CHAPTER 4 AUTHORING SENSOR-BASED INTERACTIONS

Ubiquitous computing devices such as portable information appliances — mobile phones, digital cameras, and music players — are growing quickly in number and diversity. In addition, sensing technologies are becoming pervasive, for example in game controllers, and sensor hardware is increasingly diverse and economical. To arrive at usable designs for the user interfaces of such physical devices, product designers have to be able to prototype the experience of interacting with a novel hardware device. This chapter presents two systems that bring prototyping of interactions based on sensor data input within reach of interaction designers. The first section introduces d.tools, an authoring environment that combines visual authoring of application logic with a novel plug-and-play hardware platform (Figure 4.1). The second section introduces Exemplar, an extension to d.tools that enables designers to author sensor-based interaction events through programming by demonstration.

4.1 AUTHORING PHYSICAL USER INTERFACES WITH D.TOOLS

Fieldwork with professional interaction designers revealed that the creation of ubiquitous computing prototypes has remained largely out of their reach. d.tools lowers the expertise threshold and time commitment required for creating ubiquitous computing prototypes through two contributions. The first contribution is a set of interaction techniques and

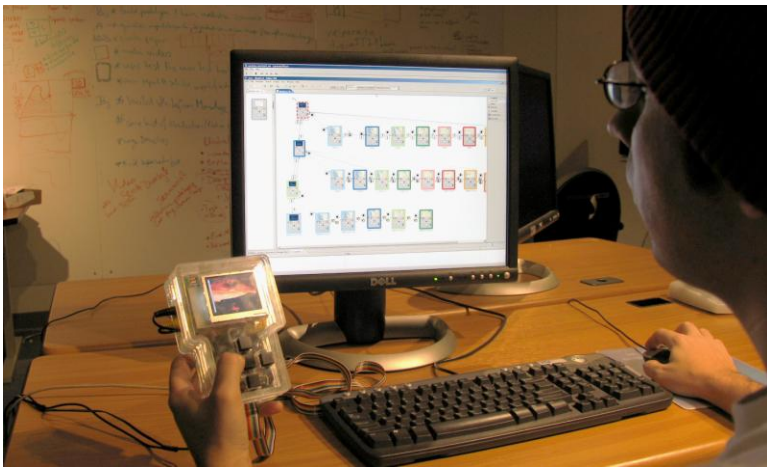

Figure 4.1: Overview of prototyping with d.tools: A designer interacts both with a hardware prototype (left) and the authoring environment (right).

architectural features that support rapid, early-stage prototyping. d.tools introduces a visual, control flow-based prototyping model that extends existing storyboard-driven design practice [126]. To provide a higher ceiling than is possible with visual programming alone, d.tools augments visual authoring with textual programming.

Second, d.tools offers an extensible architecture for physical interfaces. In this area, d.tools builds on prior work [37,43,92,93,159,185] that has shielded software developers from the intricacies of mechatronics through software encapsulation, and offers a similar set of library components. However, the d.tools hardware architecture is more flexible than prior systems by offering three extension points — at the hardware-to-PC interface, the intra-hardware communication level, and the circuit level — that enable experts to extend the library.

The rest of this section is organized as follows. We begin by outlining key findings of fieldwork that motivated our research. We then describe design principles, followed by the key interaction techniques for building, testing and analyzing prototypes that d.tools offers. We then outline implementation decisions and conclude with a report on three different strategies we have employed to evaluate d.tools.

4.1.1 FIELDWORK

To learn about opportunities for supporting iterative design of ubiquitous computing devices, we conducted individual and group interviews with eleven designers and managers at three product design consultancies in the San Francisco Bay Area, and three product design masters students. This fieldwork revealed that designing off-the-desktop interactions is not nearly as fluid as prototyping of either pure software applications or traditional physical products.

Most product designers have had at least some exposure to programming but few have fluency in programming. Design teams have access to programmers and engineers, but delegating to an intermediary slows the iterative design cycle and increases cost. Thus, while it is possible for interaction design teams to build functional physical prototypes, the cost-benefit ratio of "just getting it built" in terms of time and resources limits the use of comprehensive prototypes to late stages of their process. Comprehensive prototypes that integrate form factor (looks-like prototypes) and functions (works-like prototypes) are mostly created as expensive one-off presentation tools and milestones, but not as artifacts for reflective practice.

Interviewees reported using low-fidelity techniques to express UI flows, such as Photoshop layers, Excel spreadsheets, and sliding physical transparencies in and out of cases

(a glossy version of paper prototyping). However, they expressed their dissatisfaction with these methods since the methods often failed to convey the experience offered by the new design. In response, we designed d.tools to support rapid construction of concrete interaction sequences for experience prototyping [52] while leaving room to expand into higher-fidelity presentation models.

4.1.2 DESIGN PRINCIPLES

To guide the design of the d.tools authoring environment, we distilled the following design principles from our fieldwork observation and the general analysis of prototyping within the design process described in Chapter 2.

FAVOR CONCRETE, SPECIFIC INTERACTION SEQUENCES OVER GENERAL FUNCTIONALITY

The purpose of a UI prototype is to evoke the experience of using a future product, not to serve as an alpha version of the product. Exhibiting interactive behavior is a critical element for such prototypes, but only to the extent that it is needed to elicit the right feedback. Therefore, it is more important to rapidly build a concrete example of an interaction than to build general logic to handle different possible applications of the technique. Prototypes are concrete, narrow, and specific first; generalization and abstraction can be introduced at a later point. This guideline is a key differentiator between prototyping software and general programming tools.

MINIMIZE COGNITIVE FRICTION BETWEEN WORKING IN HARDWARE AND SOFTWARE BY BRIDGING ABSTRACTION LAYERS

When designing interactions for novel devices, more "moveable parts" exist than in traditional GUI design: the shape of the physical device, the type and layout of input and output components, and the mapping of input events to application logic have to be defined in addition to the standard concerns of interface appearance, information architecture, and behavior. To reduce some of the complexity of dealing with different levels of abstraction, d.tools introduces a device designer that serves as a virtual stand-in of the physical device being created. The goal of this device representation is to reduce the cognitive friction involved in switching between working with hardware and working with software.

OFFER IMMEDIATE, OBSERVABLE FEEDBACK ACROSS HARDWARE AND SOFTWARE

To allow the designer to experience their own design, the time between authoring a change and seeing that change, or between providing test input and observing the result, should be minimized. To this end, tight coupling between the software and hardware domains is used

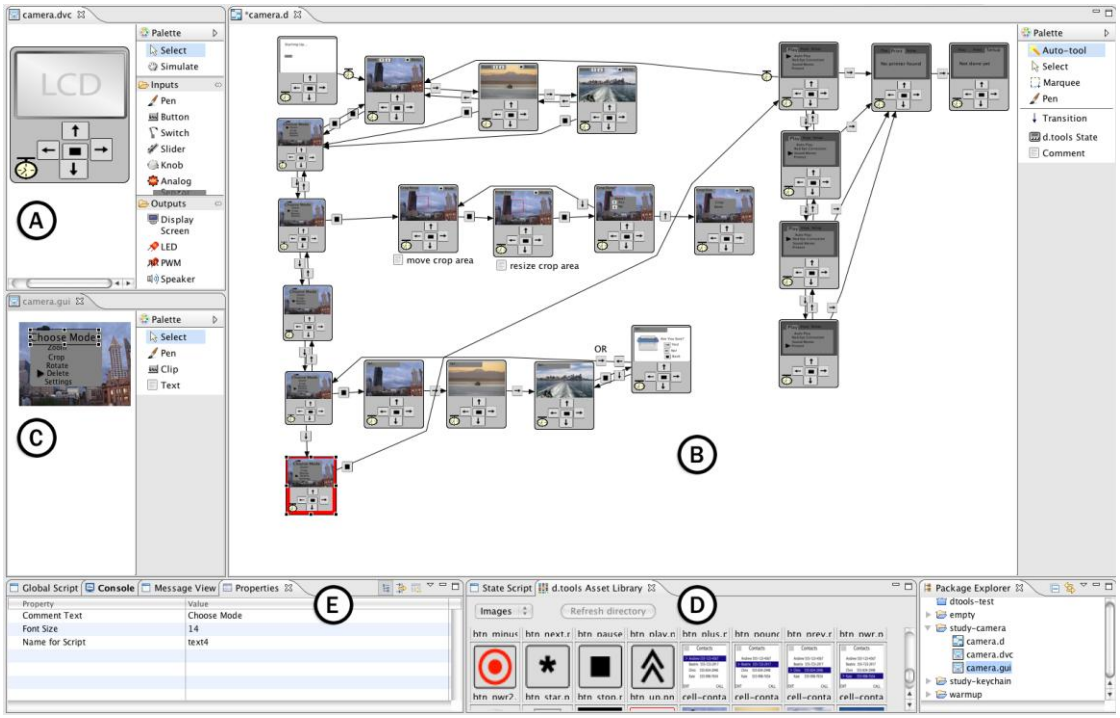

Figure 4.2: The d.tools authoring environment. **A**: device designer. **B**: storyboard editor. **C**: GUI editor. **D**: asset library. **E**: property sheet

when appropriate: an action in physical space (e.g., pressing a button) should have an immediate, observable result in the authoring environment. Vice versa, an action in the authoring environment (e.g., changing a screen graphic) should have an immediate observable result in hardware as well (e.g., show the changed graphic on an external display).

4.1.3 PROTOTYPING WITH D.TOOLS

In this section we discuss the most important interaction techniques that d.tools offers to enable the rapid design of interactive physical devices. d.tools' goal is to support design thinking rather than implementation tinkering. Using d.tools, designers place physical controllers (e.g., buttons, sliders), sensors (e.g., accelerometers, force sensitive resistors), and output devices (e.g., LEDs, LCD screens, and speakers) directly onto their physical prototypes. The d.tools library includes an extensible set of smart components that cover a wide range of input and output technologies. Software proxy objects of physical I/O components can be graphically arranged into a visual representation of the physical device (Figure 4.2A). On the PC, designers then author behavior using this representation in a visual language inspired by both storyboards and the statechart formalism [105] (Figure 4.2B). A graphical user interface editor enables composition of graphics for screen output (Figure

4.2C). Visual interaction models can be augmented by attaching code to individual states. d.tools employs a PC as a proxy for an embedded processor to prevent limitations of embedded hardware from impinging on design thinking. Designers can test their authored interactions with the device at any point in time, since their visual interaction model is always connected to the 'live' device.

4.1.3.1 Designing Physical Interactions with 'Plug and Draw'

Designers begin by plugging physical components into the d.tools hardware interface (which connects to their PC through USB) and working within the device designer of the authoring environment. When physical components are plugged in, they announce themselves to the d.tools authoring environment, creating virtual duals the device designer (Figure 4.3). Alternatively — when the physical components are not at hand or when designing interactions for a control that will be fabricated later — designers can create visual-only input and output components by dragging and dropping them from the device editor's palette. A designer can later connect the corresponding physical control or, if preferred, manipulate the behavior via Wizard of Oz [140,148] at test time.

In the device designer (Figure 4.2A), designers create, arrange and resize input and output components, specifying their appearance by selecting images from an integrated image browser, the asset library (Figure 4.2D). Building a virtual, iconic representation of the physical device affords rapid matching of software widgets with physical I/O components and reduces the cognitive friction of switching between working with hardware and working with software. The device design can also be used to simulate interaction with a device: by selecting a simulation tool from the palette, clicking (for discrete inputs) and dragging (for

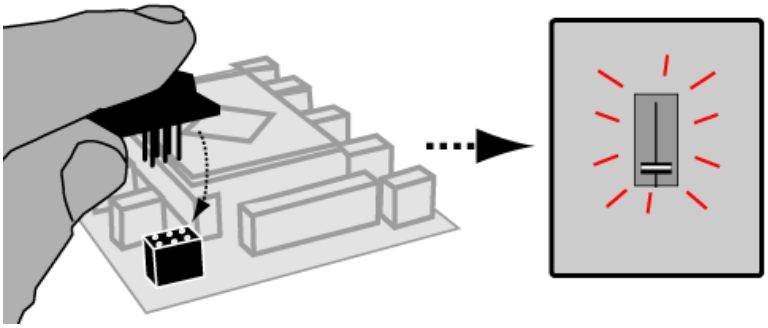

Figure 4.3: d.tools plug-and-play: inserting a physical component causes a corresponding virtual component to appear in the d.tools device designer.

continuous inputs) injects input events into the d.tools system as if the associated hardware input component had been pressed, moved, etc.

The component library available to designers comprises a diverse selection of inputs and outputs. Supported inputs include: discrete buttons and switches, rotary and linear potentiometers, rotary encoders, light sensors, accelerometers, infrared rangers, temperature sensors, force sensitive resistors, flex sensors, and RFID readers. Outputs include LCD screens, LEDs, DC motors, servo motors, and speakers. LCD and sound output are connected to the PC A/V subsystem, not our hardware interface. In addition, general purpose input and output circuit boards are available for designers who wish to build custom components. Physical and virtual components are linked through a hardware address that serves as a unique identifier of an input or output.

4.1.3.2 Authoring Interaction Models

Designers define their prototype's behavior by creating interaction diagrams in the storyboard editor (Figure 4.2B). States are graphical instances of the device design. They describe the content assigned to the outputs of the prototype at a particular point in the UI: screen images, sounds, and LED behaviors. States are created by dragging from the editor palette onto the storyboard canvas. As in the device editor, content can be assigned to output components of a state by dragging and dropping items from the asset library (Figure 4.2D) onto a component. All attributes of states, components and transitions (e.g., image filenames, event types, data ranges) can also be manipulated in text form via attribute sheets (editable tables that list attribute names and values – Figure 4.2E). To define graphic output, a graphical user interface editor provides common GUI design functionality: entering and positioning text, and loading, resizing and positioning graphical elements (Figure 4.2C). The designed graphical user interface is unique to each state.

