Ambient Programming

by

Nwanua Onochie Elumeze

B.S., Rochester Institute of Technology, 2000

M.S., University of Colorado, Boulder, 2002

M.S., University of Colorado, Boulder, 2007

A thesis submitted to the

Faculty of the Graduate School of the

University of Colorado in partial fulfillment

of the requirements for the degree of

Doctor of Philosophy

Department of Computer Science

2010

This thesis entitled:
Ambient Programming
written by Nwanua Onochie Elumeze
has been approved for the Department of Computer Science

_____

Michael Eisenberg

_____

Clayton Lewis

_____

Leysia Palen

Date _____

The final copy of this thesis has been examined by the signatories, and we find that both the content and the form meet acceptable presentation standards of scholarly work in the above mentioned discipline.

Elumeze, Nwanua Onochie (Ph.D., Computer Science)

Ambient Programming

Thesis directed by Professor Michael Eisenberg

Increasingly powerful and tiny electronics are finding their way into clothing, accessories, and everyday environments. It is becoming common to find computers in shoes, bracelets, hats, walls, pendants, and so on. However, the domain of computer programming has remained on the desktop: in order to customize their behavior, one must still connect these artifacts to a larger, general-purpose computer, ironically taking one out of the very physical context of use that ambient computing is intended for.

This document addresses the notion of end-user programmability of ambient artifacts. Specifically, it discusses how we may customize the behavior of a variety of embedded computational devices, some wearable and highly portable, and others part of the built environment. It builds on the rich tradition of end-user programming research, and introduces some ideas about how the nature of programming might change when users are empowered with new kinds of ambient interfaces and input sources.

Ambient programming might be seen as a natural corollary to ambient computing: the advent of a plethora of small, embedded, and mobile computational devices facilitates the creative expansion of informal techniques for communicating symbolic information to those devices. In contrast with the notions of traditional programming (e.g.) as a highly structured, abstract, and sedentary activity, ambient programming suggests a reconception of the practice of programming as (at least partially) informal, opportunistic, physically active, and playful. As the advent of embeddable computers helped change the traditional desktop-centric notions of computing, ambient programming suggests a new and potentially quite powerful means to challenge, complement, and extend the traditional desktop-centric notions of programming itself.

Dedication

This thesis is dedicated to my uncle, Emeka Elumeze.

Acknowledgements

First, I would like to thank Michael Eisenberg, my staunch advocate, for providing the environment, support and inspiration to pursue this thesis.

Second, I thank the other members of my committee: Mark Gross, Clayton Lewis, Ben Kirchner, and Leysia Palen, for their invaluable contributions and support. I extend these thanks to Ann Eisenberg, Tom Wrensch, Glenn Blauvelt, and Sue Hendrix, among others.

Third, I would like to thank my major collaborators, Ali Crockett and Brad Cooper who helped create the prototype systems; Leah Buechley for teaching me about the possible aethetics; and Yingdan Huang and Jane Meyers for their assistance on a number of projects.

I would like to thank the many hobbyists, teachers, researchers, parents, and children whose work, ideas, and critiques have left their indelible marks on this document. Of these I would like to specifically thank Osamu Iwasaki and Nilton Lessa for their continuing support.

I would also like to acknowledge my greatest supporters, my family, for starting and encouraging me up this path of discovery: my uncle, Emeka Elumeze, for sparking the flame; my mother, Ms. R.P.I Iweriebor, for encouraging the spirit; my siblings Nneka, Fanon, and Phillip, for their patience and understanding during this process; my wife, Miyuki and our children Chima and Amaka, who put my individual productivity in perspective as we journey together on this path of living; and my father, Baba, Dr. E.E.G. Iweriebor, for being an eternal source of encouragement.

This work was supported by the National Science Foundation under grants
EIA-0326054 and REC-0125363.

Contents

Chapter

1 Introduction 1

1.1 Motivating Problem 2

1.2 Research Question 9

1.3 Research Approach 9

1.4 Chapter Guide 9

2 Prototype Systems 11

2.1 Mimeolight 13

2.1.1 Record and playback light patterns 14

2.1.2 Optional settings 15

2.1.3 Self-programming 16

2.2 Learning Sensor 16

2.2.1 Parameter setting 18

2.2.2 Complexity by combination 21

2.3 Pendantif 23

2.3.1 As ambient program reader 25

2.3.2 As ambient input device 27

2.4 Schemer 28

2.4.1 Schemer ecosystem 30

2.4.2 Programming interfaces . . . 33
2.4.3 Web-based development environment . . . 33
2.4.4 Programming with barcodes . . . 37
2.4.5 Programming with color . . . 38
2.4.6 Programming with melody . . . 40
2.4.7 Programming with physical sliders and buttons . . . 41
2.5 Language facility . . . 42
2.6 Duckie . . . 45
2.7 Code Road . . . 48
2.8 Code Tracks . . . 52
2.9 Discussion of Prototypes . . . 57

