A Toolkit for the Construction of Real World Interfaces

D. Scott McCrickard, Dillon Bussert, David Wrighton *

Department of Computer Science
Virginia Polytechnic Institute and State University
Blacksburg, VA 24061-0106
{mccricks, dbussert, dwrighto}@vt.edu

ABSTRACT

In recent years, real world objects have been used to reflect information previously shown on the computer screen. While most earlier efforts have required significant developer knowledge and skills to construct and program the displays, our approach enables programmers to use real world objects in much the same way that they would typical user interface widgets. The programming interfaces leverage existing paradigms, simplifying the integration of off-the-desktop display and interaction techniques into standard programs. The APIs are developed using the X10 protocol for controlling power flow to electrical devices, thus avoiding engineering issues that make construction difficult and time-consuming for typical programmers. Sample programs are described that use the programming interfaces.

Categories and Subject Descriptors

H.5.2 [**Information Interfaces and Presentation**]: User Interfaces

Keywords

Ubiquitous computing, pervasive computing, user interface toolkits, tangible user interfaces

1. INTRODUCTION

With the availability of information sources like the World Wide Web, users often need to stay aware of constantly changing information, or at least have the information readily available at appropriate times and in appropriate forms. Quite often, the information is not so important that it should occupy space in a display on the computer desktop, yet it is of sufficient interest that users want it quickly accessible. One solution that has become common in recent years has been to incorporate the information in the environment, an approach termed *ubiquitous computing* or *pervasive computing*. In these fields, non-traditional objects in the environment are used to reflect information of interest. For example, changes in lighting can correspond to the weather forecast, changes in air flow could reflect fluctuations in the stock market, and changes in ambient music could signal an upcoming meeting. A nearby person could look at or sense the change peripherally without devoting significant attention to it and certainly without having a display that occupies valuable computer screen real estate.

Many devices have been constructed that use real world objects to reflect information, but typically they are not easy to construct. This goal of the work described in this paper is to provide methods and programming interfaces that support the creation of information displays using real-world objects. We call the resulting informational displays real-world interfaces, or RWIs. Typical user interfaces use buttons, scrollbars, sliders, and similar on-screen visual displays to convey and interact with information, whereas RWIs augment or replace these displays with changes in the surrounding environment that convey information in a less direct but still noticeable manner.

Our work provides programmers with the ability to use real-world devices in much the same way that they would use a standard user interface toolkit. We have developed the Real World Interface (RWI) library, a protocol and set of application programming interfaces (APIs) for use in controlling X10 devices. The RWI library empowers programmers with the ability to create their own RWIs using a familiar API to display information using any X10 device.

The X10 protocol traditionally has been used for home automation. The protocol is most commonly found in devices that control the flow of electricity to appliances by toggling power or setting power levels. Other devices that have been built using the X10 protocol include motion detectors, cameras, thermostats, and remote controls. Signals can be sent from a computer over power lines to X10 devices within a limited range like a house or a lab.

To support multiple programming paradigms, and to understand how different paradigms are applicable in the concept of X10 devices, the RWI library supports multiple languages and toolkits. Specifically, the RWI library has been developed for use in two languages (C and C++) and in conjunction with two interface toolkits (Amulet and Tcl/Tk). We attempted to model the APIs on the paradigms used when programming in that language or toolkit. See Figure 1 for an overview of the library. With compact, simple commands using a syntax familiar to programmers, a program can be written to control the power levels to electrical devices. The end product is a library that, with appropriate off-the-shelf hardware, allows programmers to create applications for real-world devices.

We have used the RWI library in the development of several applications for home and small office use. Since real world devices reflect information in a way not necessarily obvious to outsiders, it seems that individuals and smaller communities of users, where the meaning of changes in the environment can be established more easily, would benefit most from RWIs. In turn, RWIs could help