Transitions represent the control flow of an application; they define rules for switching the currently active state in response to user input (hardware events). The currently active state is shown with a red outline. Transitions are represented graphically as arrows connecting two states. To create a transition, designers mouse over the input component which will trigger the transition and then drag onto the canvas. A target copy of the source state is created and source and target are connected. Transitions are labeled with an icon of the triggering input component (Figure 4.4A).

Conditions for state transitions can be composed using the Boolean AND/OR expressions (Figure 4.4B). A single Boolean connective is applied to all conditions on a

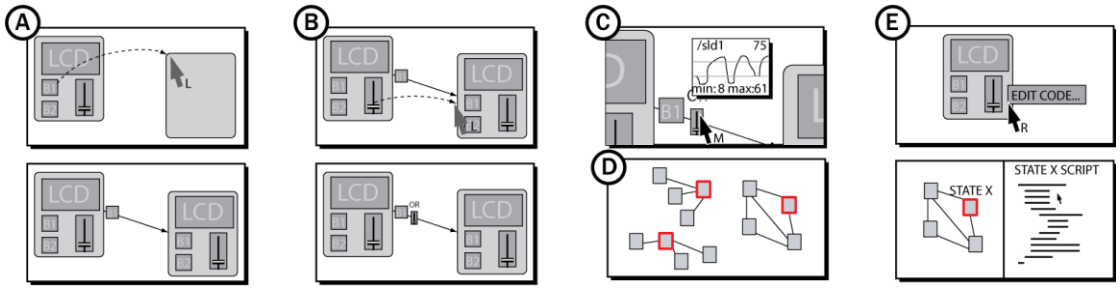

Figure 4.4: d.tools interaction techniques. **A**: creating new transitions through dragging. **B**: adding a new condition to an existing transition. **C**: Visualizing sensor signal input and thresholds in context. **D**: parallel active states. **E**: editing code attached to a state.

transition arrow, as complex Boolean expressions are error prone. Boolean combinations allow authoring conditionals such as 'transition if the accelerometer is tilted to the right, but only if the tilt-enable button is held down simultaneously.' More complex conditionals can be authored by introducing additional states.

To define discrete events for continuous sensors, designers define upper and lower thresholds for a sensor's value. Whenever the sensor value transitions into the threshold region, a transition event is generated. To help designers visualize such sensor thresholds, a graph showing both recent sensor history and threshold lines can be displayed on demand above the transition arrow utilizing the event (Figure 4.4C).

Within the visual editor, timers can be added as input components to a device to create automatic transitions or (connected with AND to a sensor input) to require a certain amount of time to pass before acting on input data. Automatic transitions are useful for sequencing output behaviors, and timeouts have proven valuable as a hysteresis mechanism to prevent noisy sensor input from inducing rapid oscillation between states.

While the storyboard aids a designer's understanding of the overall control flow of the prototype, complex designs still benefit from explanation. d.tools supports commenting with text notes that can be freely placed on the storyboard canvas.

4.1.3.3 Raising the Complexity Ceiling of Prototypes

The state-based visual programming model embodied in d.tools enables rapid design of the information architecture of prototypes, but the complexity of the control flow and interactive behavior that can be authored is limited. To support refining designs and permit higher-fidelity behaviors, d.tools provides two mechanisms that enable more complex interactions: parallel states and code extensions.

PARALLEL STATES

Expressing parallelism in single point-of-control state diagrams results in an exponentially growing number of states. Our first-use study also showed that expressing parallelism via cross products of states is not an intuitive authoring technique. To support authoring parallel, independent functionality, multiple states in d.tools can be active concurrently in independent sub-graphs (e.g., the power button can always be used to turn the device off, regardless of the other state of the model – Figure 4.4D). One limitation of parallel states in d.tools is that the system currently lacks an explicit mechanism to define what behavior should occur when two states try to assign output to the same component simultaneously.

ATTACHING CODE

To specify behaviors that are beyond the capability of the visual language (e.g., dynamically generating animations tied to user input), designers can attach textual code to visual states. The right-click context menu for states offers actions to edit and hook or unhook code for each state (Figure 4.7E). A d.tools API provides read and write access to hardware components, and allows procedural animation of graphics objects on screen. We implemented two different alternatives of d.tools code extensions — one with compiled Java classes, and one with interactively interpreted Java — to explore the tradeoffs of mixing visual and textual programming.

The compiled Java extension leverages the Eclipse programming environment's rich Java editing functionality. When users right-click on a visual state and choose the edit code command, d.tools generates a skeleton Java class file and launches the native Eclipse Java editor, which provides auto-completion, syntax highlighting, and integrated help. The primary benefit of this path is that is offers Eclipse's mature toolset. However, the toolset also brings with it a steep learning curve and a discontinuous authoring experience for two reasons. First, Eclipse targets professional software engineers and favors generality and completeness; many of the UI options offered are irrelevant to the more narrowly scoped task of writing state code in d.tools. Second, Java is a very verbose, strongly & statically typed, object-oriented language. The combination of these features requires designers to fully understand the object oriented development paradigm to make use of d.tools code extension — a barrier that proved too high in conversations with our target users.

In response to the identified complexity challenge, later versions of d.tools replaced the compiled Java architecture with an interactive interpreter which supports standard Java syntax but also offers "syntactic sugar" which results in much more concise code that focuses

on expressing the intended logic. The scripted Java extension trades off more concise and structurally simpler code against limited editor support for detecting syntax errors, suggesting corrections, and debugging.

EXECUTING INTERACTION MODELS AT DESIGN TIME

Designers can execute interaction models in three ways. First, they can manipulate the attached hardware. Second, they can imitate hardware events within the software workbench by using a simulation tool. Clicking on input components with the simulation tool then generate synthetic input events, e.g., button presses and release events, that are used to drive the interaction model as if real hardware input events had been received. Third, designers can employ a Wizard of Oz approach where they observe a user interacting with the prototype, and manually change the active state in the editor with their mouse.

4.1.4 ARCHITECTURE AND IMPLEMENTATION

Implementation choices for d.tools hardware and software emphasize both a low threshold for initial use and extensibility through modularity at architectural seams. In this section we describe how these design concerns and extensibility goals are reflected in the d.tools architecture.

4.1.4.1 Plug-and-Play Hardware

d.tools contributes a plug-and-play hardware platform that enables tracking identity and presence of smart hardware components for plug-and-play operation. I/O components for low-bandwidth data use a common physical connector format so designers do not have to worry about which plugs go where. Smart components each have a dedicated small microcontroller; an interface board coordinates communication between components and a

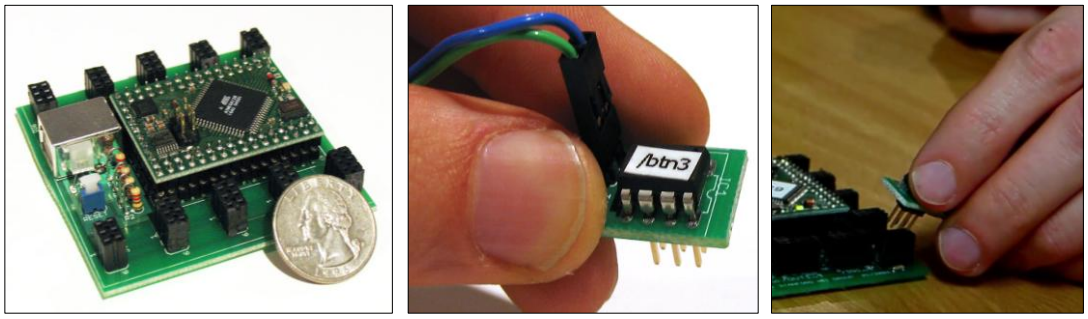

Figure 4.5: The d.tools hardware interface (left). Individual smart components (middle) are can be plugged into any bus connector (right).

PC (Figure 4.5). Components plug into the interface board to talk on a common I2C serial bus [32] (Figure 4.6). The I2C bus abstracts electrical characteristics of different kinds of components, affording the use of common connectors. The interface board acts as the bus master and components act as I2C slaves. A USB connection to the host computer provides power and the physical communication layer.

Atmel microcontrollers are used to implement this architecture because of their low cost, high performance, and programmability in C. The hardware platform is based around the Atmel ATmega128 microcontroller [22] on a Crumb128 development board from chip45 [172]. I/O components use Atmel ATtiny45 microcontrollers [23]. Programs for these chips were compiled using the open source WinAVR tool chain and the IAR Embedded Workbench compiler. Circuit boards were designed in CADsoft Eagle, manufactured by Advanced Circuits and hand-soldered.

d.tools distinguishes audio and video from lower-bandwidth components (buttons, sliders, LEDs, etc.). The modern PC A/V systems already provide plug-and-play support for audio and video; for these components d.tools uses the existing infrastructure. For graphics display on the small screens commonly found in information appliances, d.tools includes LCD displays which can be connected to a PC graphics card with video output. This screen is controlled by a secondary video card connected to a video signal converter. Displays that receive both graphics commands and power through a single USB connection are also becoming available and can be substituted.

4.1.4.2 Hardware Extensibility

Fixed libraries limit the complexity ceiling of what can be built with a tool by knowledgeable

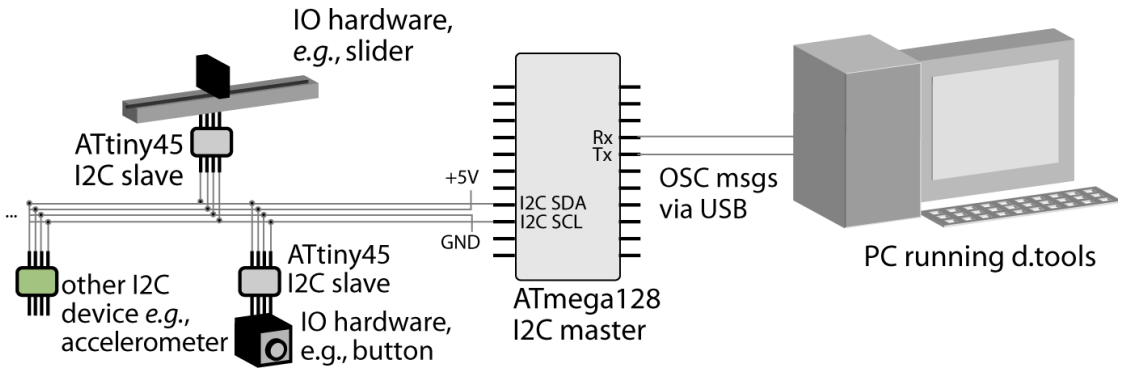

Figure 4.6: Schematic diagram of the d.tools hardware infrastructure. Smart components are networked on an I2C bus. A master microcontroller communicates over a serial-over-USB connection with the computer running the d.tools authoring environment.

users. While GUIs have converged on a small number of widgets that cover the design space, no such set exists for physical UIs because of the greater variety of possible interactions in the real world. Hence, extending the library beyond what 'comes with the box' is an important concern. In the d.tools software, extensibility is provided by its Java hooks. In the d.tools hardware architecture (Figure 4.6) extensibility is offered at three points: the hardware-to-PC interface, the hardware communication level, and the electronic circuit. This allows experts with sufficient interest and skill to modify d.tools to suit their needs.

d.tools hardware and a PC communicate by exchanging OpenSoundControl (OSC) messages [256] over a USB serial connection. OSC was chosen for its open source API, existing hardware and software support, and human readable addressing format (components have path-like addresses — e.g., buttons in d.tools are labeled `/btn1` or `/btn6`.) By substituting devices that can produce OSC messages or software that can consume them, d.tools components can be integrated into different workflows. For example, music synthesis programs such as Max/MSP [18] can receive sensor input from d.tools hardware. Connecting other physical UI toolkits to d.tools involves developing an OSC wrapper. As a proof of concept, we have written such a wrapper to connect Phidgets InterfaceKits [93].

Developers can extend the library of smart I/O components by adding components that are compatible with the industry standard I2C serial communication protocol. I2C offers a large base of existing compatible hardware. For example, the accelerometers used in d.tools projects are third party products that send orientation to d.tools via on-board analog-to-digital converters. Presently, adding new I2C devices requires editing of source code for the master microcontroller; this configuration step could also be pushed up to the d.tools authoring environment.

On the circuit level, d.tools can make use of inputs that vary in voltage or resistance and drive outputs with on/off control and pulse width modulation. This allows designers versed in circuit design to integrate new sensing and actuation technologies at the lowest level. This level of expansion is shared with other hardware platforms that offer direct pin access to digital I/O lines and A2D converters.

4.1.4.3 Software

To leverage the benefits of a modern IDE, d.tools was implemented in Sun's Java JDK 5 as a plug-in for the open-source Eclipse platform. Its visual editors are fully integrated into the Eclipse development environment [21]. d.tools uses the Eclipse Graphical Editing Framework (GEF) for graphics handling [24]. d.tools file I/O is done via serialization to XML, which enables source control of project files in ASCII format using version control tools. The design environment is platform independent except for "glue" code for USB port communication, and has been tested under Windows and Mac OS X.

CODE EXTENSIONS

The *compiled Java code extension* leverages the Eclipse programming environment's rich Java editing and compilation functionality. Eclipse automatically compiles the user's code into class files. d.tools, on entering a new state, uses a custom Java class loader to search for any new class files for the current state, and, if found, instantiates the class and calls its API methods. The initial d.tools Java API is reproduced in Table 4.1.

