Phidgets: Incorporating Physical Devices into the Interface

Saul Greenberg and Chester Fitchett
Department of Computer Science
University of Calgary
Calgary, Alberta, Canada T2N 1N4
+1 403 220 608
saul@cpsc.ucalgary.ca

ABSTRACT
Physical widgets, or *phidgets,* comprise devices and software that are almost direct analogs of graphical user interface widgets. Like widgets, phidgets abstract and package input and output devices: they hide implementation and construction details while exposing functionality through a well-defined API. They also have an (optional) on-screen interface. Phidgets also require: a connection manager to track how devices appear on-line; a way to link a software phidget with its physical counterpart; and a simulation mode to allow the programmer to develop, debug and test a system using phidgets even when no physical device is present.

INTRODUCTION
In the last decade, a variety of researchers have championed human-computer interface designs that include 'out of the box' physical devices augmented by computing power. Various movements have embraced this approach: *ubiquitous computing* and *calm technology* [13], *pervasive computing* [1], *tangible user interfaces* [7,2], *information appliances* [10], and *context-aware computing* [3].

While an exciting new area, everyday programmers now face considerable hurdles if they wish to create even simple device-dependent applications. Perhaps the biggest—but most easily solved—obstacle is the sheer difficulty of developing and combining physical devices and interfacing them within the application software. Several specific problems are listed below.

1. Even simple devices made out of cheap and readily available components (switches, sensors, motors) are hard to build unless one has a background in hobby electronics, circuit design or electrical engineering.
2. Commercially available devices may have no published application programmer's interface (API). As a result cannot be programmed by an outsider unless the device is 'hacked' (e.g., Microsoft's Actimates hacked by Kaminsky, Dourish and Edwards [8]; and Fujitsu's email notificaton figurine hacked by Greenberg and Kuzuoka [4]).
3. Alternatively, commercial devices with an API are often at the wrong level of abstraction for easy use. Some are designed for particular application settings: the device and its accompanying software may be difficult to subvert to new situations (e.g., X10 protocol devices packaged as home and security products, but see [5]). Others may be abstracted at a very low level, where designers may have to do extensive programming to do even the simple things.
4. Programmers may not have these devices readily available at all stages of their programming effort (perhaps due to expense, shipping delays, cost factors, etc.) While a program can be written without a device, they are difficult to test and debug.

Greenberg, S. and Fitchet, C. (2001) **Phidgets: Incorporating Physical Devices into the Interface.** In M. Newman, K. Edwards and J. Sedivy (Eds) *Proceedings of the Workshop on Building the Ubiquitous Computing User Experience.* (Held at ACM CHI'01, Seattle). Also as *Report 2001-681-04*, Dept Computer Science, University of Calgary, Alberta, Canada.

http://www.cpsc.ucalgary.ca/grouplab/papers/index.html

OUR FRUSTRATING FIRST EXPERIENCES
Our own first experiences echoed these problems. We were designing a reactive media space environment (Figure 1) built around several simple interoperating devices [4]. The devices illustrated in the figure were built upon proximity

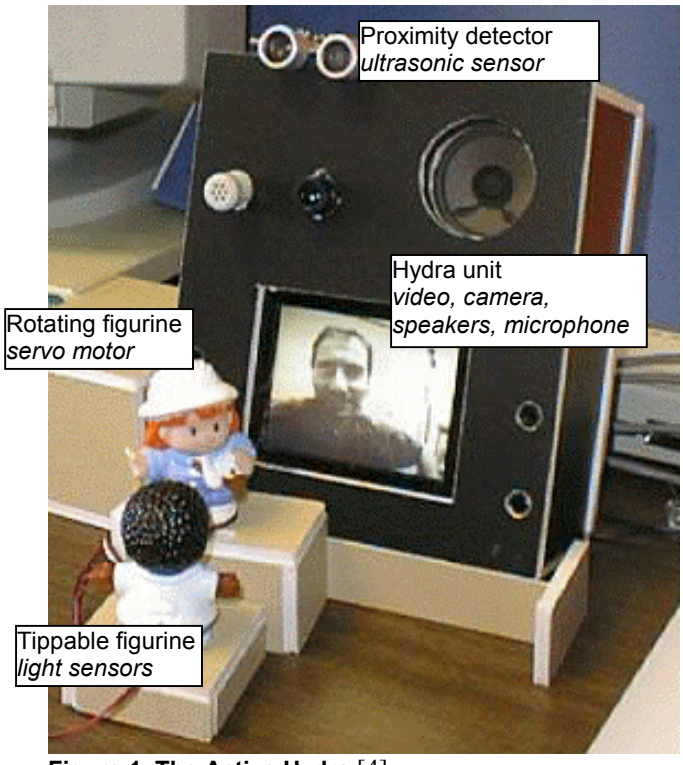

Figure 1. The Active Hydra [4].