*Author now at Microsoft, email wrighton@acm.org

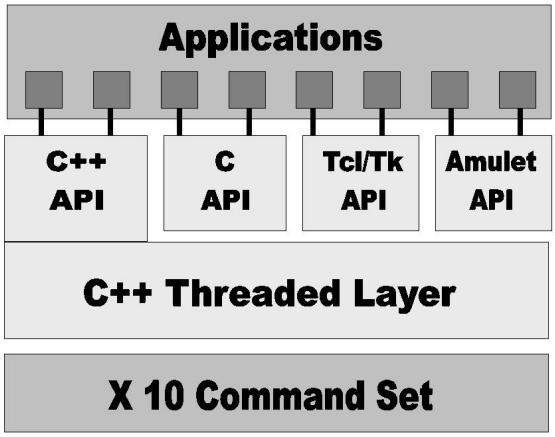

Figure 1: The RWI library architecture. At the base is the interface for interacting directly with X10 devices. On top of this is a C++ layer to support non-blocking threads of execution. At the higher levels, we created APIs that provide familiar programming interfaces for three popular languages and toolkits.

increase the sense of community by generating conversation about information they represent, including anything from the state of machines in a lab to the current weather conditions.

This paper describes the difficulties in creating RWIs using traditional means, and introduces our RWI library that addresses these difficulties. We also describe several applications built using the RWI library. Note that this paper does not provide the full API syntax and description; please visit http://www.cs.vt.edu/rwi for additional information and access to the library.

2. RELATED WORK

This work was inspired by several projects in the area of ubiquitous computing. Mark Weiser from Xerox PARC developed the notion of "calm technology" that seeks to encalm and inform simultaneously [16]. The work is illustrated by artist Natalie Jeremijenko in her "Dangling String," an attractive wire hanging from the ceiling in a hallway that reacts to signals from an Ethernet connection. A quiet network results in only occasional twitches, while a busy network causes the wire to whirl around noisily.

Others have constructed similar displays using real world objects. Hiroshi Ishii and his Tangible Media Group at MIT introduced the concept of tangible user interfaces that use physical objects as embodiments of digital information [2, 3, 9, 11, 10]. For example, a light pattern on the ceiling reflects the activity of the lab hamster, traffic noises indicate the level of network traffic, and spinning pinwheels can represent informational changes. Scott Hudson at CMU designed an information percolator that uses bubbles passing through transparent tubes to display images and patterns, thus communicating information in a pleasing yet non-intrusive manner [8]. The percolator can show presence status, short words and phrases, and other information. Greenberg and Kuzuoka developed digital but physical surrogates that represented personal proximity and presence using figurines [6]. Dan Gruen of Lotus used X10 devices to construct information displays that showed presence information of workers at remote sites [7]. This information could be reflected by the power level to any electrical device that could be plugged into a standard socket.

While these projects provide interesting theories and examples of interfaces outside the desktop, they do little to enable programmers to build their own devices. To construct most of the devices described previously one needs knowledge in electronics and circuit design, skills that typical computer scientists do not have, not to mention the average person with minimal programming skills. Only recently have researchers begun to extend toolkits to address devices in the real world. Some of the earliest examples come from context-aware computing that considers ways to support the gathering of context about users and external devices [15]. Dey et al expanded this idea by considering ways to make it easy for programmers to use this information in their Context Toolkit [4]. While the framework they establish is important to our work, the Context Toolkit supports but does not directly enable the collection and dissemination of information from the real world. The iStuff toolkit from Stanford is a collection of configurable physical objects and an underlying architecture to map real-world object events to applications [1]. The iStuff team has primarily focused on input devices, though their architecture supports output as well.

Perhaps the effort most similar to our own comes from Saul Greenberg and his phidgets project [5]. Phidgets, short for physical widgets, provide a layer of abstraction between physical devices and the programmer, allowing programmers to focus on the creation of the physical device and its inclusion in a software package. Phidgets described in the paper include a blooming flower that opens and closes its petals, a nerf emailer that shoots soft disks when new email arrives, and phidget eyes that open and close. Note that phidgets, as with the iStuff objects, still require costly or specially adapted hardware, and most of the phidget applications required some effort to be spent in the physical construction of the interaction device. Our goal is to provide a programming interface that minimizes or eliminates the need for electronics expertise. However, the similarity of this effort to our own has caused it to be very influential to our planning and framework.

3. LEVERAGING THE X10 PROTOCOL

In selecting a protocol to build on, we wanted hardware that was inexpensive and widely available, yet that could be used to control a wide variety of devices. We chose to use the X10 protocol and hardware. X10 hardware is produced by numerous companies and is available in stores and online[1]. With a set of basic pieces available for under $100, X10 hardware is fairly inexpensive, but there are a large number of devices available for controlling lamps, appliances, thermostats, cameras, motion detectors, and more.