The subsequently developed *scripted code extension* model builds on BeanShell, a Java interpreter [200]. The interpreter's namespace is populated by d.tools with objects matching both hardware I/O components and graphics components defined in each state. For example, if a button with the name "upButton" exists in the device designer, then a variable corresponding to a Button object with name "upButton" will be present in the interpreter. Similarly, if a screen graphic object with the name "menuGraphic" is defined in a particular state, then a corresponding object with name "menuGraphic" will be accessible in the script

Function	Description
`enterState()`	Is called when the code's associated state receives focus in the statechart graph.
`update(String component, Object newValue)`	Is called when a new input event is received while the code's state has focus. The component's hardware address (*e.g.*, "/btn5" for a button) is passed in as an identifier along with the updated value (Booleans for discrete inputs, Floats for continuous inputs, and Strings for RFID tags).
`getInput(String component)`	Queries the current value of an input.
`setOutput(String component, Object newValue)`	Controls output components. LCD screens and speakers receive file URLs, and LEDs and general output components Booleans for on/off.
`println(String msg)`	Outputs a message to a dedicated debug view in our editor.
`keyPress(KeyEvent e)` `keyRelease(KeyEvent e)`	Inserts keyboard events into the system's input queue (using Java Robots) to remote control external applications.

Table 4.1: The d.tools Java API allows designers to extend visual states with source code. The listed functions serve as the interface between designers' code and the d.tools runtime system. Standard Java classes are also accessible.

of that state. The technique of creating objects based on name properties entered in a direct manipulation interface is also present in other GUI editors such as Adobe Flash.

For programming animations, the interpreter uses a polling method for calling user-defined graphics routines. If the user defines a function called loop(), that function is called repeatedly at 30Hz, a rate sufficient for generating animations. This polling technique is conceptually more straightforward than event callbacks and was inspired by its successful use in the end-user graphics programming environment Processing [210].

A challenge we noted in the compiled Java model was that persisting data for sharing between different states was cumbersome. To facilitate defining globally accessible variables and functions, the scripted Java extension therefore added the concept of a "global script" in addition to individual state scripts. Variables and functions declared in the global script are accessible to all state scripts. Global scripts are reloaded whenever the project files are saved. State-specific scripts are executed whenever the associated graphical state becomes active. The complete d.tools scripting API is reproduced in Table 4.2 and some examples of the API in use are given in Figure 4.7

```
//working with text objects
text1.setText("Hello");
text1.setText(text1.getText()+", World!");
text1.setFontSize(24);

//resize the image "clipImg" based on
//two button events
loop() {

  //scale down
  if(btnDown.getValue())
    clipImg.setScale(clipImg.getScale()-5);

  //scale up
  if(btnUp.getValue()) {
    clipImg.setScale(clipImg.getScale()+5);

  //center image on stage
  clipImg.setXY(stage.getWidth()/2-
                clipImg.getWidth()/2,
                stage.getHeight()/2-
                clipImg.getHeight()/2);
}
```

Figure 4.7: Code examples for the d.tools scripting API.

Global Functions for Drawing and Accessing Hardware	
Function	**Description**
`void loop()`	If the user's state script defines a loop function, the function will be called repeatedly at interactive rates while the given state is active.
`void print(String msg)`	Print a message to the debug console.
`boolean ditigalRead(String compName)` `void digitalWrite (String compName, boolean value)`	Read the last known state of a discrete input component such as a button or switch with identifier compName, Write a new value.
`float analogRead (String compName)` `void analogWrite (String compName, float value)`	Read the last known value of a continuous input. Write to a pulse-width modulated output component such as a PWM LED
Hardware Component Proxy Objects	
`component.setValue(boolean v)` `component.setValue(float v)`	Set the state of the component named "component"; overloaded based on component type (discrete or PWM output). Corresponds to digitalWrite() and analogWrite() above.

`boolean component.getValue()`
`float component.getValue()`
`int component.getValue()`
	Read the last known state of the component named "component"; overloaded based on component type (discrete, continuous or identity-reporting). Corresponds to digitalRead() and analogRead() above.

GUI Objects: Stage	
`int getWidth(), int getHeight()`	Return the width and height of stage, as defined in the device designer.
`void setColor (int r, int g, int b)` `int getColor(String which)`	Set/get the stage color.
GUI Objects: Graphic Clips	

`void setWidth(int w)`
`void setHeight(int h)`
`int getWidth(), int getHeight()`
	Set/get the width and height of the clip object.

`int getX(), int getY()` `void setX(int x), void setY(int y)`	Set/get the position of the clip object.
`setVisible(boolean v)` `boolean isVisible()`	Set/get the visibility of the clip object.
`setImage(String filename)` `String getImage()`	Set/get the image displayed by this clip object.
GUI Objects: Text	
`setX(int x), setY(int y),` `int getX(), int getY()`	Set/get the position of the text object.
`setVisible(Boolean v)` `boolean isVisible()`	Set/get the visibility of the text object.
`setText(String text)` `String getText()`	Set/get the string displayed by the text object.
`setFontSize(int size)` `int getFontSize()`	Set/get the text font size.

Table 4.2: The d.tools scripting API provides both global and object-oriented functions to interact with hardware, and a concise object-oriented set of function for manipulating GUI elements.

4.1.5 EVALUATION

In this section, we outline the methodological triangulation we employed to evaluate and iteratively refine d.tools. First, to ascertain the expertise threshold of d.tools, we conducted a first-use lab study with thirteen design students and professional designers. Second, the author and other members of our research group rebuilt prototypes of existing devices and used d.tools in two research projects. Third, we made d.tools hardware kits available to students in a project-centric interaction design course at our university.

4.1.5.1 Establishing Threshold with a First Use Study

We conducted a controlled laboratory study of d.tools to assess the ease of use of our tool; the study group comprised 13 participants (6 male, 7 female) who had general design experience. Three participants served as pilot testers to refine the testing protocol. Participants were given three design tasks of increasing scope to complete with d.tools within 90 minutes. Most participants were students or alumni of design-related graduate programs at our university.

Sessions started with a demonstration of the d.tools software editor and the hardware components by the experimenters. We then gave participants two narrowly defined tasks and one open-ended design project. For the first task, participants were asked to complete a menu navigation design that the experimenter had started during the demonstration. For the second task, participants were asked to build a functional physical prototype of a device with one button and one switch as inputs, and one LED and a speaker as outputs. Pressing the button should play a sound clip and toggling the switch should turn the LED on or off. The two components were to function independently of each other.

The third assignment was to begin prototyping a digital music player for children. Participants were given written guidelines such as "children prefer dedicated controls and like elements that move better than buttons." As the study allotted only 30 to 45 minutes for this part, participants were informed that they were not expected to produce a finished product. To sketch and build physical prototypes, we provided an 18" × 24" paper pad, sheets of foam core, pens, a selection of tools, glue and tape, and a label printer. As the final step of the study, participants were asked to complete a 26 question survey.

STUDY RESULTS

All participants successfully completed both close-ended tasks, regardless of prior experience in user interface design or physical computing. Task one took a mean of 9 minutes while task two took a mean of 24 minutes to complete (Figure 4.8).

For the music player design task, participants followed heterogeneous approaches: some started by exploring the ergonomics of different shapes to determine input component placement; others focused on requirements analysis on paper; yet others worked exclusively in software. d.tools was most frequently used for determining layout of interaction components in the device designer, and reasoning about the interaction model in the storyboard designer. Two participants with prior physical computing experience built functional physical prototypes with navigation and sound playback in less than 30 minutes.

SUCCESS OF A LOW THRESHOLD AND TIGHT COUPLING

Almost all users commented positively on the tight coupling of hardware components and their software counterparts, especially the automatic recognition of hardware connections. Authoring storyboards through link-and-create actions was immediately intuitive. Refining default behaviors through text properties and expressing functional independence in a

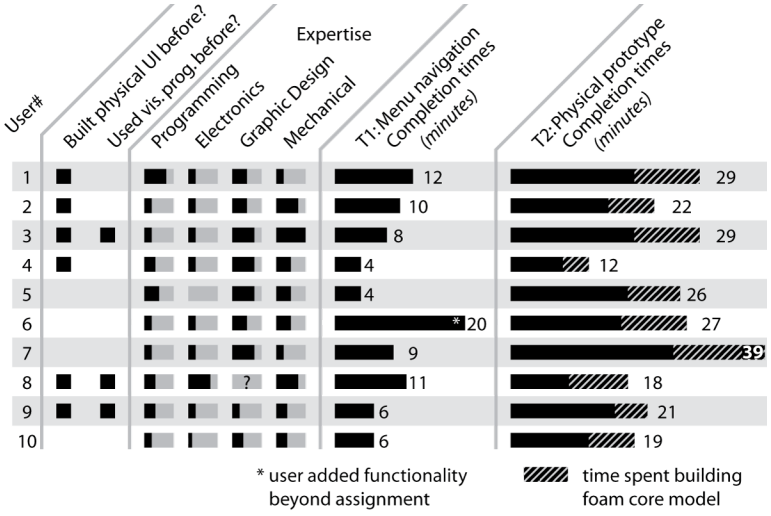

Figure 4.8: Task completion times, and prior experience and expertise of d.tools study participants. Participants completed task 1 in an average of 9 minutes, and task 2 in an average of 24 minutes. These times demonstrate that prototyping with d.tools is fast enough to be appropriate for early-stage design.

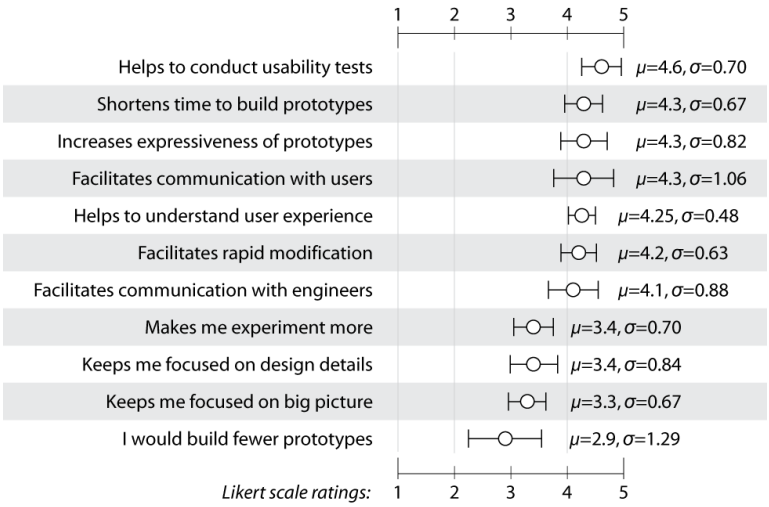

Figure 4.9: Post-test survey results from the d.tools user study. Participants provided responses on Likert scales.

storyboard took longer; nevertheless, participants mastered these strategies by the end of the session.

After an initial period of learning the d.tools interface, participants spent much of their time with *design thinking*—reasoning about how their interface should behave from the user's point of view instead of wondering about how to implement a particular behavior. This was especially true for authoring UI navigation flows.

The experimenter asked participants to hand over the devices built for the second task to test whether the required functionality had been achieved —while observing this on-the-spot user test, many subjects expressed the wish to iterate on their designs and produced another version two to ten minutes later. This suggests the advantage of the rapid iteration cycles that d.tools enables. In a post-test survey (Figure 4.9), participants consistently gave d.tools high marks for enabling usability testing (μ=4.6 on 5 point Likert scale), shortening the time required to build a prototype (μ=4.3), and helping to understand the user experience at design time (μ=4.25).

NEEDS: SOFTWARE SIMULATION, LARGER LIBRARY, RICHER FEEDBACK

One significant shortcoming discovered through the study was the lack of software simulation of an interaction model: the evaluated version did not provide a mechanism for stepping though an interaction without attached hardware. . This prompted the addition of our software simulation mode. Specifying sensor parameters textually worked well for subjects who had some comfort level with programming, but were judged disruptive of the

visual workflow by others. Interaction techniques for graphically specifying sensor ranges were added to address this issue. Users also wished for aggregate inputs that have become standard navigation elements for information appliances such as combined up/down buttons, five-way joysticks, and keypads.

4.1.5.2 Rebuilt Existing and Novel Devices

To evaluate the expressiveness of d.tools' visual language, we recreated prototypes for four existing devices — an Apple iPod Shuffle music player, the back panel of a Casio EX-Z40 digital camera (Figure 4.10A), Hinckley et al.'s Sensing PDA [121] (Figure 4.10B), and Partridge et al.'s TiltType [202] text entry device. The iPod Shuffle is a digital music player without a screen where all playback options are controlled through tactile switches. For the digital camera, we prototyped image review mode, where users can navigate through images, zoom, crop, and delete images. The Sensing PDA uses an accelerometer to detect device pose and adjust display orientation accordingly. An infrared distance sensor can detect whether the device is held close to a user's face; a force-sensitive touch sensor detects whether the device is held. The TiltType device uses a dual-axis accelerometer in conjunction with momentary switches under the user's index fingers to explore orientation-based text entry techniques. We distilled the central functionality of each device and prototyped these key interaction paths.

Additionally, two research projects in our group used d.tools to provide physical input for table and wall interfaces. The *tangible drawers* project explored physical drawers as a file access metaphor for a shared tabletop display [112]. The author built four drawer mechanisms mounted underneath the sides of a DiamondTouch interactive table (Figure 4.10C, Figure 4.10D). Opening and closing these drawers controlled display of personal data collections, and knobs on the drawers allowed users to scroll through their data. Ju et al. used d.tools to explore proxemics for interactive whiteboards through an array of infrared distance sensors mounted to the frame of a wall display [137] (Figure 4.10E).

From these exercises, we learned that interactive physical prototypes have two scaling concerns: the complexity of the software model, and the physical size of the prototype. d.tools diagrams of up to 50 states are visually understandable on a desktop display (1920 × 1200); this scale is sufficient for the primary interaction flows of current devices. Positioning and resizing affords effective visual clustering of subsections according to gestalt principles of proximity and similarity. However, increasing transition density makes maintaining and troubleshooting diagrams taxing, a limitation shared by other visual authoring environments.

