (We should note that generating this path correctly is a technically difficult problem). This name is retrievable in each widget tap through its `Path` property. For example, if we were to display the `wTap.Path` property in Figure 5 after the New Mail Message menu item was assigned to `wTap`, we would see:

```
outlook:Standard,MsoCommandBar:New/Mail Item
```

This name allows us to re-find widgets in several ways. Because the first part of the name path specifies the process responsible for creating the widget, we now know which process to restart if the application is not running. The second part of the name allows us to find a heavyweight widget by querying the window system. If we are actually linked to a lightweight widget, the third part of the name allows us to find the lightweight widget relative to its heavyweight container. All this is done automatically by the library if the `wTap.AutoReconnect` property of the tap object is set to true.

We should note that although the grammar of names will be necessarily platform-specific, the idea of instance-invariant names is a general solution to re-finding widgets.

Some Widgets are not Initialized with the Application

If a tapped widget is destroyed, then the widget will be re-found as a simple and direct consequence of restarting the application process that created the widget. Unfortunately, this only works for widgets that are initialized along with the application. Yet some programs create widgets on-the-fly, e.g., those that live in short-lived or modal dialog boxes that are not part of the persistent UI for an application, or those that are created on demand as a user switches between panes in tabbed dialogs. We have no solution to this problem. However, we suspect most people will attach physical controls to permanent widgets representing standard application functions rather than these short-lived ones.

Lightweight Widgets

Widgets come in two flavors. *Heavyweight widgets* have a representation known to the windowing system i.e., they are true windows. Thus we can obtain information about them directly from the windowing system. In contrast,

lightweight widgets are completely managed by the widget toolkit, and are not known by the windowing system. The problem is that there is no toolkit-independent way of finding these lightweight widgets.

Our solution uses information obtained from the windowing system about the heavyweight container to deduce which toolkit provided it. Knowing this, we can proceed to use toolkit-specific techniques to locate and tap into the lightweight widget.

This implies that we must know how all toolkits operate if we are to do this robustly. The widget tap library currently uses the *Active Accessibility* features of Microsoft Windows, an API intended to give people with disabilities alternate means for accessing computer functions (see msdn.microsoft.com). Unfortunately not all toolkits implement Active Accessibility. There are also language-specific problems with other toolkits that limit how well we can support lightweight widgets. While we implemented a limited point solution, there is as yet no universal solution to this problem.

Widgets are not Self-Describing

For a library like widget tap to work, it needs to understand how each widget works. For example, the library provides two implementations of the `CommandTap` base class. One understands the various flavors of system-supplied push buttons. The other understands toolbars and menubars as they appear in Microsoft Office applications. A configuration file is used to determine which implementation is appropriate for a given widget.

The problem is that there is no way of automatically discovering the semantic operations and messages implemented by a widget. That is, widgets are not self-describing, nor do they describe their relationship with the containing application.

Currently, we must consult each widget's documentation, and manually write code in our library to handle that particular widget type. This is tedious to do, as it implies we need to write code that understands every single possible type of widget that fits within the meta class. Even if we went through this effort, this only accounts for the widgets available today. If a system release included new widgets, then new code would have to be written to handle them. Another serious problem is that there is no easy way of including undocumented widgets that were perhaps created as part of a custom application or from a non-standard toolkit.

A solution is in sight, for operating systems are encouraging toolkit designers to implement accessibility APIs. However, this is still not standard practice. Even if it were done, it may not apply to applications using older version of these toolkits.

Modes are Difficult to Handle

Many interfaces are moded, where operating a widget in some modes makes little sense. For example, a widget that draws a border in a table is of little use if a table is not selected. Conventional interfaces typically manage modes by visual feedback (graying out or hiding the widget) and by disabling the widget.

These modes lead to two problems for customizable physical controls. First, a tap into a widget has only a crude and perhaps unreliable means of detecting whether it is in a valid mode. It can check if the widget is enabled, disabled, visible or hidden. Second, it is difficult to show that physical controls are disabled. Yet because they are always present, people feel that they can use them at any time. This means the end user may try to invoke a widget action at an inappropriate time through its physical control. At worse we hope this will lead to a null action that may be puzzling to the user.

HISTORY AND RELATED WORK

In 1963, Ivan Sutherland [23] demonstrated Sketchpad, the very first interactive graphical user interface. Films of Sketchpad [14] 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 [14,23]. 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

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)

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) modify mouse actions [4].

While Engelbart's system did away with most special purpose physical controls, they appeared again as special purpose function keys in the Xerox Star [11]. 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.

Figure 9. Star's left function key cluster.

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. There are new input devices e.g., [20], and novel ways to control new classes of computers e.g., tilting and panning actions for scrolling through items on a PDA [8]. Researchers are bridging physical world objects with computer objects through tagging and tracking [19,22,25], or by creating physical remote controls [15,16] that operate on conventional graphical user interfaces. Perhaps the closest to our work is tangible media [9,10], which describes how physical media can be attached to digital information and controls. An excellent example is Ullmer, Ishii and Glas's mediaBlocks [24]. 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.