The X10 protocol defines signals sent over power lines between X10 hardware devices. Computers send signals over a serial cable to the X10 Powerlinc adaptor, which sends and receives signals over the power lines to X10 devices. Lamps, fans, and other appliances can be plugged into X10 appliance modules, and specialized X10 devices can control cameras, motion sensors, and thermostats.

The protocol has been in use for over 20 years, primarily for security systems (programming lights to turn on when away) and convenience (remote access to lights and other devices, cameras, motion detectors, etc). The earliest X10 hardware devices were capable merely of toggling the power to lights and appliances, but current technology provides dimming and querying capabilities, and it is easier to interface with computers. Again, the primary purpose for these new capabilities was for improved security and convenience, allowing users to control and check the status of household devices from a computer.

Our approach is to support the use of X10 devices in reflecting information that is of importance to people in the surrounding envi-

[1]See www.x10.com and www.smarthome.com, for example.

Figure 2: X10 components used in constructing a RWI. The black box is the Powerlinc adaptor that allows a computer to send signals via the serial port through power lines. The white boxes are X10 receivers into which appliances can be plugged. Each receiver has one of 256 available codes.

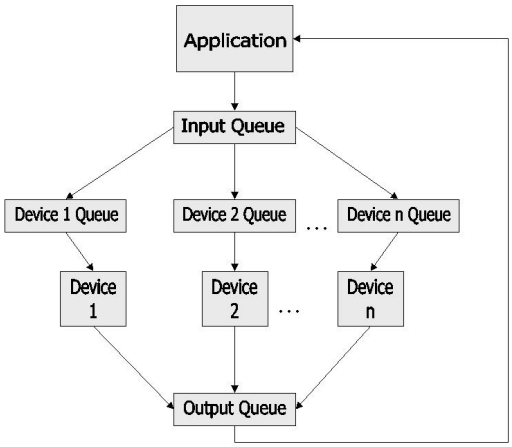

Figure 3: The C++ multithreaded support layer. X10 device messages are entered into the input queue ordered by timestamp. A helper thread dequeues input queue messages and enqueues them in the appropriate device queue. Messages are dequeued from the device queues in circular order at the appropriate times and sent to the proxy, or virtual device, which sends the message to the actual X10 device and reports success or failure.

ronment. Lights could come on, fans could blow, and heating pads could heat up in accordance with approaching meetings, changes in the weather, or traffic jams on the way home. However, the X10 protocol has a number of issues that make using X10 devices difficult, issues that often are not relevant for typical in-home use but are important when using X10 to reflect the state of information. Three key issues that we found to be important in constructing interfaces are as follows:

- **Slow information transfer.** The X10 protocol allows for a transfer rate of 30 bits per second over the power line to the device. As such, X10 is impractical for large numbers of signals sent in short time periods. To some degree, this fit with our philosophy for the design of real-world interfaces: changes in the environment should be subtle, not intrusive. While we do not see real world interfaces changing rapidly, it is still necessary to manage the flow of information to avoid exceeding the transfer rate and losing signals.

- **Minimal error detection and handling.** There is a lack of error handling, detection, and control in the X10 protocol. When multiple signals are sent in a short period of time, or if the power line is noisy, signals can be lost. When this protocol was designed, this was not a problem as typically few signals would be sent at any time and often a person would be present to notice that the protocol did not work properly. However, when incorporated into a user interface toolkit, a mechanism is needed to detect and recover from errors.

- **Blocking by the X10 computer interface.** When X10 commands are issued by the X10 computer interface, the interface blocks until a response is received. This is impractical, particularly if the access to an X10 device is part of a complex information processing program. The program should continue to operate, and when the blocked command succeeds (or fails) the program should be alerted.

While these issues are not vital for X10 as it is generally used, they are important when constructing interfaces that rely on X10.

The next section discusses how each of these issues is addressed by our Real World Interfaces library.

4. BRIDGING TO THE REAL WORLD

Our initial concern was to make it practical to construct real world interfaces. The limitations inherent to real world devices (and particularly X10 devices) described previously require significant programming to overcome, and some researchers feel that X10 in particular is simply too high-level and limited for general device development [5]. However, we have found that shielding the programmer from these issues with threading and error handling makes it possible to use X10 devices at an acceptable performance level.