Figure 4.10: Some applications built with d.tools in our research group. **A**: digital camera image navigation. **B**: sensor-enhanced smart PDA. **C** & **D**: tangible drawers for a multi-user interactive tabletop. **E**: proxemics-aware whiteboard. **F**: TiltType for orientation-based text entry.

In the tangible drawers project, the presence of multiple independent drawers prompted the need for multiple concurrently active states. This project as well as Range also required sensor data access from an existing Java application. d.tools can interact with existing applications in one of two ways: state change information and raw sensor data can be received by a 3rd party application using socket communication; or d.tools can inject mouse and keyboard events into the operating system event queue, a technique termed *screen poking* (similarly to Hudson's Thumbtacks project [127]). The first method was used to interface with the our other research project code bases; it raises the question which parts of the interaction should be authored in d.tools, and which parts in the external application's source code. Screen poking was used by the author to prototype accelerometer-based zoom and pan control for the Google Earth application in less than 30 minutes. However, screen poking is a brittle technique as d.tools is unaware of the internal state of the controlled application; it is therefore less useful for more complex prototypes.

The first author also served as a physical prototyping consultant to a prominent design firm. Because of a focus on client presentation, the design team was primarily concerned with the polish of their prototype — hence, they asked for integration with Flash. From a research standpoint, this suggests — for "shiny prototypes" — a tool integrating the visual richness of Flash with the computational representation and hardware abstractions of d.tools.

4.1.5.3 Teaching Experiences — HCI Design Studio

We deployed the d.tools hardware and software to student project teams in a master's level HCI design course at Stanford [150]. Students had the option of using d.tools (among other technologies) for their final project, the design of a tangible interface. Seven of twelve groups used d.tools. In the following year, d.tools was offered again to students in the class. In this real-world deployment, we provided technical assistance and tracked usability problems, bug reports, and feature requests. Figure 4.11 provides an overview of some projects built by students.

SUCCESSES

Students successfully built a range of innovative interfaces. Examples include a wearable watch that allows children to record and trade secret audio messages, a color mixing interface in which children can "pour" color from tangible buckets onto an LCD screen, and an augmented clothes rack that offers product comparisons and recommendations via hanger sensors and built-in lights.

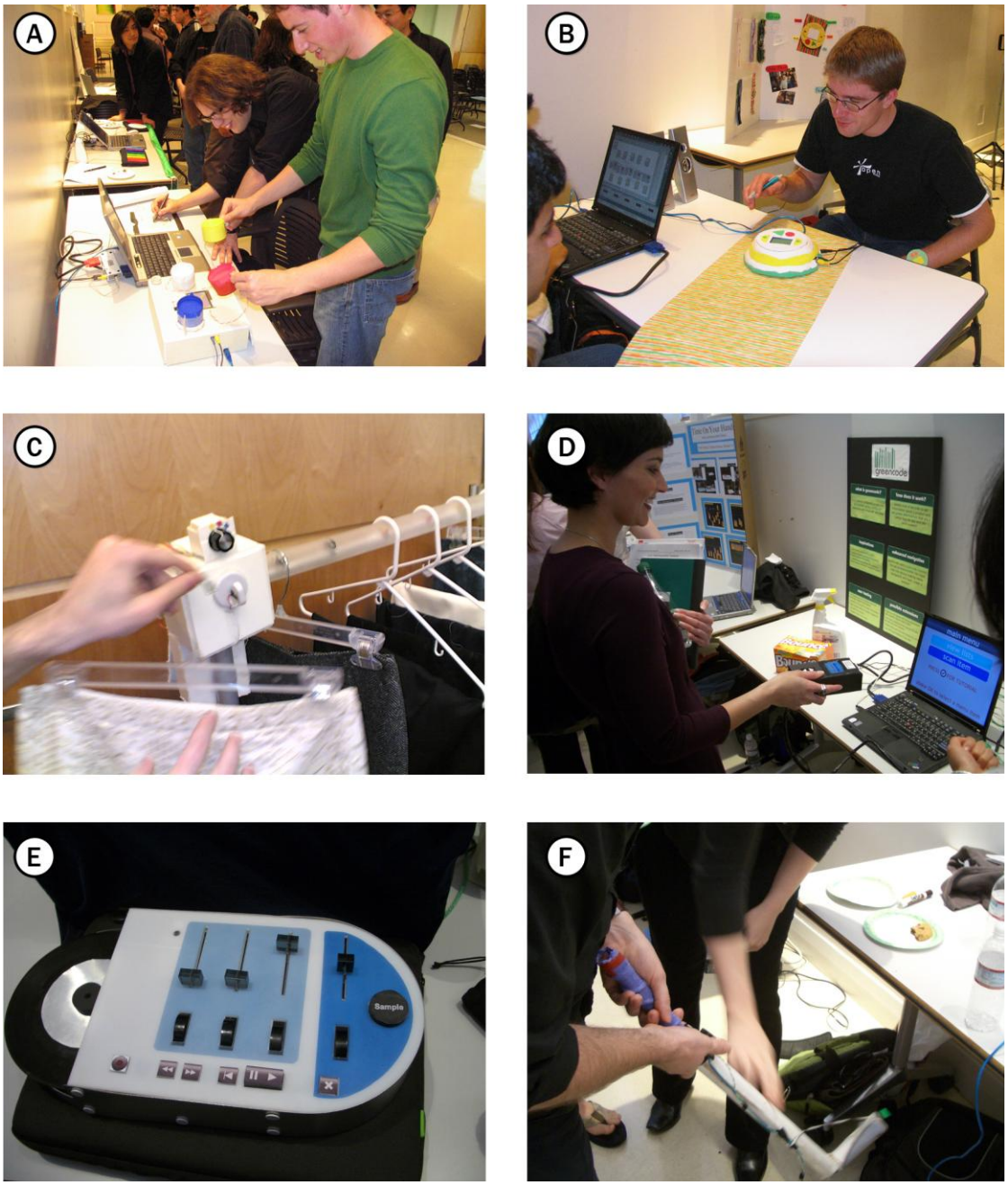

Figure 4.11: Some student projects built with d.tools. **A**: a tangible color mixing device where virtual color can be poured from physical paint buckets by tilting them over an LCD screen. **B**: a message recording system for children to exchange secrets. **C**: a smart clothes rack can detect which hangers are removed from the rack and display fashion advice on a nearby screen. **D**: a mobile shopping assistant can scan barcodes of grocery items and present sustainability information relating to the scanned item on its screen. **E**: a tangible audio mixer to produce cell phone ring tones. **F**: an accelerometer-equipped golf club used as a game controller.

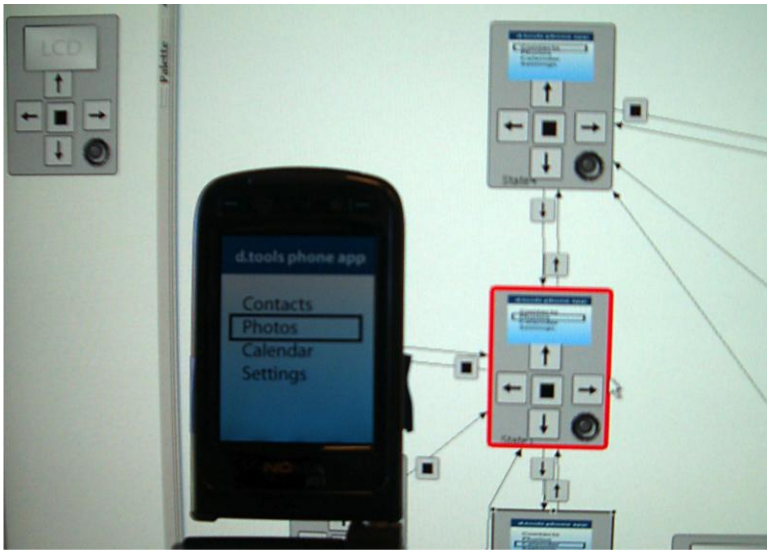

Figure 4.12: A d.tools mobile prototype on a Nokia N93 smart phone, with the storyboard logic of the prototype in the background.

Students were able to work with supplied components and extend d.tools with sensor input not in the included library. For example, the color mixing group integrated mechanical tilt switches and vibration motors into their project.

SHORTCOMINGS DISCOVERED

Remote control of third party applications (especially Flash) was a major concern for students — in fact, because d.tools did not have a graphical user interface editor in the supplied version, two student groups chose to develop their project with Phidgets [93], as it offers a Flash API. To address this need, we first released a Java API for the d.tools hardware with similar connectivity. We observed that student groups that used solely textual APIs ended up writing long-winded state machine representations using switch or nested conditional statements; the structure of their code could have been more concisely captured in our visual language. The need for direct control over GUI graphics also motivated the later addition of the d.tools graphical user interface editor.

4.1.6 D.TOOLS MOBILE

The d.tools architecture was designed to focus on prototypes that involve custom hardware. Might it also offer benefits for prototyping interfaces for commodity hardware, such as smart phones? To understand the utility of d.tools for mobile interaction design, we collaborated with Nokia to enable real-time input from and output to smart phones (Figure 4.12).

With the d.tools mobile system, designers author functionality in the standard visual environment. Designers do not need to create a device definition; they can load a pre-created model that matches layout of phone input keys and screen. For running and testing such prototypes, a custom d.tools client application is loaded onto a phone. This client intercepts all input events (i.e., key presses) and sends them over a wireless connection to the PC running d.tools, where they are used to trigger state transitions. Output commands resulting from state transitions are then sent to the phone to display graphics or play sounds (Figure 4.13). In essence, the phone is turned into a terminal, while all application logic executes in the d.tools authoring environment. Our current implementation was written in Python for Nokia S60 phones. We are using a Wi-Fi connection for message passing. Messages are sent as OpenSoundControl packets over UDP.

BENEFITS

We have tested the d.tools mobile approach informally in our lab and with collaborators at Nokia. A primary benefit of our approach is that it sidesteps many of the pain points of developing and deploying prototypes on phones, since development and execution both remain on the PC. In addition, the state of the phone application can be monitored in d.tools. It is also possible to change the interaction logic in the middle of a test. d.tools mobile is especially suited for quick exploration of applications with relatively static individual screens — a storyboard can be assembled in a few minutes and tested on the target device. Because of the reliance on a common messaging protocol, OSC, it is also possible to add external sensors connected to a d.tools hardware interface and explore interactions that rely on sensor input not provided by the phone itself.

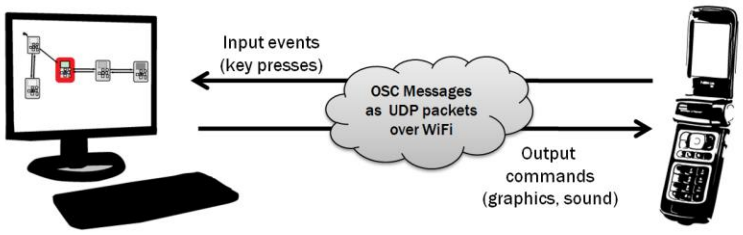

Figure 4.13: The d.tools mobile system architecture uses socket communication over a wireless connection to receive input events and send output commands to a smart phone.

LIMITATIONS

d.tools mobile also has multiple important limitations, some fundamental to its execution model, others merely due to its nature as research software. Fundamentally, interactivity is limited by the roundtrip latency of sending an event to d.tools, and receiving a message with output commands in return. Mobile devices have to trade off network latency and battery life, and as a result, we observed roundtrip latencies of 200-1000 milliseconds, with a large amount of jitter (the variation in latency). While fast enough for discrete control tasks such as navigating from screen to screen, d.tools mobile is not fast enough for continuous control tasks such as smooth panning or zooming. Latency and jitter will further increase if users try to take the device out of the lab and switch from a wireless Ethernet connection to a cellular data connection. This limits the applicability of d.tools mobile for testing outside the lab.

An important pragmatic limitation of our implementation is that the d.tools scripting language has not been ported to d.tools mobile yet. Thus dynamic behaviors cannot yet be implemented. Adding script execution is not trivial as it requires deciding which commands should be executed on the phone itself, and which commands should be executed on the PC. For example, graphics commands such as translation, rotation, and scaling are best executed on the phone itself so large graphics files don't have to be transmitted. Also, only keyboard input is currently supported. Although processing data from built-in phone sensors is certainly possible, the appropriate modules exposing such data to the Python programming language were not available to us.

To understand the relative benefits and limitations of d.tools mobile, we compare design decisions of d.tools mobile and important related work in Table 4.3. Ballagas' iStuff Mobile [42] is the most related system, as it also executes logic on a PC. iStuff mobile targets higher-fidelity development of mobile applications where phones are one among multiple devices in a ubiquitous computing ecology, while d.tools mobile targets lower-fidelity UI walkthroughs.

	d.tools mobile	iStuff mobile	Flash Lite	Juxtapose mobile
Authoring Environment	Visual (d.tools state diagrams)	Visual + Code (Quarts Composer + JavaScript)	Visual + Code (Adobe Flash IDE + ActionScript)	Code (ActionScript)
Where does computation happen?	PC	PC	Phone	Phone
Supported Input	Phone keyboard, external sensors	Phone keyboard, external sensors	Phone hardware only	Phone hardware only
Supported Output	Phone screen, sound	Phone screen, sound, external screens	Phone screen, sound	Phone screen, sound
Can application be inspected while running?	Yes	No?	No	No
Can application be modified while running?	Yes	No	No	Yes (Tuning + Alternatives)

Table 4.3: Comparison of d.tools mobile and related mobile prototyping tools.

4.1.7 LIMITATIONS & EXTENSIONS

To conclude our discussion of the d.tools system, this final section points out important limitations of the current architecture and implementation, and suggests paths for extensions.