sensors, servo motors and light sensors, as well as switched cameras, microphones, speakers, and small video displays. While our overall focus was on media space design (details in [4]), we found ourselves immersed in a quagmire of tediousness: selecting and purchasing small electrical components and hobby kits, circuit board design, microprocessor programming, wire protocol development, and so on. Fortunately for us computer scientists, visiting collaborator Kuzuoka (an electrical engineer) did all the hardware/microprocessor work. Still, we expended considerable time (months) and effort developing and debugging these devices and their related low-level software. Although successful as a stand-alone project, the tale ended poorly: after Kuzuoka left the software and the devices themselves became almost impossible to maintain or extend. The problem was that we had built a working prototype, but had not really considered how individual devices and its software could be maintained, modified and reused in different ways. Consequently, the devices ended up in a cardboard box, full of unrealized potential.

THE PHIDGET CONCEPT

We then made a concerted effort to think about how we could package devices and their accompanying software. We wanted devices that were easy to program, test, debug, and extend. Most importantly, we wanted devices that were simple enough so that we could concentrate on the overall user interface design instead of low-level device construction and implementation. Our approach was to develop physical widgets, or *phidgets*, that are almost direct analogs to how graphical user interface widgets are packaged and 'dropped into' applications[1]. As we will see, a phidget comprises a device and an API to it via a corresponding software.

Why GUI Widgets are so successful

GUI widgets have greatly simplified the programmer's development of interactive software. They abstract and package well-designed standard and non-standard input and output controls. They hide (often difficult) implementation details, while exposing functionality through a well-defined API. Through relatively simple programming, they can be interconnected so they can work in concert with one another. As a toolkit set, widgets give the programmer a good repertoire of graphical components that can be used to assemble an interface [9]. The result is that programmers using widgets could concentrate on GUI interface design rather than low-level graphical programming[2].

Phidgets as Physical Widgets

As with conventional GUI widgets, the important idea of a phidget is that it presents the programmer with an easily used entity that can be inserted into an application. They both provide an abstracted and well-defined interface: one to a graphical interactive device, the other to a physical one. Both hide details of how the entity is implemented. Unlike conventional widgets, phidgets require a few more things.

1. *Connection manager.* Whereas GUI widgets are always available to the application at run time, physical devices may appear and disappear. For example, during run time a device may come on-line or go off-line, or it may have intermittent connectivity. The job of a connection manager is to inform the application program about the appearance and disappearance of particular devices, and to give the programmer a 'handle' to devices as they appear.
2. *Identification.* There must be a way to link a software phidget with its physical counterpart. While not a problem when there are only a few well-known devices attached to a single computer, this can become an issue when several devices of the same type (but perhaps with different end uses) are attached to the computer, or where the types and numbers of devices are not known ahead of time.
3. *Simulation mode.* For software development purposes, the same phidget code should work in a simulation mode. That is, the software designer should be able to program, debug and test the system even if the actual physical device corresponding to the phidget is absent. This could include an extended API to set the simulation characteristics of the device, and a graphical representation that allows a person to interactively see and optionally set the device state.

WHAT WE BUILT

With these features in mind, we designed and built several phidgets. Most of our phidgets are built around the CY7C63000 USB micro-controller from Cypress Semiconductor. Our phidgets connect via the USB port to a computer running MS Windows, and are seen by Windows as a USB device[3]. Each device knows and can transmit its phidget type, as well as an identification number that is unique for a phidget instance of that type (see Point 2 above). On the software side, we wrapped all the software used to interact with a particular device type (including the wire protocol and the device driver interface) as an ActiveX COM Component. That is, programmers can create a software instance of a phidget component, and can access any of its (abstracted) properties, methods and events via a documented and simple API. This phidget component can operate in a simulated mode (where the software mimics the

[1] Phidgets differ from Phicons [12]. Phicons are input instruments. Phidgets are programmable components representing physical objects.

[2] Myers [9] argues that there is a disparity in many GUI toolkits, where building control panels of widgets is extremely easy, but composing non-widget graphics is hard.