To avoid blocking and to better support error checking, we developed a queue-based multithreaded layer based on the concept of passing message objects to virtual device objects that then control the X10 commands (see Figure 3). Within the C++ threaded layer, X10 commands execute in parallel with other application code. This level is based on a series of message queues that help to prevent blocking and failure by limiting the number and type of commands that are sent during any given time period.

There are three distinct types of queues: input queue, execute queue, and output queue. The *input queue* holds messages to be executed maintained in order by the time in which they are to be executed. This queue is fully accessible to programmers in that they have the ability to query the queue based upon parameters in the message, and also the ability to add and remove items (that is, schedule or cancel events) in the queue. The library's interaction with this queue is controlled by a helper thread. This helper thread moves messages from the input queue into a specific device's *execute queue* when the current time reaches the message's timestamp. This allows for multiple messages per device to be pending at any one time and forces the messages to execute in FIFO order for each device.

The messages must be passed to the devices, represented by proxies, or *virtual devices* that submit commands and query for success or failure the physical devices. By maintaining the separate device queues, if any single device is failing repeatedly, blocking for other devices will still be limited. The helper thread asks each of the virtual devices in circular order to execute one operation at a time. When each message is finished processing, the virtual device places a completion message into the *output queue*. This message is used to report success or failure of a message and can be leveraged by the APIs in several ways: a blocking wait function that returns when a new message arrives, a query state that returns a true or false to represent whether or not a new message is waiting, or a callback that invokes a function when a message is added to the queue.

Originally, X10 devices provided no method for the application to determine the state of the device, and should the device fail or have some other error there would be no indication. While we support these devices (without the error checking obviously), using the more recently designed X10 devices that support status querying is preferable. We employ this capability to check whether a command successfully executed and retry as needed. This series of required steps has a side effect that device commands take a long time to execute, with the time varying between 4 and 16 seconds. This limits the type of programs that can be created, but as we show in Section 6, there are still important classes of appropriate applications.

One important design decision made regarding this interface relates to the highly customizable functionality and features that are inherent in an extensible system. For example, when sending a message there is no framework available to enforce that messages are valid for the device where they are sent. This is because messages are based on a mapping of parameters to values. There is no enforcement that a particular parameter must exist or meet certain criteria. Additionally, when users create their own virtual devices and add them to the system, there is no way to enforce that a user-created device performs the needed actions and does not block. The final problem is that since the interface is so flexible, valuable combinations of functions may not be obvious. An API on top of this layer could help combine functionality and suggest possible actions to programmers.

Despite these limitations, the message passing solution is a significant improvement over the previous X10 interface for designing RWIs. However, it has proved awkward to use because a great deal of the mechanics of the system are still exposed. To solve this problem, we developed other APIs as layers on top of this system. The layers serve to hide message passing and separate thread of execution in several ways. The next section describes these APIs.

5. APIS FOR RWIS

In the previous section, we described a layer that supplements the standard X10 command set to make it possible to write programs that use real world objects to reflect information. In this section, we describe several application programming interfaces (APIs) that not only make it possible but furthermore simplify the job of the programmer. The APIs provide abstractions that hide many of the details of the C++ multithreaded layer using familiar programming paradigms for several languages and toolkits: C++, C, Amulet, and Tcl/Tk.

The C++ interface extends the C++ multithreaded layer with commands that support easy access to the functionality. The C interface exposes a procedural interface and the use of an opaque handle to maintain state. The Amulet interface creates an Amulet object class which can take advantage of constraints and command objects common to widgets in the Amulet system [13]. Finally, the interface for Tcl/Tk supports rapid development of interfaces with the Tcl language whereby real world objects are manipulated similar to graphical objects in the Tk toolkit. The remainder of this section focuses on the Tcl/Tk API – the most compact syntax best suited for short, focused interfaces.

Tcl/Tk is a scripting language with a powerful regular expression package, Web access capabilities, and a tightly integrated graphical user interface toolkit [14]. These characteristics make it well-suited for creating information monitoring and notification systems in which users are alerted when information changes. In addition, because Tcl/Tk is open source and extensible, it is easy to add extensions and widget packages (see, for example, [12]).