3 Related Work 62
3.1 Tortis Button box - tangible programming . . . 65
3.2 Crickets - general purpose embedded computers . . . 67
3.3 roBlocks - distributed computational construction kit . . . 68
3.4 Topobo - kinetic programming by example . . . 70
3.5 Sniff - reactive toy . . . 72

4 Technical Implementation 76
4.1 Mimeolight . . . 77
4.2 Learning Sensor . . . 79
4.3 Pendantif . . . 81
4.4 Schemer . . . 83
4.4.1 Interpreter and Bytecodes . . . 86
4.4.2 Communication bus . . . 91
4.5 Current and Projected System Costs . . . 98

5 A Vision for Ambient Programming 101

5.1 Themes of Ambient Programming 101

5.1.1 Programmability, Reprogrammability, and Power 102

5.1.2 Informality and Serendipity 103

5.1.3 The Ambient Ecosystem 106

5.1.4 "Low-Tech" Ambient Code 108

5.1.5 Programming as Playful Physical Activity and Construction 111

5.2 Ambient Programming for Children 112

6 Ongoing Work & Future Directions 119

6.1 Toward Democratized Pervasive Computing 119

6.2 Ongoing Work 120

6.2.1 Collaborative programming in the large 121

6.2.2 Monitoring and tracking personal health data 123

6.2.3 Accessories, "Practical stuff", and other Ambient devices 126

6.2.4 Expanding the ecosystem 134

6.3 Future directions 136

Bibliography 140

Tables

Table

2.1 List of available melodies. 41

2.2 Matching colors to default train actions. 54

2.3 Summary of prototypes and their capabilities. 59

3.1 Summarizing additional related work. 74

4.1 Prototype systems costs per component 99

6.1 Level of involvement of the author in the different projects. 121

Figures

Figure

1.1 Code Road. 5

2.1 Mimeolight wristband. 13

2.2 Mimeolight cellphone cozy. 15

2.3 Mimeolight shirts. 16

2.4 Learning Sensors. 17

2.5 Schematic of an example learning sensor system. 19

2.6 Smart drink coaster. 20

2.7 Cascading learning sensors. 22

2.8 Additional learning sensor combinations. 22

2.9 Light- and temperature-sensitive tapestry. 23

2.10 Pendantif worn, close up. 24

2.11 Pendantif picking up colors. 25

2.12 Laying down a list of color cards. 26

2.13 Using the list card to activate a light sequence. 26

2.14 Cycling colors in a list by pressing a switch. 27

2.15 Using Pendantif as an ambient input device. 27

2.16 Schemer wristband 28

2.17 Closeup picture of Schemer 29

2.18 Schemer connected with some components 31
2.19 Closeup picture of controllers underneath Schemer, sensors, and lights 32
2.20 Closeup picture of lightboards, indicating ID dots (1-5) 32
2.21 Closeup picture of sensors, with identifying shapes 33
2.22 Schemer programming interface. 34
2.23 Beaming a program to Schemer. 35
2.24 Menu-driven interface for programming Schemer. 36
2.25 Text interface for writing Schemer programs. 37
2.26 Some Schemer instructions on barcoded cards 38
2.27 An example Schemer program on barcoded cards 38
2.28 9 colors of felt patches recognizable by Schemer 39
2.29 Two Schemer programs “written” in felt 40
2.30 Schemer programming card 42
2.31 Duckie 46
2.32 Duckie at the desktop 47
2.33 Hand coding some programs for Duckie. 48
2.34 Sending handwritten barcoded programs to Duckie. 49
2.35 Code Road. 49
2.36 Four Code Road patterns. 50
2.37 A simple Code Road program. 50
2.38 Repeating the `turn-right` instruction thrice 51
2.39 Conditionally ignoring the `turn-right` 52
2.40 An example layout of section of tracks and trains 53
2.41 A train about to read two expressions from the track. 53
2.42 Negotiating a two-loop setup with counter command. 55
2.43 A section of track showing how to define procedures 56
2.44 Redefining procedures in terms of themselves 56