[3] We also built phidgets atop the 16F84 micrcontroller from Microchip Inc. which connected to the RS-232 serial port. We may build future versions atop X10 and/or wireless protocols.

device's behavior: see Point 3), or it can be connected to an actual physical device (Point 1). Each phidget component also has a corresponding visual component (an ActiveX control). This provides a visual on-screen interface to the device that display its real or simulated state, and that optionally lets an end-user interactively control it. Finally, another ActiveX component acts as a connection manager: it raises an event at run time when devices connect or disconnect. The programmer can check the device's type and identification (if needed), and then connect that device to its matching phidget component.

Example

Phidgets we have built (or have almost completed) include:

- *GlabServo:* a controller for several servo motors, where the position of each motor can be set programmatically (see Figure 2);
- *GlabPowerBar:* a power bar where individual outlets can be programmatically turned on and off for various time durations;
- *GlabProximitySensor:* a device that periodically determines how close something is to it;
- *GlabIO:* a device that controls up to 8 simple output devices (e.g., LEDs) and returns the state of up to 8 simple input devices (e.g., switches and heat sensors).

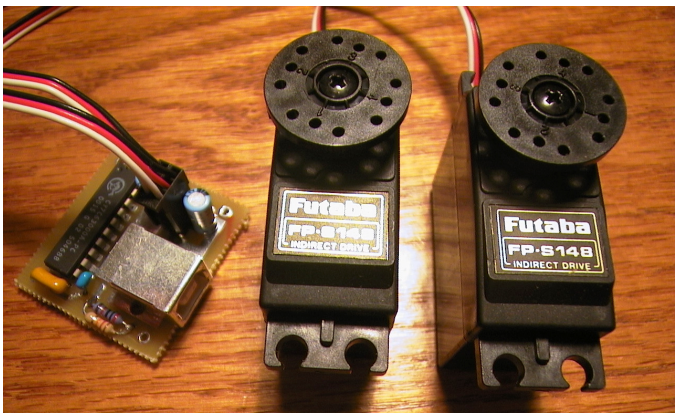

Figure 2. The undecorated Servo phidget device

The Visual Basic program in Figure 3 illustrates the complete source for an application that uses the PhidgetManager and the GlabServo phidget to implement a physical 'clock' with two flaps that open or close every second (the flaps are just bits of plastic glued each motor platform). While a nonsense application, it does serve to illustrate how simple it can be to program physical devices. We could also (with few changes) simulate this application by using the visual version of the servo phidget and seeing its behavior on screen.

```
Private WithEvents PM As New GlabPhidgetManager     'The phidget manager
Private Servo As GlabServo                          'The servo phidget

'Create a new instance of them on start up, and configure a timer
Private Sub Form_Load()
   Set PM = New GlabPhidgetManager
   Timer1.Enabled = False                  'Set up the timer to tick once/second
   Timer1.Interval = 1000                  'Timer measurements in milliseconds
End Sub

''When the phidget manager detects that a new Servo controller has been connected,
' link it to the servo phidget, set their intial positions, and start the timer
Private Sub PM_OnAttach_(ByVal Phidget As GlabPhidget.IGlabPhidget)
  If Phidget.DeviceType = "GLAB Servo Controller" Then
      Set Servo = Phidget
      Servo(1).ServoPosition = 0                'Settings are in degrees
      Servo(2).ServoPosition = 180              'Start the timer
      Timer1.Enabled = True
  End If
End Sub

'Turn off the timer when the device is disconnected,
Private Sub PM_OnDetach(ByVal Phidget As GlabPhidget.IGlabPhidget)
    If Phidget.DeviceType = "GLAB Servo Controller" Then
        Timer1.Enabled = False
    End If
End Sub

'On every tick of the timer, flip the two servos 180°
Private Sub Timer1.Timer()
    Servo(1).ServoPosition = Servo(2).ServoPosition
    Servo(2).ServoPosition = 180 - Servo(1).ServoPosition
End Sub


'The servo phidget generates an event every time its position is changed.
'However, we don't do anything with this event in this example: its here just for illustration
Private Sub Servo_OnServoPosition (Index, position)
End Sub
```

Figure 3. The complete program for controlling an odd physical clock

Even this simple phidget set can let people replicate existing physical devices. Natalie Jeremijenko pioneering *Dangling String*—an 8 foot plastic string that vibrates to indicate the amount of local Ethernet traffic [13]—can be easily created using the GlabServo with a program similar to that illustrated in Figure 2. Dahley, Wisneski, and Ishii's *Pinwheels* [2]—a motorized toy fan used to broadcast events— can be built atop the GlabPowerBar (to control the motors that spins the pinwheels). Heiner, Hudson and Tanaka's *information percolator*—water-filled tubes that can display patterns as bubbles [6]—can be built using the GlabPowerBar to rapidly switch the aerator pumps on and off.