4.1.7.1 Dynamic Graphics Require Scripting

One important limitation of the current d.tools authoring environment is that achieving dynamic graphic output, e.g., continuous animations, is only possible through the built-in scripting API; it cannot be authored visually. This is partially a side-effect of choosing states as the first-level abstraction. Consequently, information architecture can be rapidly prototyped, but more detailed work on temporal aspects of the user interface is not well supported.

Commercial tools [6,1] exist that focus on rapid creation of animated traditional, desktop-bound user interfaces. Flash Catalyst for instance also uses states as an abstraction principle, but lets designers specify explicitly how to animate transitions for individual graphical elements for each transition. Other research has looked into how to specify animations directly through stylus input [68,164]. However, it is likely insufficient to translate these techniques directly into the d.tools environment, as they do not offer support

for binding animation to the variety of possible input devices and input events in d.tools. Promising directions for this problem are to either use a visual dataflow paradigm [129] to link input events to graphical objects or to author constraints by demonstration [164].

4.1.7.2 Hierarchical Diagrams Not Supported

d.tools in its current version does not support hierarchical levels of abstraction for states. This limits the complexity of prototypes that can be built with d.tools. While we implemented parallel state machines (independent sub-graphs where one state is currently active in each sub-graph), we did not implement support for hierarchical abstraction. Abstraction has three primary benefits:

1. expressing multi-level logic, e.g., events that should apply to a set of states
2. enabling reuse of previously authored components
3. preserving screen real estate by collapsing the visual representation of clusters

Harel's original conception of statecharts [105] derives its visual economy from the notion of state clusters. Clustering also exists in dataflow languages such as Max/MSP [18]. One challenge with introducing a more powerful authoring abstraction is ensuring that this concept does not raise the expertise threshold required for novices to the tool. In informal testing, we found the notion of parallel states not well received by designers. One reason is that in parallel states, what will be shown to the user of a designed prototype is never completely visible in a single point in the diagram. Reasoning about program state now requires mentally combining the behavior of multiple active states. Similarly, reasoning about "what happens next" can be quite complex under multiple states active in parallel.

4.1.7.3 Screen Real Estate Not Used Efficiently

The current version of d.tools needlessly expends a large area of screen real estate by repeatedly displaying the device design, i.e., the set of input and output components arranged in a 2D layout, for every state in the state diagram. Having such a depiction of the device in each state enables the current authoring technique for creating transitions: clicking on an input component, then dragging out an arrow and releasing over a different state. But most of the pixels dedicated to this state display are only needed during this transition authoring. At other times, they clutter the diagram and reduce overall legibility. Hiding the device design would free up more pixels which could either be used to show more complex diagrams, or to devote more screen real estate to showing the graphical output of each state, by making the states bigger.

One possible solution achieve space savings while keeping the current authoring technique would be a dynamic visualization where the device design is hidden by default and states only show output and transitions. On mouse-rollover or another explicit invocation mechanism, the full design is temporarily shown to allow easy transition authoring. The implementation of such a technique is straightforward for the standard GUI case, where there is only a single display. It is less clear how to automatically create a suitable state abstraction when multiple output components are defined. Two possible options are to automatically rearrange the component layout; or to give the designer explicit control over how this second representation should look in the device designer.

4.1.7.4 Lack of Support for Actuation

While d.tools supports output to LEDs, DC motors, and servo motors, most of our effort has been concentrated on how to support sensor data input. We have not yet sufficiently explored the space of more complex actuation. In particular, output is controlled at the single output component level — one has to author behavior for each LED in each state individually. Such limitations are analogous to programming screen output by only writing single pixels. Many interactive projects employ arrays of displays or mechanical actuators (e.g.,Hansen and Rubin's Listening Post [104], or Rozin's Wooden Mirror [215]). Tools should therefore support output abstractions that address collections of output devices. Also, hardware and power supply design becomes a consideration when dealing with multiple outputs. The current hardware interface would need redesign to support a wider variety of actuators.
Yet a different problem arises from the tethered nature of the d.tools kit: one can't currently explore interactions that rely on precise timing and low latency feedback loops, such as for haptic interactions. Haptic motor control routines require update rates near 1kHz. In the current d.tools architecture, control loops execute at less than 100Hz, because every message has to be relayed from the hardware interface to the PC and back.

4.1.7.5 Prototypes Have to be Tethered to PC by Wire

d.tools prototypes (other than d.tools for mobile phones) are currently restricted to be used inside the design studio because because of the required tether cable linking them to a PC. The tether has two functions: it is used for data exchange, since the interaction model itself lives on the PC, and it provides power for the hardware interface sensors (actuators may need additional, separate power). There are two general strategies for cutting this tether.

First, replacing the cable with a wireless data connection and operating the hardware platform with batteries. D.tools mobile follows this approach: mobile phones send input events to the PC over a WiFi connection, and receive output events in return. The advantage of this approach is that the designer can follow in real-time on the PC what state the prototype is in, and can make on-the-fly changes. The disadvantage is that one has to be within range of the wireless signal.

A second approach is to execute interaction models directly on embedded hardware. This could be achieved by either a) running the d.tools Java state machine code on an embedded processor that can execute Java or b) by generating code for the target embedded platform separately. We have done preliminary work in the first direction by connecting components to an embedded Intel XScale platform that can execute interaction models. Stepping beyond 8-bit microcontrollers also enables on-board graphics. The advantage of this approach is that the created devices are completely standalone and do not require a PC anymore. The disadvantage is that prototype behavior can no longer be tracked and visualized on the PC.

4.2 EXEMPLAR: PROGRAMMING SENSOR-BASED INTERACTIONS BY DEMONSTRATION

d.tools and other physical computing toolkits have lowered the threshold for connecting sensors and actuators to PCs [37,43,93,127,159], and for prototyping the application logic of systems that make use of sensors and actuators. Accessing sensor data from software has come within reach of designers and end users.

However, our experience of deploying d.tools in the classroom showed that specifying the relationship between sensor input and application logic remains problematic for designers and students alike for three reasons. First, most current tools, such as Arduino [185], require using textual programming to author sensor-based behaviors. Representations are most effective when the constraints embedded in the problem are visually manifest in the representation [201]. Thus, numbers alone are a poor choice for making sense of continuous signals as the relationship between performed action and reported values is not visually apparent. Second, existing visual tools (e.g., LabView [15]) were created with the intent of helping engineers and scientists perform signal analysis; as such, they do not support straightforward authoring of interactions. This leaves users with a significant gulf of execution, the gap between their goals and the actions needed to attain those goals with the system [132]. Third, the large time and cognitive commitment implied by a lack of tools inhibits rapid iterative exploration. Creating interactive systems is not simply the activity of translating a pre-existing specification into code; there is significant value in the epistemic experience of exploring alternatives [145]. One of the contributions of direct manipulation and WYSIWYG design tools for graphical interfaces is that they enable this 'thinking through doing' — the aim of our work is to provide a similarly beneficial experience for sensor-based interactions.

This section contributes techniques for enabling a wider audience of designers and application programmers to turn raw sensor data into useful events for interaction design through programming by demonstration. It introduces a rapid prototyping tool, Exemplar (Figure 4.14), which embodies these ideas. The goal of Exemplar is to enable users to focus on design thinking (how the interaction should work) rather than algorithm tinkering (how the sensor signal processing works). Exemplar frames the design of sensor-based interactions as the activity of performing the actions that the sensor should recognize — we suggest this approach yields a considerably smaller gulf of execution than existing systems. With Exemplar, a designer first demonstrates a sensor-based interaction to the system (e.g., she

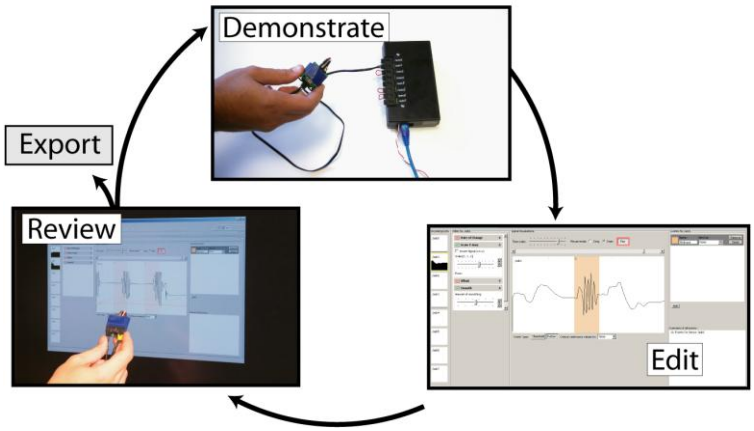

Figure 4.14: Iterative programming by demonstration for sensor-based interactions: A designer performs an action; annotates its recorded signal in Exemplar; tests the generated behavior; and exports it to d.tools.

shakes an accelerometer). The system graphically displays the resulting sensor signals. She then edits that visual representation by marking it up, and reviews the result by performing the action again. Through iteration based on real-time feedback, the designer can refine the recognized action and, when satisfied, use the sensing pattern in d.tools or other prototyping applications. The primary contributions of this work are:

1) A method of programming by demonstration for sensor-based interactions that emphasizes designer control of the generalization criteria for collected examples.
2) Integration of direct manipulation and pattern recognition through a common visual editing metaphor.
3) Support for rapid exploration of interaction techniques through the application of the design-test-analyze paradigm [109,148] on a much shorter timescale as the core operation of a design tool.

Programming by demonstration as a technique was introduced in Chapter 3.5. The rest of this section is organized as follows: we first describe relevant characteristics of sensors and sensor-based interactions to position our work. We provide an overview of the design principles embodied in Exemplar, then describe the research system, its interaction techniques and implementation. We finally report on two evaluation methods we have employed to measure Exemplar's utility and usability.

4.2.1 SENSOR-BASED INTERACTIONS

This section introduces an analysis of the space for sensor-based interactions from the designer's point of view. Prior work has successfully used design spaces as tools for thinking about task performance [57] and communicative aspects [46] of sensing systems. Here we apply this approach to describe the interaction designer's experience of working with sensors. This design space foregrounds three central concerns: the nature of the input signals, the types of transformations applied to continuous input, and techniques for specifying the correspondence between continuous signals and discrete application events.

4.2.1.1 Binary, Categorical, and Continuous Signals

As prior work points out [214], one principal distinction is whether sensing technologies report continuous or discrete data. Most technologies that directly sample physical phenomena (e.g., temperature, pressure, acceleration, magnetic field) output continuous data. For discrete sensors, because of different uses in interaction design, it is helpful to distinguish two sub-types: binary inputs such as buttons and switches are often used as general triggers; while categorical data inputs (multi-valued) such as RFID are principally used for identification. A similar division can be made for the outputs or actuators employed. Exemplar focuses on continuous input in one or more dimensions; it does not support working with categorical input data.

4.2.1.2 Working with Continuous Signals

Sensor input is nearly always transformed for use in an interactive application. Continuous transformation operations fall into three categories: signal conditioning, calibration, and mapping. Signal conditioning is about 'tuning the dials' so the signal provides a good representation of the phenomenon of interest, thus maximizing the visual signal-to-noise ratio. Common steps in conditioning are de-noising a signal and adjusting its range through scaling and offsetting. Calibration relates input units to real-world units. In scientific applications, the exact value in real-world units of a measured phenomenon is of importance. However, for the majority of sensor-based interfaces, the units of measurement are not of intrinsic value. Mapping refers to a transformation from one parameter range into another. Specifying how sensor values are mapped to application parameters is a creative process, one in which design intent is expressed. Exemplar offers support for both conditioning sensor signals and for mapping their values into binary, discrete, or continuous sets. When calibration is needed, experts can use Exemplar's extensible filter model.

4.2.1.3 Generating Discrete Events

A tool to derive discrete actions from sensor input has to choose both a detection algorithm and appropriate interaction techniques for controlling algorithm parameters. The computationally most straightforward approach is thresholding — comparing a single data point to fixed limits. However, without additional signal manipulations, e.g., smoothing and derivatives, thresholds are susceptible to noise and cannot characterize events that depend on change over time. Matching tasks such as gesture recognition require more complex pattern matching techniques. Exemplar offers both thresholding with filtering and pattern matching.

Equally important is the user interface technique employed to control how the computation happens. Threshold limits can be effectively visualized and manipulated as horizontal lines overlaid on a signal graph. The parameters of more complex algorithms are less well understood in our experience. Exemplar thus frames threshold manipulation as the principal mechanism for authoring discrete events. Exemplar contributes an interaction technique to cast parameterization of the pattern matching algorithm as a threshold operation on matching error. Through this technique, Exemplar creates a consistent user experience for authoring with both thresholding and pattern matching.

4.2.2 DESIGN PRINCIPLES

The following four design principles were derived from our analysis of sensor-based interactions.

FOCUS ON GENERATING DISCRETE EVENTS FROM CONTINUOUS SIGNALS

What kind of output should Exemplar produce? The previous section has argued that many of the most interesting potential input sources are continuous, and that discrete events are an important output category. Discrete events can be used to trigger transitions in d.tools, which provided the original motivation for this project, as well is an in other rule-based authoring systems. Signal mapping and parameter estimation (extracting not only a discrete category but also continuous parameters from sensor data) are separate problems left for future work.

LEVERAGE DEMONSTRATION TO PARTIALLY SPECIFY COMPUTATION; GIVE THE DESIGNER EXPLICIT CONTROL OF THE REMAINING STEPS

The crucial step in the success of any Programming by Demonstration system is the generalization from a small set of examples to general rules that can be applied to new input (see Section 3.5). When authoring recognizers for sensor data traces generated from human action, one has to contend with the ambiguity inherent in any recognition-based system:

there will be both misses and false positives. Giving the designer an understanding of the performance of the authored interaction and a handle on improving recognition accuracy requires showing a representation of what was learned. Exemplar uses data visualization for this task. Importantly, these visualizations are interactive — they can be manipulated to change parameters of the recognition algorithm.