2.45 Removing all bindings to user-defined procedures. 57

2.46 Dancing grasshopper. 60

3.1 Slot machine for Logo Floor Turtle 65

3.2 MIT Cricket 67

3.3 A sample roBlocks construction 68

3.4 A catalog of roBlocks modules 69

3.5 Manipulating Topobo 70

3.6 Sniff reading an RFID tag 72

3.7 Internal arrangement of Sniff electronics 73

4.1 Mimeolight Schematic. 77

4.2 Mimeolight Memory. 78

4.3 Light Emission and Detection Cycle. 79

4.4 Analog Learning Sensors Schematic. 80

4.5 Pendantif Schematic. 81

4.6 Schemer Schematic. 83

4.7 Schematic of a lightboard. 84

4.8 Schematic of a temperature sensor. 85

4.9 Turning sensor trends into pulses. 88

4.10 Picture of a bracelet with Schemer and four lights. 89

4.11 Example diagram of Schemer-bus messages. 94

4.12 Example of the low-level timing of a message signal. 97

4.13 Power and data over single line. 98

5.1 Embedding Schemer programs into pictures and paintings. 105

5.2 Embedding Schemer programs into railings. 106

6.1 Quilt of 8x8 mimeolight patches with Moleque de Idéias. 122

6.2 Motion-logging bracelet. 124
6.3 Weight-sensitive backpack 125
6.4 Mimeolight cellphone cozy. 126
6.5 Sound-sensitive bunny house 127
6.6 Model Japanese inn 128
6.7 A pair of beaded pendants 128
6.8 Closeup of beaded Pendant 129
6.9 Programmable necklace 129
6.10 Glowing knitted heron. 130
6.11 Light-activated flashing collar 130
6.12 Coffee cup holder 131
6.13 Pacman cellphone strap 132
6.14 Bristlebot-mod-mod 132
6.15 Fairy Wings 133
6.16 Sound-sensitive mask. 133
6.17 Another light-up bra 134
6.18 Repurposing a lightboard to run a motor. 136

Chapter 1

Introduction

Our everyday experience is filled with a large variety of electronics that require customization and programming but are notoriously difficult and annoying to deal with: toys, alarm clocks, DVD players, televisions, microwave ovens, laundry machines, etc. They typically present us with inscrutable one- or two-button interfaces, and more often than not, frustrate us as we try to change some aspect of their functionality. As more objects in our daily lives – even clothing - become computational, there is a need to seriously consider how to improve the interaction, how we may sensibly program all these different types of devices, before we create another generation of computational artifacts that are also difficult to manage. In other words, this is an opportunity to get it right this time, since we have gotten it so horribly wrong in the past.

Ubiquitous computing, a phrase coined by pioneer researcher Mark Weiser in his manifesto[141] (and subsequently elaborated in [142, 143], is the umbrella term used to describe a future where computational resources are made available in all parts of one's environment, (e.g.) in smart phones, clothing, running shoes, picture frames, and coffeemakers. The field, also known as "pervasive" or "ambient" computing, typically refers to the design of systems in which small embedded computers are strategically placed within physical objects and environments to make them more responsive, adaptive, personalized, or (broadly speaking) "intelligent."

True to that vision, we have witnessed a remarkable proliferation in the number and variety of small, portable computing devices. They run the gamut from "desktop furnishings" (e.g. orbs[20], desktop widgets [79], and umbrellas[21] that change appearance in response to the weather forecast

or stock activity, or a pet rabbit[139] that dances to announce the presence of an online friend), to health monitoring systems (e.g. chest straps[115] and shoes[104] that record an athlete's statistics). Some of these artifacts are "helpful appliances" (e.g. the autonomous Roomba robotic vacuum cleaner[82], or the still experimental toaster that nags its owner to eat at regular intervals[11]). Yet others form part and parcel of the living environment (e.g. a wall that identifies individuals and displays personalized information[128], or a house sound-system that routes music automatically to follow an individual).

1.1 Motivating Problem

As computation is embedded within an ever-larger array of physical artifacts, the range of scenarios for interaction likewise increases. Still, the overwhelming majority of these scenarios exhibit an intriguing "blind spot" – an area that is rarely discussed: namely, the notion of how all these devices may be (re)programmed, by the end-user. Designers envision a world in which users carry, wear, and interact with ubiquitous computers, but never actually write programs for those computers themselves. Generally, the rhetoric behind pervasive computing stresses notions such as invisibility, "transparency," ease of use, and so forth. While these terms are a bit informal, they collectively identify a set of design interests. An archetypal pervasive-computing project is one in which an object or environment is made responsive or adaptive to a user's needs, relieving the user of tedium, effort, or discomfort, and doing so without any need for active effort or participation by the user. Pervasive-computing artifacts are intended to work smoothly, autonomously, and quite possibly outside the user's conscious awareness.