As with most scripting languages, Tcl/Tk allows programmers to develop applications quickly and with minimal overhead. For example, the simple alarm clock application described in the next section was written in only 10 lines of code. This makes it easy for people with minimal programming to develop custom monitors that reflect information of personal interest in ways that are meaningful and non-intrusive, yet informative.

In our Tcl/Tk API, real world objects are created and controlled using the same syntax as typical graphical objects. Just as one can create a scrollbar with the command `scrollbar .s` or an OK button with `button .ok`, a RWI can be created, for example for a lamp, using `rwi .lamp`. This command includes two parameters, a house code (A-P) and a unit code (1-16), both configurable using the dials on the X10 receivers (see Figure 2).

The command generates a instance of the RWI (in this example, `.lamp`), that can be instructed to turn on, off, or dim in certain cases in a similar way to how scrollbars can be set to control a listbox and buttons set to execute a command when pressed. The object can be set within the range 0 to 100, and an existing object can be queried to find its value. For example, the lamp instance can be set to 50 percent brightness using the command `.lamp set 50`, and the status of the lamp object can be queried with `.lamp get`.

The other important feature of the Tcl/Tk API is the listener, allowing implicit and explicit input from sources like remote controls, motion detectors, and wall switches to be incorporated into programs. For example, a person could walk into an area that contains a motion detector, and the motion detector would send a coded signal (using the same house and unit codes described previously) over the power line. When a coded signal is detected from a remote source, the listener, created using the command `rwi listener callback`, executes a function `callback` written by the programmer. The `callback` function has access to three variables, `%n`, `%h`, and `%u`, representing the unit name (if one exists), house code, and unit code, respectively. The listener feature provides interactive capabilities that significantly broaden the set of applications for the toolkit.

The C, C++, and Amulet toolkits provide similar functionality using procedural, object-oriented, and constraint-based approaches, respectively. For complete documentation of the syntax for all of the APIs visit at the RWI Web site at http://www.cs.vt.edu/rwi/

6. RWI APPLICATIONS

The limitations inherent to real world devices and to X10 in particular constrain the types of applications that can be designed for the RWI library. Specifically, the low information transfer rate makes displays that change frequently or in information dense ways difficult or impossible to reflect in full. One often-suggested real world interface is an intrusive display that rapidly flashes a light or changes some other visual cue to alert someone of a change in information of interest. While the low transfer rate makes this sort

of display impossible, we also argue that this interface is undesirable. If the goal is to grab a person's attention because some urgent event is taking place, a visual desktop alarm or audio warning seems ideal. If, on the other hand, the goal is to make information available, using the environment to reflect the state of information seems appropriate. When users care deeply about some specific change, they are more tolerant of intrusive interfaces that pop up on their computer screens or create an audio alert.

We share the vision of Mark Weiser that interfaces integrated into the environment should encalm, not intrude [16]. That is, the display of information should blend into the environment with minimal negative impact upon other tasks, yet it should be visible at the times when a user wants or needs it. We have created several applications to demonstrate both the functionality of the RWI library and areas where we feel real world interfaces are appropriate.

Alarm clock Perhaps the simplest application we developed was an alarm clock that dimmed a desk lamp as a meeting approached. As the meeting time approaches, the lamp dims, providing a subtle reminder of the approaching meeting. When the meeting time arrives, the lamp turns off completely, providing a more noticeable alert yet still seemingly less intrusive than typical alarms on watches or handhelds. A user trying to finish a project before leaving for the meeting can complete the project without interruption, while the off lamp provides a constant reminder of the meeting in progress.

To use the application, a user types a command `alarm 60` to set a RWI alarm for 60 minutes from the current time. Upon issuing the command, the RWI device turns on. Starting ten minutes before the indicated time, the RWI device begins to dim, and one minute before the time is up, the device turns off completely. This application was written using the Tcl/Tk RWI package in only 10 lines of code.

Meeting manager Inspired by the alarm clock idea, we considered ways to leverage people's computer-based calendars to help them manage their appointments. Handheld computers have enjoyed a burst of popularity in recent years, but their small screen sizes make their output abilities somewhat limited. By grabbing desktop synchronization data, we have developed RWIs that integrate with a calendar program to allow users to associate alarms with RWI activities. For example, as a meeting approaches, a user could simulate the behavior of the alarm clock RWI by having a lamp gradually go off, or for a more noticeable display one could cause a fan to blow progressively harder as a meeting in the calendar approaches. Again, the RWI seems less intrusive than typical alarms, and it provides a pervasive reminder that other alarms lack.