PROVIDE REAL-TIME VISUAL FEEDBACK OF BOTH HARDWARE EVENTS AND APPLICATION-GENERATED EVENTS

One primary challenge for a designer of sensor-based interactions is trying to make sense of both the data streams from sensors, as well as the interaction events that are generated as a result. To aid this sensemaking task, Exemplar provides real-time visualizations of both incoming sensor data and outgoing event data in a unified graph window. Combining the two types of information in the same display helps designers reason about why a particular action did (or did not) happen.

PROVIDE ACCESS TO HISTORY OF COLLECTED SENSOR DATA

When tuning parameters of recognition algorithms, it is important to determine how those changes affect not only new performances of an action that should be recognized, but also past performances that have served as demonstrations or tests. Exemplar therefore records the entire history incoming sensor data and can visualize how any of these previous actions would have been recognized (or not recognized) given the latest recognition parameters. Reviewing this history can act as a lightweight regression test, to ensure that actions that were correctly recognized in the past are still recognized after a parameter change.

In the next section, we describe how the design principles outlined here are manifest in Exemplar's UI.

4.2.3 DESIGNING WITH EXEMPLAR

Designers begin by connecting sensors to a compatible hardware interface, which in turn is connected to a PC running Exemplar (Figure 4.15). As sensors are connected, their data streams are shown inside Exemplar. The Exemplar UI is organized according to a horizontal data-flow metaphor: hardware sensor data arrives on the left-hand side of the screen, undergoes user-specified transformations in the middle, and arrives on the right-hand side as discrete or continuous events (Figure 4.16). The success of data-flow authoring languages such as Max/MSP attests to the accessibility of this paradigm to non-programmers.

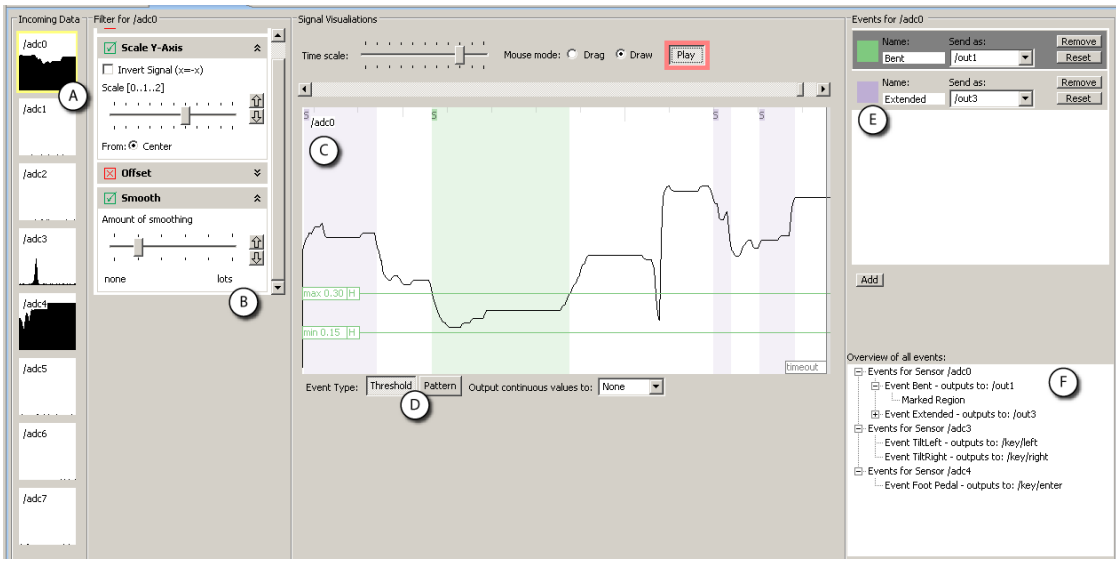

Figure 4.15: The Exemplar authoring environment offers visualization of live sensor data and direct manipulation techniques to interact with that data.

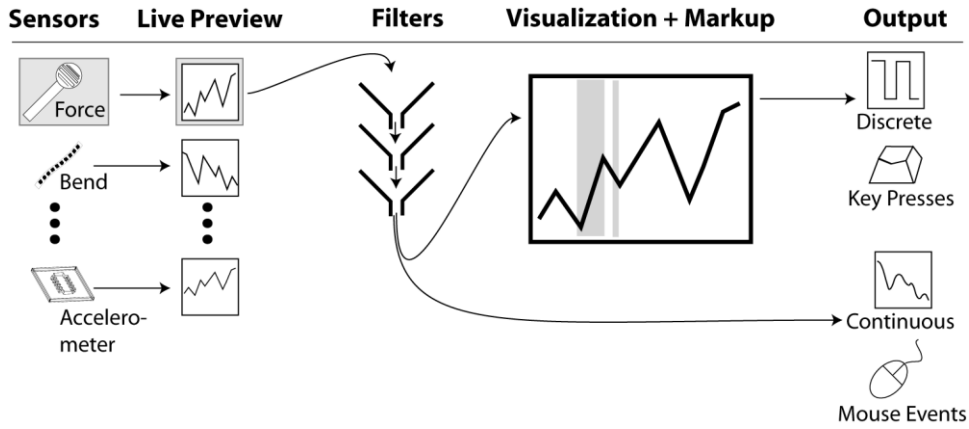

Figure 4.16: Sensor data flows from left to right in the Exemplar UI.

4.2.3.1 Peripheral Awareness

Live data from all connected sensors is shown in a small multiples configuration. Small multiples are side-by-side 'graphical depictions of variable information that share context, but not content' [245]. The small multiples configuration gives a one-glance overview of the current state of all sensors and enables visual comparison (Figure 4.15A). Whenever a signal is 'interesting,' its preview window briefly highlights in red to attract the designer's attention, then fades back to white. In the current implementation, this occurs when the derivative of a sensor signal exceeds a preset value. Together, small multiple visualization and highlighting afford peripheral awareness of sensor data and a visual means of associating sensors with their signals. This tight integration between physical state and software representation encourages discovery and narrows the gulf of evaluation, the difficulty of determining a system's state from its observable output [132]. For example, to find out which output of a multi-axis accelerometer responds to a specific tilt action, a designer can connect all axes, tilt the accelerometer in the desired plane, and look for the highlighted thumbnail to identify the correct input channel. Constant view of all signals is also helpful in identifying defective cables and connections.

4.2.3.2 Drilling Down and Filtering

Designers can bring a sensor's data into focus in the large central canvas by selecting its preview thumbnail (Figure 4.15C). The thumbnails and the central canvas form an overview + detail visualization [227]. Designers can bring multiple sensor data streams into focus at once by control-clicking on thumbnails. Between the thumbnail view and the central canvas, Exemplar interposes a filter stack (Figure 4.15B). Filters transform sensor data interactively: the visualization always reflects the current set of filters and their parameter values. Exemplar maintains an independent filter stack for each input sensor. When multiple filters are active, they are applied in sequence from top to bottom; filters can be reordered. Exemplar's filter stack library comprises four operations for conditioning and mapping:

1. Offset: adds a constant value
2. Y-axis scaling: multiplies the sensor value by a scalar, including signal inversion
3. Smoothing: convolves the signal with one-dimensional Gaussian kernel to suppress high frequency noise
4. Rate of change: takes the first derivative.

These four operations were chosen as the most important for gross signal conditioning and mapping; a later section addresses filter set extensibility.

Interaction with the filtered signal in the central canvas is analogous to a waveform editor of audio recording software. By default, the canvas shows the latest available data streaming in, with the newest value on the right side. Designers can pause this streaming visualization, scroll through the data, and change how many samples are shown per screen. When fully zoomed out, all the data collected since the beginning of the session is shown.

4.2.3.3 Demonstration and Mark-Up

To begin authoring, the designer performs the action she wants the system to recognize. As an example, to create an interface that activates a light upon firm pressure, the designer may connect a force sensitive resistor (FSR) and press on it with varying degrees of force. In Exemplar, she then marks the resulting signal curve with her mouse. The marked region is highlighted graphically and analyzed as a training example. The designer can manipulate this example region by moving it to a different location through mouse dragging, or by resizing the left and right boundaries. Multiple examples can be provided by adding more regions. Examples can be removed by right-clicking on a region.

In addition to post-demonstration markup, Exemplar also supports real-time annotation through a foot switch (chosen because it leaves the hands free for holding sensors). Using the switch, designers can mark regions at the same time they are working with sensors. Pressing the foot switch starts an example region; the region grows while the switch remains pressed, and concludes when the pedal is released. While this technique requires some amount of hand-foot coordination, it enables true real-time demonstration.

4.2.3.4 Recognition and Generalization

Recognition works as follows: interactively, as new data arrives for a given sensor, Exemplar analyzes if the data matches the set of given examples. When the system finds a match with a portion of the input signal, that portion is highlighted in the central canvas in a fainter shade of the color used to draw examples (Figure 4.15C). This region grows for the duration of the match, terminating when the signal diverges from the examples.

Exemplar provides two types of matching calculations — thresholds and patterns — selectable as modes for each event (Figure 4.15D). With thresholding, the minimum and maximum values of the example regions are calculated. The calculation is applied to filtered signals, e.g., it is possible to look for maxima in the smoothed derivative of the input.

Incoming data matches if its filtered value falls in between the extrema. Pattern matching compares incoming data against the entire example sequence and calculates a distance metric (to what extent incoming data resembles the example). Input matches when the distance metric is closer than a user-specified value.

Matching parameters can be graphically adjusted through direct manipulation. For threshold events, min and max values are shown as horizontal lines in the central canvas. These lines can be dragged with the mouse to change the threshold values (see Figure 2G). Parameters can be adjusted interactively: matched regions are automatically recalculated and repainted whenever parameters change. Thus, Exemplar always shows how the signal would have been classified. This affords rapid exploration of how changes affect the overall performance of the matching algorithm.

Sensor noise can lead to undesirable oscillation between matching and non-matching states. Exemplar provides three mechanisms for addressing this problem. First, a smoothing filter can be applied to the signal. Second, the designer can apply hysteresis, or double thresholding. In double thresholding, a boundary is represented by two values which must both be traversed for a state change. Dragging the hysteresis field of a graphical threshold manipulator (indicated by "H" in Figure 4.15G) splits a threshold into two boundary lines. The difference between boundary values is determined by the drag distance. Third, designers can drag a timeout bar from the right edge of the central canvas to indicate the minimum duration for a matching or non-matching state to be stable before an event is fired.

For pattern matching, Exemplar introduces a direct manipulation technique that offers a visual thresholding solution to the problem of parameterizing the matching algorithm (Figure

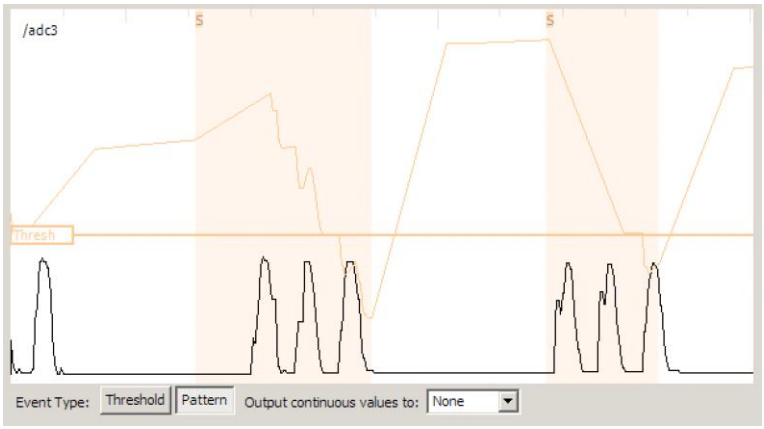

Figure 4.17: Exemplar shows output of the pattern matching algorithm on top of the sensor signal (in orange). When the graph falls below the threshold line, a match event is fired.

4.17). Exemplar overlays a graph plotting distance between the incoming data and the previously given example on the central signal canvas. The lower the distance, the better the match. Designers can then adjust a threshold line indicating the maximum distance for a positive match. When the distance graph falls below the threshold line, an event is fired. With this technique, the designer's authoring experience is consistent whether applying thresholds or pattern matching. In both cases, dragging horizontal threshold bars adjusts the specificity of the matching criteria.

4.2.3.5 Event Output

Exemplar supports the transformation from sensor-centric input into application-centric events. Exemplar generates two kinds of output events: continuous data streams that correspond to filtered input signals; and discrete events that are fired whenever a thresholding or pattern matching region is found. With these events in hand, the designer then needs to author some output, e.g., she needs to specify the application's response to the force sensor push. To integrate Exemplar with other design tools, events and data streams can be converted into operating system input events such as key clicks or mouse movements. Injecting OS events affords rapid control over third party applications (cf. [127]). However, injection is relatively brittle because it does not express association semantics (e.g., that the key 'P' pauses playback in a video application). For tighter integration with application logic, Exemplar can also be linked to d.tools. Exemplar events are then used to trigger transitions in d.tools' interaction models.

4.2.3.6 Many Sensors, Many Events

Exemplar scales to more complex applications by providing mechanisms to author multiple events for a single sensor; to run multiple independent events for different sensors simultaneously; and to author events that combine multiple sensors' data to create a single event.

To the right of the central canvas, Exemplar shows a list of event definitions for the currently active sensor(s) (Figure 4.15E). Designers can add new events and remove unwanted events in this view. Each event is given a unique color. A single event from this list is active for editing at a time, and regions drawn by the designer in the central canvas always apply to that active event.