We do not object to this vision in the abstract; rather we seek to complement it, to add to these systems the ability for their behavior to be customized by their users. Customization in this case is not merely changing options that have been predetermined by the original designer, but to change behavior through programming. The design philosophy behind this work is that computational artifacts should not be invisible or transparent, but should rather be part of one's conscious understanding of the environment. To quote an analogy from Michael Eisenberg, "a

clarinet, for example, is not intended to be transparent, but highly controllable; it should not relieve tedium, but should be a medium of expression; it should not work like magic, but should invite understanding and mastery; it should not save time, but should have purpose." Along these same lines then do we see pervasive artifacts as expressive media to enrich one's environments and personal spaces, rather than as technological means of relieving some putative sense of discomfort. In this sense, our vision for pervasive (or ambient, or ubiquitous) artifacts may be seen as diverging somewhat from the traditional concerns of pervasive computing research and industry. ([141, 37, 27, 88, 74, 83, 45])

One recurring criticism of ambient computing artifacts is that they are little more than passive displays for ambient data representation[2]. Some responses to these criticisms have been delightful interactive systems (tangibles) like Bishop's Marble Answering Machine[117], Ishii's Bottles[83], Pattens's Audiopad[45], and Ryokai's I/O Brush[46]. These are certainly steps in the right direction, but they do not offer the sort of end-user programming that users have come to expect of general purpose computers. After all, from the user's perspective, they are still input devices to sometimes elaborate programs written by somebody else; the end-users cannot change these systems' fundamental behaviors.

A standard retort to the desire for in-system programmability is that the embedded systems themselves are not general purpose computers; they do not have the space for keyboards and displays, nor do they have the computational resources to support editors, compilers and filesystems. The first argument is being answered by novel approaches that use (e.g.) physical blocks, barcoded cards, and radio frequency tags as input devices. The second argument (about computational resources) is beginning to ring hollow as even the most basic embedded systems being designed now contain – or can access, via the Internet – more computational power than, say, desktop computers from 10 years ago. There is certainly room to accommodate, if not a full blown development system, at least an interpreter with which an artifact's behavior can be modified.

The confluence of richly featured microcontrollers, open development platforms, and discussion forums are also empowering a growing community of builders who create and customize a new

range of computational artifacts, large and small. Indeed, the past decade has witnessed a burgeoning expansion of devices and techniques specifically created for personal (or educational) embedded computing, e.g., a flower pot that informs its owners of water levels via the twitter messaging platform[5], or a picture frame whose picture changes in response to ambient temperature[151]. An increasing number of these are also being incorporated into articles of clothing, jewelry, and other accessories which (for instance) tell the wearer which way is North[123], or a portable music player that plays different songs depending on the wearer's heartbeat[108]. They represent the first steps into what might well be a large territory of democratized design for pervasive computing, and it is important that we get it right from the start this time, by exploring how we can program and customize all these pervasive devices within the same physical context of use.

These projects (examples of pervasive, ubiquitous, and ambient computing) suggest some exciting things one should be able to do with these artifacts. One should be able to augment or replace running programs "in situ" with programs scattered throughout the environment (e.g.) represented in paintings, encoded in stockings, beamed from the desktop, and contained in other computational devices. One should also be able to opportunistically gather input for running programs: data could be the day's humidity levels, temperature readings, light sources, sequences of motion, etc. that can then be (for example) played back, augmented, or simply brought back to other, more powerful computational devices (including a desktop computer) for further processing.

The advent of tiny – and increasingly capable – embeddable computation suggests new and potentially quite powerful ways that they can be programmed, different from the notion that embedded, pervasive, and wearable computers can only be programmed at the desktop, and often not by the user/wearer. However, the current paradigm for interacting with these programmable, portable systems limits their true potential: they still need to be brought over and connected to a general purpose computer. Even when the connection is wireless (e.g. infrared or Bluetooth), the locus for this activity remains at a computer screen.

Just as small embedded computers need no longer sit on a desktop, programming – informally, "in the small" – can take place in a variety of settings and physical spaces as well. Just as

the advent of embeddable computers helped change the traditional desktop-centric notions of computing, ambient programming suggests a new and potentially quite powerful means to challenge, complement, and extend the traditional desktop-centric notions of programming itself. However, continuing along the same lines of traditional programming precludes these scenarios from ever happening.