The implementation was fairly straightforward. There are several open-source packages available that provide access to synchronized information from Palm devices. This implementation used PilotLink, which allows external programs to access Palm information. Our C extension grabs alarm data from each synchronization and schedules X10 devices as specified by the user.

Weather monitor The Web provides abundant sources of changing information that could be reflected using RWIs. As our lab has no windows to the outside world, one important information source of interest is the current weather. All too often someone would leave the lab, walk down the hall, and up the stairs to the door only to discover that it was raining or the temperature had dropped significantly.

To alleviate this problem, we created a RWI that monitored the weather and used a lamp and a fan to reflect the temperature and precipitation at the time (see Figure 4). The brightness of the lamp reflected the current temperature relative to the expected temperature for the region. We situated the displays by the door of the lab, as we expected people would most need the information when they were leaving. The nature and location of this RWI minimized undesired interruptions, as users tended to focus on it only during natural breaks in their activities or when passing by them while entering or exiting the room.

Figure 4: The weather monitor RWI. The brightness of the lamp reflects the current temperature relative to the expected temperature for the region, and the fan reflects precipitation. The RWIs are situated by the door as a reminder when leaving to consider bringing a jacket or umbrella.

Computer use monitor Often it is important to maintain a sense of how much time has been spent using a computer, device, or application. For example, the RWI project group does most of its development on a single machine in our lab, and it is often interesting to speculate on how much time has been spent on development recently. We constructed a RWI using an electric clock that we reset to 12:00 at each lab meeting. Upon login to the RWI account from the console, the clock begins to run, and upon logout it stops. Someone coming in to use the computer can get a sense of how long others have been spending at the machine and could thus get a rough sense of how much work had been done. At each group meeting, we can tell how much time has been spent on development in the past week.

An important and straightforward alternative to this application would be a keyboard usage monitor. By setting the clock to 12:00 each day, it could continually reflect how much time has been spent using the keyboard during a day. We see this as highly useful for repetitive stress injury (RSI) sufferers who need to limit the time that they spend typing. The clock would provide a constant but off-the-desktop reminder of how much time has been spent during the day typing and could help an RSI sufferer limit typing, manage breaks, and plan time intervals for preventative exercises. Both of these monitors can increase the understanding of a user or group as to the use of a resource, yet neither requires users to fetch information through direct interaction. In general, the clock can act as a decorative object in the environment, and when desired it can be used to increase understanding of resource usage.

Other potential uses As we continue to develop the RWI library, we plan to construct additional RWIs for different information sources and appliances. Certainly one area of use is in showing computer behavior at both the user level and the administrator

level. RWIs could be used to show the CPU usage, battery availability, network usage, buddy-list-buddy availability, or server hits. One way to show this information would be to use an aquarium bubbler, where levels in the system would be reflected with a loud motor and lots of bubbles, thus providing audio and visual cues of the problem. As another example, the constantly changing nature of many World Wide Web sites provides many examples of information that could be monitored using RWIs, including stock prices, traffic data, news, and sports scores. In constructing RWIs, we expect that we will learn more about the needs of programmers in developing RWIs, knowledge to be used in improving the module.

7. CONCLUSIONS AND FUTURE WORK

The work described in this paper has outlined a method for controlling real world devices using APIs similar to those found in widely used languages and toolkits. Our RWI library shields a programmer from the speed and reliability problems that are common when using real world devices, particularly those that make use of the X10 protocol. In addition, our four APIs simplify the job of the programmer in creating interfaces that leverage objects in the real world. Four applications were described that have been implemented using the RWI library, and several other possible applications were described.

While this and other research efforts have addressed issues related to the representation of information using real world objects, the research area is still in its infancy with many problems remaining to be solved. An initial step will be to extend the framework to include more interactive devices like motion detectors, cameras, and remote controls, all available as X10 devices. Allowing the user to input information to the system in a "real world" way (without direct computer-based input) will enable programmers to create richer, more complex applications. As existing user interface widgets like scrollbars and buttons support interaction, we expect that the extension should be feasible and logical to programmers.