There are many more exciting examples of how new technology can use physical devices. 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, albeit in a limited way. However, we feel that the pendulum has been swung too far, and applications are now so dependent on GUI widgets that we have lost the benefits of judicious application of physical controls.

In this paper, we presented our notion of customized physical interfaces to existing applications. We described how we combined our phidget and widget tap package to allow programmers to seize upon this design idea, and offered examples demonstrating its use. We also discussed many of the problems found in external attachment. While we have made good headway in solving the external attachment problem, there remain several impediments. In the end, some of these issues may be only addressable by windowing system and GUI toolkit makers.

Our future work in this area will focus partly on finding solutions to the external attachment problems presented here, but mostly on the design opportunities afforded by customizable physical interfaces.

Software and hardware availability. Phidget hardware and software is available through www.phidgets.com. The widget tap library and examples will be available fall 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. We also thank the referees who reviewed an earlier version of this paper. They were very

knowledgeable and gave excellent recommendations for improving it.

REFERENCES

1. Bharat, K. and Sukaviriya, P. Animating User Interfaces Using Animation Servers. *Proc ACM UIST'93*, 69-79, 1993.
2. Buxton, W. and Myers, B. A Study of Two-Handed Input. *Proc ACM CHI'86,* 321-326, 1986.
3. Card, S., Mackinlay, J. and Robertson, G. A morphological analysis of the design space of input devices. *ACM Trans Information Systems*, 9 (2), 99-122, 1991.
4. Engelbart, D. and English, W. A Research Center for Augmenting Human Intellect, *AFIPS Conference Proc Fall Joint Computer Conference* (33), 395-410, 1968.
5. Greenberg, S. *The computer user as toolsmith: The use, reuse, and organization of computer-based tools.* Cambridge University Press, 1993.
6. Greenberg, S. and Fitchett, C. Phidgets: Easy Development of Physical Interfaces through Physical Widgets. *Proc ACM UIST'01*, 209-218, 2001.
7. 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.
8. 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
9. Ishii, H. and Ullmer, B. Tangible bits: Towards seamless interfaces between people, bits and atoms. *Proc ACM CHI'97*, 234-241, 1997.
10. Ishii, H., Mazalek, A., Lee, J. Bottles as a minimal interface to access digital information. *Extended Abstracts of ACM CHI*, 2001.
11. 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.
12. Kaminsky, M., Dourish, P., Edwards, K. LaMarca, A., Salisbury, M. and Smith, I. SWEETPEA: Software tools for programmable embodied agents. *Proc. ACM CHI'99,* 144-151, 1999.
13. McGrenere, J., Baecker, R. and Booth, K. An evaluation of a multiple interface design solution for bloated software. *Proc ACM CHI'02*, 163-170, 2002.
14. MIT. Sketchpad. ACM CHI'83 Video Program in SIGGRAPH Video Review Issue 13. 1983.
15. Myers, B., Peck, C., Nichols, J., Kong, D. and Miller, R. Interacting At a Distance Using Semantic Snarfing, *ACM UbiComp'01*, 305-314, 2001.
16. Myers, B., Miller, R. Bostwick, B. and Evankovich, C. Extending the Windows Desktop Interface With Connected Handheld Computers. *4th Usenix Windows Systems Symposium*, 79-88, 2000.
17. Mynatt, E and Edwards, W. K., Mapping GUIs to Auditory Interfaces, *Proc ACM UIST'92*, 61-70, 1992.
18. Patten, J. and Ishii, I. A comparison of spatial organization strategies in graphical and tangible user interfaces. *Proc ACM DARE'00*, 41-50, 2000.
19. Pedersen, E., Sokoler, T. and Nelson, L. PaperButtons: Expanding a Tangible User Interface. *Proc ACM DIS'00*, 216-223, 2000.
20. Rekimoto, J. and Sciammarella, E. ToolStone: effective use of the physical manipulation vocabularies of input devices. *Proc ACM UIST'00,* 109-117, 2000.
21. Resnick, M. Behavior construction kits. *Communications of the ACM* 36(7), 64-71.
22. Siio, I. and Mima, Y. IconsStickers: Converting Computer Icons into Real Paper Icons. *Proc HCI International* (Volume 1), 271-275, LEA Press. 1999.
23. Sutherland, I. Sketchpad: A man-machine graphical communications systems, *Proc Spring Joint Computer Conference*, 329-346, Spartan Books, 1963.
24. Ullmer, B., Ishii, H. and Glas, D. mediaBlocks: Physical Containers, Transports, and Controls for Online Media. *Proc ACM SIGGRAPH'98*, 379-386, 1998.
25. Want, R., Fishkin, K., Gujar, A. and Harrison, B. Bridging Physical and Virtual Worlds with Electronic Tags. *Proc ACM CHI'99*, 370-377, 1999.
26. Winograd, T., and Flores, F. Understanding Computers and Cognition: A New Foundation for Design, Ablex, 1986.