The authored events for all sensors are always evaluated, and corresponding output is fired, regardless of which sensor is in focus in the central canvas — this allows designers to

author multiple interactions simultaneously. To keep this additional state visible, a tree widget shows authored events for all sensors along with their example regions in the lower right corner of the UI (Figure 4.15F).

Finally, Exemplar enables combining sensor data in Boolean AND fashion (e.g., 'scroll the map only if the accelerometer is tilted to the left and the center button is pressed'). When designers highlight multiple sensor thumbnails, their signals are shown stacked in the central canvas. Examples are now analyzed across all shown sensor signals and events are only generated when all involved sensors match their examples. Boolean OR between events is supported implicitly by creating multiple events. Together, AND/OR combinations enable flexibility in defining events. They reduce, but do not replace the need to author interaction logic separately.

4.2.3.7 Demonstrate-Edit-Review

The demonstrate-edit-review cycle embodied in Exemplar is an application of the design-test-think paradigm for tools introduced in prior work [109,148]. This paradigm suggests that integrating support for evaluation and analysis into a design tool enables designers to gain more insight about their project, faster. Exemplar is the first system to apply design-test-think to the domain of sensor data analysis. More importantly, Exemplar radically shortens the iteration times by an order of magnitude (from hours to minutes) by making demonstration, edit, and review actions the fundamental authoring operations in the user interface.

4.2.4 IMPLEMENTATION & ARCHITECTURE

Exemplar was written using the Java 5.0 SDK as a plug-in for the Eclipse IDE. Integration with Eclipse offers two important benefits: first, the ability to combine Exemplar with the d.tools prototyping tool to add visual authoring of interaction logic; second, extensibility for experts through an API that can be edited using Eclipse's Java tool chain. The graphical interface was implemented with the Eclipse Foundation's SWT toolkit [25].

4.2.4.1 Signal Input, Output, and Display

Consistent with the d.tools architecture, our hardware communicates with Exemplar using OpenSoundControl (OSC) [256]. This enables Exemplar to connect to any sensor hardware that supports OSC. At the present time, three hardware interfaces boards are supported: the d.tools I/O board, and the Wiring [45] and Arduino [185] boards with OSC firmware. OSC

messages are also used to send events to other applications, e.g., d.tools, Max/MSP, or Flash (with the help of a relay program). Translation of Exemplar events into system key presses and mouse movements and clicks is realized through the Java Robots package.

Exemplar visualizes up to eight inputs. This number is not an architectural limit; it was chosen based on availability of analog-to-digital ports on common hardware interfaces. Sensors are sampled at 50 Hz with 10-bit resolution and screen graphics are updated at 15-20 Hz. These sampling and display rates have been sufficient for human motion sensing and interactive operation. However, we note that other forms of input, e.g., microphones, require higher sampling rates (8-40 kHz). Support for such devices is not yet included in the current library.

4.2.4.2 Pattern Recognition

We implemented a Dynamic Time Warping (DTW) algorithm to match demonstrated complex patterns with incoming sensor data. DTW was first used as a spoken word recognition algorithm [218], and has recently been used in HCI for gesture recognition from sensor data [186]. DTW compares two time-series data sets and computes a metric of the distortion distance required to fit one to the other. It is this distance metric that we visualize and threshold against in pattern mode. DTW was chosen because, contrary to many machine learning techniques, only one training example is required. The DTW technique used in this work is sufficiently effective to enable the interaction techniques we have introduced. However, we point out that — like related work utilizing machine learning in UI tools [70,75] — we do not claim optimality of this algorithm in particular.

More broadly, this research — and that of related projects — suggests that significant user experience gains can be realized by integrating machine learning and pattern recognition with direct manipulation. From a developer's perspective, taking steps in this direction may be less daunting than it first appears. For example, Exemplar's DTW technique comprises only a small fraction of code size and development time. We have found that the primary challenge for HCI researchers is the design of appropriate interfaces for working with these techniques, so that users have sufficient control over their behavior without being overwhelmed by a large number of machine-centric 'knobs.'

4.2.4.3 Extensibility

While Exemplar's built-in filters are sufficient for a large number of applications, developers also have the option of writing their own filters, leveraging Eclipse's auto-compilation feature

for real-time integration. Developers derive from an abstract filter base class in their code and override functions for processing data. Users then specify a directory where Exemplar should search for compiled filter class files. Exemplar periodically scans that directory and adds successfully loaded extensions to the filter stack UI panel where they can be activated, deactivated and reordered like built-in filters. This architecture allows engineers on design teams to add to the filter arsenal and for users to download filters off the web. Exemplar's filter architecture was inspired by audio processing architectures such as Steinberg's VST [26], which defines a mechanism how plug-ins receive data from a host, process that stream, and return results. VST has dramatically expanded the utility of audio-editing programs by enabling third parties to extend the library of processing algorithms.

4.2.5 EVALUATION

Our evaluation followed a three-pronged approach. First, we applied the Cognitive Dimensions of Notation framework to Exemplar to evaluate the design tradeoffs of Exemplar as a visual authoring environment. Second, we conducted a first-use study in our lab to determine threshold and utility for novices, as well as to find usability problems. Third, we used Exemplar in public demonstrations and interactive installations to measure real-world performance with a larger group of participants.

4.2.5.1 Cognitive Dimensions Usability Inspection

The Cognitive Dimension of Notation (CDN) framework offers a high-level inspection method to evaluate the usability of information artifacts [89,90]. In CDN, artifacts are analyzed as a combination of a notation they offer and an environment that allows certain manipulations of the notation. CDN is particularly suitable for analysis of visual programming languages. We conducted a CDN evaluation of Exemplar following Blackwell and Green's Cognitive Dimensions Questionnaire [47] to allow the reader to revisit Exemplar according to categories independently identified as relevant, and to facilitate comparison with future research systems. This analysis begins with an estimate of how time is spent within the authoring environment, and then proceeds to evaluate the software against the framework's cognitive dimensions.

TIME SPENT

Exemplar's main notation is a visual representation of sensor data with user-generated mark-up. Lab use of Exemplar led us estimate that time is spend as follows:

30% Searching for information within the notation
(browsing the signal, visually analyzing the signal)

10% Translating amounts of information into the system (demonstration)

20% Adding bits of information to an existing description
(creating and editing mark up, filters)

10% Reorganizing and restructuring descriptions
(changing analysis types, redefining events)

30% Playing around with new ideas in notation without being sure what will result
(exploration)

This overview highlights the importance of search, and the function of Exemplar as an exploratory tool.

DIMENSIONS OF THE MAIN NOTATION

We present a discussion of the most relevant CDN dimensions here.

VISIBILITY AND JUXTAPOSABILITY (ABILITY TO VIEW COMPONENTS EASILY):

All current sensor inputs are always visible simultaneously as thumbnail views, enabling visual comparison of input data. Viewing multiple signals in close-up is also possible; however, since such a view is exclusively associated with 'AND' events combining the shown signals, it is not possible to view independent events at the same time.

VISCOSITY (EASE OR DIFFICULTY OF EDITING PREVIOUS WORK):

Event definitions and filter settings in Exemplar are straightforward to edit through direct manipulation. The hardest change to make is improving the pattern recognition if it does not work as expected. Thresholding matching error only allows users to adjust a post-match metric as the internals (the 'how' of the algorithm) are hidden.

DIFFUSENESS (SUCCINCTNESS OF LANGUAGE):

Exemplar's notation is brief, in that users only highlight relevant parts of a signal and define a small number of filter parameters through graphical interaction. The length of event descriptions is dependent on the Boolean complexity of the event expressed (how many ORs/ANDs of signal operations there are).

HARD MENTAL OPERATIONS:

Most mental effort is required to keep track of events that are defined and active, but not visible in the central canvas. To mitigate against this problem we introduced the overview list of all defined interactions (Figure 4.15F) which minimizes cost to switch between event

views. One important design goal was to make results of operations visible immediately in Exemplar.

ERROR-PRONENESS (SYNTAX PROVOKES SLIPS):

One slip occurred repeatedly in our use of Exemplar: resizing example regions by dragging their boundaries. This was problematic because no visual feedback was given on what the valid screen area was to initiate resizing. Lack of feedback resulted in duplicate regions being drawn, with an accompanying undesired recalculation of thresholds or patterns. Improved mouse manipulators on regions can alleviate this problem.

CLOSENESS OF MAPPING:

The sensor signals are the primitives users are operating on. This direct presentation of the signal facilitates consistency between the user's mental model and the system's internal representation.

ROLE-EXPRESSIVENESS (PURPOSE OF A COMPONENT IS READILY INFERRED):

Problems with role-expressiveness often arise when compatibility with legacy systems is required. Since Exemplar was designed from scratch for the express purpose of viewing, manipulating and marking up signals, this is not a problem. While the result of applying operations is always visible, the implementation "meaning" of filters and pattern recognition is hidden.

SECONDARY NOTATIONS:

Currently, Exemplar permits users to label events, but not filter settings or regions of the signal. If deemed important, this is an area for future work.

PROGRESSIVE EVALUATION:

Real-time visual feedback enables evaluation of the state of an interaction design at any point. Furthermore, Exemplar sessions can be saved and retrieved through serialization to disk.

In summary, Exemplar performs well with respect to visibility, closeness of mapping, and progressive evaluation. Many of the identified challenges stem from the difficulties of displaying multiple sensor visualizations simultaneously. These can be addressed through interface improvements — they are not inherent to the approach.

4.2.5.2 First-Use Study

We conducted a controlled study of Exemplar in our laboratory to assess the ease of use and felicity of our tool for design prototyping. The study group comprised twelve participants. Ten were graduate students or alumni of our university; two were undergraduates. While all participants had some prior HCI design experience, they came from a variety of educational backgrounds: four from Computer Science/HCI, four from other Engineering fields, two from Education, and two from the Humanities. Participants' ages ranged from 19 to 31; five were male, seven female. Two female participants served as pilot testers. Eight participants had had some prior exposure to sensor programming, but none reported to be experts (Figure 4.19).

STUDY PROTOCOL

Participants were seated at a dual-screen workstation with a d.tools hardware interface. The Exemplar software was shown on one screen, a help document on sensors was shown on the other. Participants were asked to author interactions that controlled graphics on a large projection display (Figure 4.18). We chose this large display to encourage participants to think beyond the desk(top) in their designs. We chose graphical instead of physical output since our study focused on authoring responses to sensor input only, not on actuation.

Individual study sessions lasted two hours. Sessions started with a demonstration of Exemplar. We also introduced the set of available sensors, which comprised buttons,

Figure 4.18: Exemplar study setup: participants were seated at a dual monitor workstation in front of a large wall display.

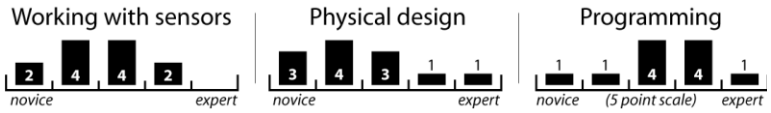

Figure 4.19: Self-reported prior experience of Exemplar study participants.

switches, capacitive touch sensors, light sensors, infrared distance rangers, resistive position sensors, force sensitive resistors (FSRs), load cells, bend sensors, 2D joysticks and 3D accelerometers. Participants were given three design tasks. For each task, we provided a mapping of triggers available in Exemplar to output behaviors in the instructions (e.g., sending an event called "hello" activated the display of the hello graphic in the first task).

The first task was a simple "Hello World" application. Subjects were asked to display a hello graphic (by issuing the "hello" event) when a FSR was pressed (through thresholding) while independently showing a world graphic when a second FSR was pressed three times in a row (through pattern recognition).

The second task required participants to augment a provided bicycle helmet with automatic blinkers such that tilting the helmet left or right causes the associated blinkers to signal. This task was inspired by Selker et al.'s Smart Helmet [225]. While blinking output was simulated on a "mirror" on the projection display, participants had to attach sensors to the real helmet.

Our last task was an open-ended design exercise to author new motion-based controls for at least one of two computer games. The first game was version of Lunar Lander in which the player has to keep a spaceship aloft, collect points and safely land using three discrete events to fire thrusters (up, left, and right). The second game was a shooting game with continuous x/y control used to aim and a discrete trigger to shoot moving targets.

STUDY RESULTS

In our post-test survey, participants ranked Exemplar highly for decreasing the time required to build prototypes compared to their prior practice (mean=4.8, median=5 on a 5-point Likert scale, σ=0.42); for facilitating rapid modification (mean=4.7, median=5, σ=0.48); for enabling them to experiment more (mean=4.7, median=5, σ=0.48); and for helping them understand user experience (mean=4.3, median=4; σ=0.48). Responses were less conclusive on how use of Exemplar would affect the number of prototypes built, and whether it helped focus or distracted from design details ($\sigma > 1.0$ in each case). Detailed results are shown in Figure 4.20.

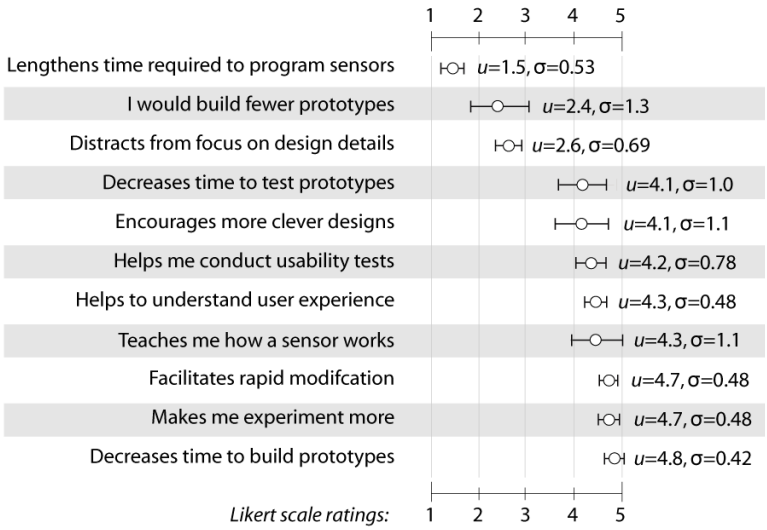

Figure 4.20: Exemplar post-experiment questionnaire results. Error bars indicate ½ standard deviation in each direction.