We can enrich the kinds of applications we build by including one more software component: a notification server [11, 4, 5]. Our version of the notification server implements a shared dictionary: any distributed process can set key/value pairs into this dictionary, and all processes see changes to these keys/values as events [4]. It then becomes very simple to program groupware based on physical devices,

such as those shown in Figure 1. For example, an application can capture a person's presence using a well-positioned GlabProximitySensor and write that into the shared dictionary. Other applications can see this information and use it to activate physical devices. For example, it can use the GlabServo to rotate a figurine as shown in Figure 1, or turn a lamp on and off [5] with the GlabPowerBar. Our example programs for controlling and interconnecting these devices are surprisingly short, each taking only minutes to write.

FINAL THOUGHTS

Our main message is that packaging devices as physical widgets or *phidgets* greatly simplifies programming these devices, which in turn allows designers to concentrate on how devices can be crafted to fit within the environment *vs.* low-level implementation details. Of course, this is not a revolutionary idea: we suspect that existing practitioners have already packaged their own devices for internal reuse. We are surprised, however, that there has been no real push to publish, standardize and even to commercialize devices as phidgets. Yet there is a real need for this: almost all the people we have talked to who developed systems based on physical devices—researchers, developers, artists—had to start from scratch.

As well, we need to define a 'standard' phidget set. This already exists for GUI toolkits; for example, virtually all sets include buttons, listboxes, checkboxes, textboxes and so on. However, it is unclear what phidgets would be included in a standard phidget set. Certainly the ones we mentioned are likely candidates, but there are likely many more. As with GUI widgets, this phidget set must provide the programmer with conceptual building blocks that are not only individually useful, but can be assembled in a way that lets the designer build a rich ubiquitous computing experience.

Acknowledgements. We thank our collaborator Professor Hideaki Kuzuoka (University Tsukuba), whose visit started this going. The Collaboration Group at Microsoft Research, the National Sciences and Engineering Research Council of Canada (NSERC), and the Alberta Software Engineering Research Consortium (ASERC) partially funded this work.

REFERENCES

1. Ark. W. and Selker, T. "A look at human interaction with pervasive computers." *IBM Systems Journal* 38 (4), 1999.
2. Dahley, A., Wisneski, C. and Ishii, H. "Water Lamp and Pinwheels: Ambient projection of digital information into architectural space." *Summary of CHI '98*, 269-270, 1998.
3. Dey, A. K., Salber, D., Abowd, G. D. A conceptual framework and a toolkit for supporting the rapid prototyping of context-aware applications. Human-Computer Interaction, 16, in press-2001.
4. Greenberg, S. and Kuzuoka, H. "Using digital but physical surrogates to mediate awareness, communication and privacy in media spaces." Personal Technologies, 4(1), January, Elsevier. 2000.
5. Gruen, D., Rohall, S., Petigara, N. and Lam, D. (2000) "In your space" displays for casual awareness. Demonstration at ACM CSCW 2000.
6. Heiner, J., Hudson, S. and Tanaka, K. "The information percolator: ambient information display in a decorative object." *Proc ACM UIST'99 Symposium on User Interface Software and Technology*. 141-148, 1999.
7. Ishii, H. and Ullmer, B. "Tangible bits: Towards seamless interfaces between people, bits and atoms. *Proc ACM CHI'97 Conference on Human Factors in Computing Systems,* 234-241, 1997.
8. Kaminsky, M., Dourish, P., Edwards, K. LaMarca, A., Salisbury, M. and Smith, I. "SWEETPEA: Software tools for programmable embodied agents." *Proceedings of ACM CHI 99 Conference on Human Factors in Computing Systems,* 144-151, 1999.
9. Myers, B. (1995) State of the Art in User Interface Software Tools. In Baecker, R., Grudin, J. Buxton, W. and Greenberg, S. *Reading in Human Computer Interaction: Towards the Year 2000.* Morgan Kaufmann.
10. Norman, D.A. *The Invisible Computer*. MIT Press, 1998.
11. Patterson, J., Day, M. and Kucan, J. Notification servers for synchronous groupware. *Proceedings of ACM CSCW'96 Conference on Computer-Supported Cooperative Work*, p122-129. 1996.
12. Ullmer, B., Ishii, H. and Glas, D. "mediaBlocks: Physical Containers, Transports, and Controls for Online Media." *Proceedings of the 25th Annual Conference on Computer Graphics.* 1999.
13. Weiser, M. and Brown, J. "Designing calm technology", *Powergrid Journal,* v1.01, July, 1996.