Finally, it is necessary to understand the effectiveness of real world interfaces through experimental testing. While the RWIs we have constructed have been useful and entertaining in our own offices and laboratories, experimental testing is needed to determine the usefulness and appropriateness of RWIs in a variety of situations. We suspect that there is a balance between informing users and distracting them, and by identifying key aspects of this balance, we can understand better how real world interfaces can and should be constructed and used. Our approach investigates the parameters discussed throughout this paper–interruption, reaction, and comprehension–to determine the effectiveness of RWIs in comparison to other communication techniques.

8. REFERENCES

[1] R. Ballagas, M. Ringel, M. Stone, and J. Borchers. istuff: A physical user interface toolkit for ubiquitous computing environments. In *Proceedings of the ACM Conference on Human Factors in Computing Systems (CHI 2003)*, Fort Lauderdale, FL, Apr. 2003.

[2] S. Brave, H. Ishii, and A. Dahley. Tangible interfaces for remote collaboration and communication. In *Proceedings of the ACM 1998 Conference on Computer Supported Cooperative Work (CSCW '98)*, pages 169–178, 1998.

[3] A. Dahley, C. Wisneski, and H. Ishii. Water lamp and pinwheels: Ambient projection of digital information into architectural space. In *Conference Companion of the ACM Conference on Human Factors in Computing Systems (CHI '98)*, pages 269–270, Los Angeles, CA, Apr. 1998.

[4] A. K. Dey, D. Salber, and G. Abowd. A conceptual framework and a toolkit for supporting the rapid prototyping of context-aware applications. *Human-Computer Interaction*, 16, 2001.

[5] S. Greenberg and C. Fitchett. Phidgets: Easy development of physical interfaces through physical widgets. In *Proceedings of the ACM Conference on User Interface Software and Technology (UIST '01)*, Orlando, FL, Nov. 2001.

[6] S. Greenberg and H. Kuzuoka. Using digital but physical surrogates to mediate awareness, communication and privacy in media spaces. *Personal Technologies*, 4(1), Jan. 2000.

[7] D. Gruen, S. Rohall, N. Petigara, and D. Lam. "in your space" displays for casual awareness. In *Demonstrations at the ACM Conference on Computer Supported Collaborative Work (CSCW '00)*, 2000.

[8] J. M. Heiner, S. E. Hudson, and K. Tanaka. The information percolator: Ambient information display in a decorative object. In *Proceedings of the ACM Symposium on User Interface Software and Technology (UIST '99)*, pages 141–148, Asheville, NC, Nov. 1999.

[9] H. Ishii, A. Mazalek, and J. Lee. Bottles as a minimal interface to access digital information. In *Conference Companion of the ACM Conference on Human Factors in Computing Systems (CHI 2001)*, Seattle, WA, Apr. 2001.

[10] H. Ishii and B. Ulmer. Tangible bits: Towards seamless interfaces between people, bits, and atoms. In *Proceedings of the ACM Conference on Human Factors in Computing Systems (CHI '97)*, pages 234–241, Atlanta, GA, Mar. 1997.

[11] H. Ishii, C. Wisneski, S. Brave, A. Dahley, M. Gorbet, B. Ullmer, and P. Yarin. ambientROOM: Integrating ambient media with architectural space. In *Conference Companion of the ACM Conference on Human Factors in Computing Systems (CHI '98)*, pages 173–174, 1998.

[12] D. S. McCrickard and Q. A. Zhao. Supporting information awareness using animated widgets. In *Proceedings of the 2000 USENIX Conference on Tcl/Tk (Tcl2K)*, pages 117–127, Austin, TX, Feb. 2000.

[13] B. A. Myers, R. G. McDaniel, R. C. Miller, A. S. Ferrency, A. Faulring, B. D. Kyle, A. Mickish, A. Klimovitski, and P. Doane. The amulet environment: New models for effective user interface software. *IEEE Transactions on Software Engineering*, 23(6):347–365, June 1997.

[14] J. K. Ousterhout. *Tcl and the Tk Toolkit*. Addison-Wesley, Reading, MA, 1994.

[15] B. Schilit. *System architecture for context-aware mobile computing*. PhD thesis, Columbia University, 1995.

[16] M. Weiser and J. S. Brown. Designing calm technology. *PowerGrid Journal*, 1.01, July 1996.