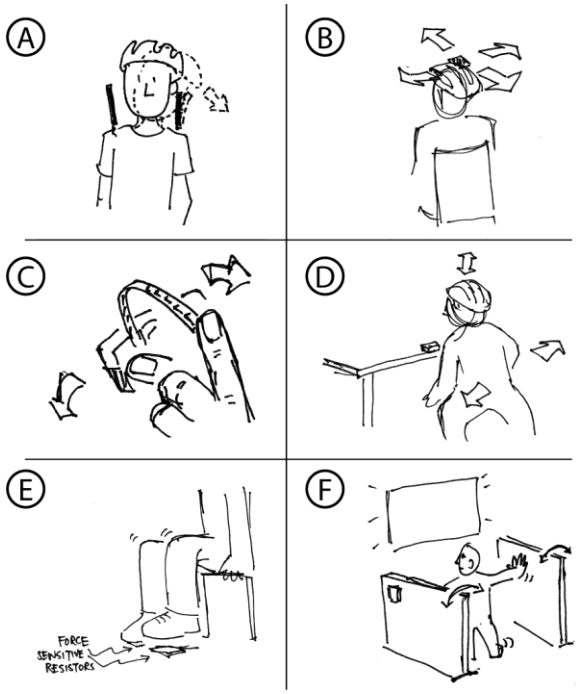

Figure 4.21: Interaction designs from the Exemplar user study. **A**: turning on blinkers by detecting head tilt with bend sensors; **B**: accelerometer used as continuous 2D head mouse; **C**: aiming and shooting with accelerometer and bend sensor; **D**: navigation through full body movement; **E**: bi-pedal navigation through force sensitive resistors; **F**: navigation by hitting the walls of a booth.

SUCCESSES

All participants successfully completed the two first two tasks and built at least one game controller. The game controller designs spanned a wide range of solutions (Figure 4.21). Once familiar with the basic authoring techniques, many participants spent the majority of their time sketching and brainstorming design solutions, and testing and refining their design. This rapid iteration cycle allowed participants to try out up to four different control schemes for a game (Figure 4.22). We see this as a success of enabling epistemic activity: participants spent their time design thinking rather than implementation tinkering.

Exemplar was used for exploration: given an unfamiliar sensor, participants were able to figure out how to employ it for their purposes. For example, real-time feedback enabled participants to find out which axes of a multi-axis accelerometer were pertinent for their design. Participants also tried multiple sensors for a given interaction idea to explore the fit between design intent and available technologies.

Interestingly, performance of beginners and experts under Exemplar was comparable in terms of task completion time and breadth of ideation. Two possible explanations for this situation are that either Exemplar was successful in lowering the threshold to entry for the types of scenarios tested; or that it encumbered experts from expressing their knowledge. The

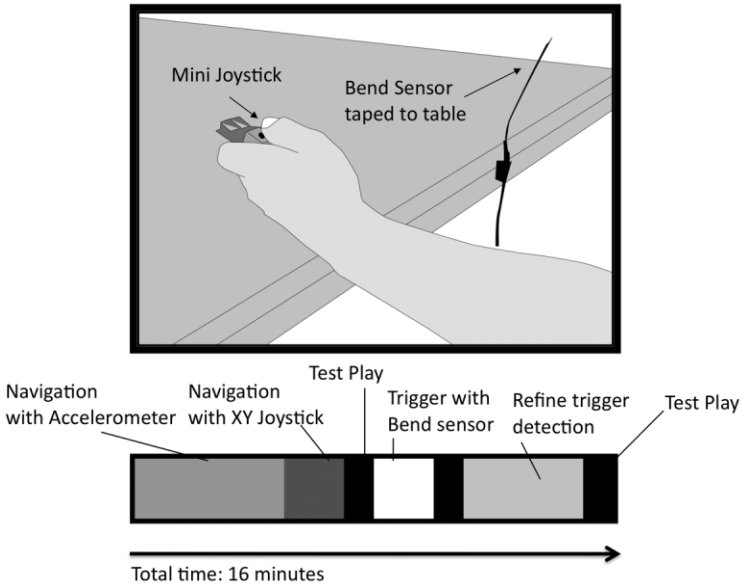

Figure 4.22: Example of one study participant's exploration: the participant created two different navigation schemes and two iterations on a trigger control; he tested his design on a target game three times within 16 minutes.

absence of complaints by experts in the post-test surveys provides some support for the first hypothesis.

SHORTCOMINGS DISCOVERED

Participants identified two key areas for improvement. One recurring theme in our feedback was the need for visualization of Exemplar's hidden state. At the time of the study, participants could only see events authored for the sensor in focus. While other events were still active, there was no comprehensive way of listing them. Also, highlighted regions corresponding to training examples were hard to retrieve after more data was collected, as the regions were pushed farther into the history of the signal. To address these difficulties, Exemplar now displays a full list of active events, along with the corresponding example regions. Selecting those regions jumps to the time of their definition in the central canvas.

Expert users expressed a need for finer control over hysteresis parameters for thresholding and a visualization of time and value units on the axes of the signal display. In response to these requests, we added direct manipulation of hysteresis and timeout parameters to threshold events.

The importance of displaying quantitative data in addition to visualization to aid the designer's mental model of events deserves further study. Participants also requested ways to provide negative examples, techniques for displaying multiple large sensor visualizations simultaneously, and finer control over the timing for pattern matching (both in terms of latency and duration).

4.2.5.3 Using Exemplar to Create Game Controllers

To gain real-world experience with a larger number of users we exhibited Exemplar at the 2007 San Mateo Maker Faire, and created a motion-controlled game for the Interactivity exhibit at the 2007 CHI conference (Figure 4.23).

The Maker Faire is a large annual gathering of amateurs interested in electronics, crafts and do-it-yourself projects. We exhibited Exemplar under the theme "Build your own game controller." We supplied the set of sensors and games used in the Exemplar lab study, as well as a collection of household items such as garden gloves, staplers, and frying pans to attach sensors to. Interested visitors were invited to come up with their own game control scheme and implement it in Exemplar with the help of one of the researchers. Several hundred visitors took part over the course of two days. Preparing for this installation sensitized us to the limitations of the Java Robot event injection technique to control closed-source 3rd party

applications: because generated keyboard and mouse events cannot be targeted to a specific application, it is easy for novices to inadvertently direct keyboard and mouse input back into Exemplar itself, which is certainly not intended. A workable but expensive solution is to use two computers: one to run Exemplar, and another to run the game. The game computer then also requires a helper application that receives socket messages from Exemplar and translates them into system keyboard and mouse events.

For the CHI conference exhibition, we used Exemplar as the back end for a wireless gaming system [261]. The game, based on Zhang's Control Freaks concept [260], featured a portable, wireless 3D accelerometer mounted to a clamp (disguised as a plush cartoon character) that could be attached to clothing or other objects to turn those objects into game controllers (Figure 4.23, right). For example, people could attach the clamp to their shoes to detect running and jumping, or to a chair to detect swiveling the chair left and right. Using Exemplar for this installation sensitized us to the limits of pattern recognition for fast-paced game play — pattern recognition incurs a compulsory latency cost a pattern can only be detected *after it has happened*. Thresholds can detect the onset of an action but may require additional application logic to suppress spurious matches beyond timeouts and hysteresis.

4.2.6 LIMITATIONS & EXTENSIONS

Exemplar currently focuses on recognizing discrete actions in low-frequency continuous sensor signals. This assumption limits the applicability of Exemplar in the following ways.

4.2.6.1 Lack of Support for Other Time Series Data

Much human motion can be adequately sampled at 50-100Hz (Winter for example reports

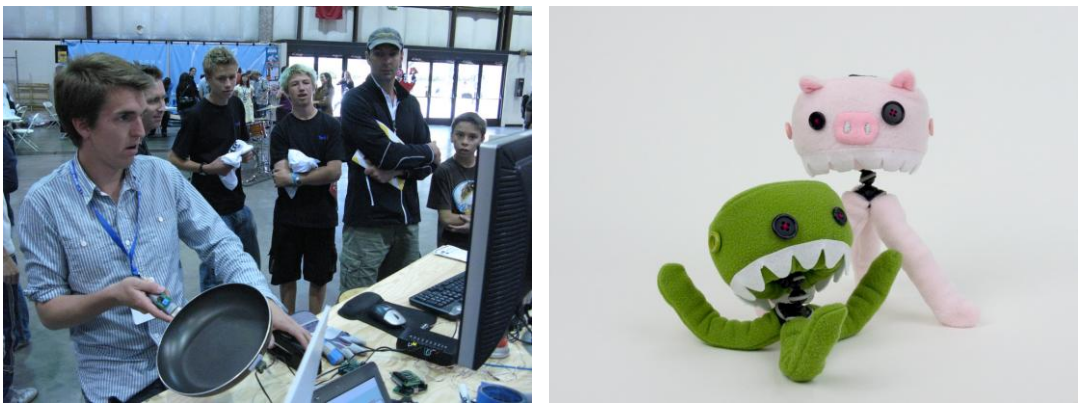

Figure 4.23: Exemplar was used for public gaming installations at the San Mateo Maker Faire and at CHI 2007. For the CHI installation, wireless accelerometers were disguised as plush characters; the characters could be attached to clothing or objects in the environment. Characters and game concept were developed by Haiyan Zhang.

that many gait analyses can be performed at 24Hz [252:Ch. 2]). However, there are applications and types of sensors for which this rate is insufficient. Audio input, for example for recognizing the scratching of fingernails on a surface [106], is commonly sampled at rates of 10-96kHz. Such higher frequency signals need different real-time visualization algorithms (which we could borrow from audio editing). We have not yet investigated to what extent dynamic time warping can be run in realtime on many parallel audio signals, or if it would offer comparative recognition performance.

4.2.6.2 Matching Performance Degrades for Multi-Dimensional Data

The employed dynamic time warping algorithm was created to compare one-dimensional time series data. The sequence alignment algorithm does not extend in a straightforward manner to matching in multiple dimensions. Exemplar uses the following generalization to make matching in multiple dimensions possible: An example for an event spanning multiple input dimensions for a given time interval is defined as individual examples $ex_1, ex_2, ..., ex_n$ in each input dimension. For new input data $in_1, in_2, ..., in_n$, a set of n DTW algorithms is then run in parallel, one for each dimension. Each outputs a binary match/no-match decision, based on individual thresholds on matching distance. Only if all dimensions independently report a match is the multi-dimensional event fired:

$$dtw\_match\big((ex_1, \dots, ex_n), (in_1, \dots, in_n)\big) = \bigcap_{k=1}^{n} dtw\_match(ex_k, in_k)$$

This approach ignores the fact that the data dimensions are interdependent and may distort different dimensions differently.

4.2.6.3 Lack of Visualization Support for Multi-Dimensional Data

Exemplar relies on the designer to make decisions about recognition algorithms and parameters based on a visualization of live sensor data. It is therefore important that the designer can interpret the visualization and make sense of it. While we found straightforward timeline visualization to be sufficient for one-dimensional sensors, this is not true for more complex sensors that return multi-dimensional data. For example, a resistive touch screen will return an *(x,y)* position; a three-dimensional accelerometer will return *(x,y,z)* acceleration data. The inherent structure of such signal spaces cannot currently be shown in Exemplar. Future work should investigate to what extent different visualizations can be used to give a

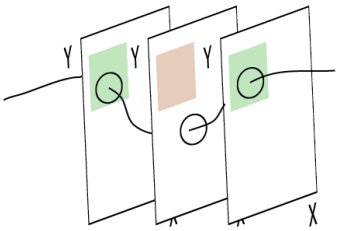

Figure 4.24: A possible visualization for 2D thresholding in Exemplar.

designer greater leverage. One challenge will be how to visualize time in higher-dimensional time-series data.

As an example, take the (x,y) position task: instead of two independent timelines, it may be advantageous to enable the designer to see a 2D space and to define thresholds as regions within that space (Figure 4.24). The 2D space could be shown as stacked, rotated slices through which the signal then describes a 3D trajectory. Events would be fired whenever the signal moves within the threshold region. A recent survey of possible visualization techniques that can inform future development can be found in [33].

4.2.6.4 Lack of Support for Parameter Estimation

Exemplar's recognizers only make binary decisions, e.g., they recognize that a tennis swing has occurred from accelerometer data. They do not yet offer parameter estimation, e.g., detecting how fast the racket was swung. A new demonstration technique would be needed that elicits examples for different parameter values from a designer. In addition, new algorithms would be needed that, given multiple examples with parameter values and new input data, can output parameter estimates. Note that it is already possible to author categorical recognizers by defining multiple events on a given signal dimension — the recognizers are then run in parallel and the single best match wins. But generalizing to the continuous case is not possible.

4.2.6.5 Difficult to Interpret Sensor Data History

Whenever the parameters of an event recognizer are changed by the designer, e.g., by moving a threshold line in the user interface, Exemplar recomputes how past data would have been classified given the new definition and updates its event visualization accordingly. However, it is hard to match the highlighted sensor signal traces back to the specific actions that produced these traces. A promising way to give the designer a better handle on understanding

how actions affect past demonstrations would be to also record live video of the demonstration and replay it inside the authoring environment as the designer reviews the history of collected data. The technique resembles interaction with the d.tools video editor (see Section 6.1), but with a much more focused role: instead of reviewing the usability of an entire prototype, the video is used to review examples used to define interaction events that are used within that prototype.