Ambient programming is about spreading program throughout a physical setting. Traditionally, one thinks of a "program" as a collection of symbolic (generally textual) instructions stored in computer memory or printed out in paper form. Increasingly, however, it is feasible to use small mobile computational elements as "portable program readers": devices that read elements of programs that have been "written out" in various forms around one's physical environment. In effect, one can think of this idea as having program "chunks" – routines, commands, variables, and so forth – spread around one's setting in forms that are readable by portable devices.

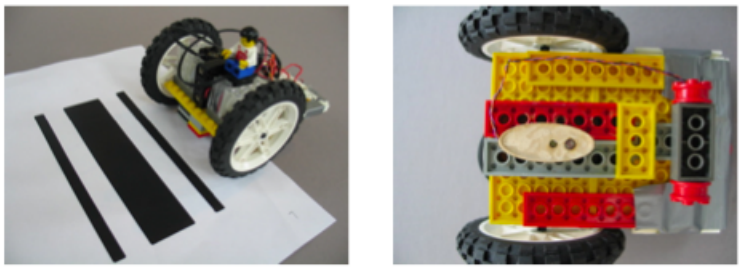

Figure 1.1: At left, a view of the ambient program-reading car about to encounter a "turn left" pattern on the ground. At right, a view of the underside of the car, showing the program reader itself.

One of the prototype examples from Chapter 2 (Code Road) might help to make these ideas more concrete. Consider, then, the small Lego car shown in Figure 1.1. The car includes a PIC microprocessor, and its undercarriage contains a "program reader" – essentially, a light source that can read the reflectance of the surface underneath the car. Figure 1.1 not only depicts the car and (at right) a view of the program reader, but it also shows something else of crucial importance in the picture at left: namely, a view of a program instruction (as it happens, a "turn to the left" instruction) written out as a bar code. Thus, if one were to start the car from the position shown

in Figure 1.1, it would roll over (and thus read) a "turn left" instruction, after which it would turn and continue on its way.[1] The car in Figure 1.1 may thus be viewed not simply as a "programmed robot", but as a device that moves about in accordance with instructions written out on the surface beneath it. By creating a small set of distinct commands that the car can read and laying them out on the floor, one can in effect "write a program" on the floor of one's room.

The purpose of this particular example is to illustrate a larger theme in ambient programming: namely, that programs can be symbolic artifacts placed in one's physical environment in all sorts of interesting ways. In our example, the program for the car's behavior is represented as cards spread about the floor; one might imagine other types of scenarios in which programs are represented via color patterns on wallpaper, or a sequence of flashing lights, or a tactile pattern of textures on a wall, to name just a few. The idea of ambient programming suggests that programs can be constructed in informal, moment-to-moment ways: one might alter the program shown in Figure 1.1 by physically messing about with the cards on the floor, changing positions and putting down new cards. Indeed, one can simply draw out (using a felt-tip marker) new patterns of the type shown in the figure; so one need not have a computer (or keyboard) to "write a program".

The key point for now is that ambient programming implies a type of programming activity that looks and feels quite different from the traditional method of desktop composition. Programs may (depending on the example) be placed around a room, drawn by hand, scrawled onto a wall, changed by whistling particular tunes, and so forth. This is clearly a very different style of programming from what is traditionally purveyed, and in our view, may be better suited for ambient, pervasive computing.

This document then, is about end-user programming. Specifically, it is about how we may customize the behavior of a variety of embedded computational devices, some wearable and highly portable, and others part of the built environment. It builds on the rich tradition of end-user programming research, and introduces some ideas about how the nature of programming might

[1] There are other instructions, further elaborated upon in Chapter 2, that (for example) cause the car to pause, repeat or ignore instructions, or to randomly take a left or right path.

change when users are empowered with new kinds of ambient interfaces and input sources. It is not necessarily about creating programming environments for novice programmers, nor is it about making programming "easier." Instead, it forms the beginning of a much wider ranging discussion about a new type of fully symbolic tangible programming – ambient programming – where the elements of instructions can be gathered from a variety or sources: encoded in everyday physical items, generated from the desktop, produced on the fly with a felt marker, embedded into a painting, or stored in other ambient computational sources for later use.

The idea of tangible programming is not new; without a doubt, there is an ample body of prior research that address "programming using physical stuff" in a variety of ways. In some systems, users generate programs by manipulating artifacts that can record and play back those motions; in others they construct programs by assembling physical blocks where each block represents a token in the language; in yet others they create programs out of the artifacts' embodied algorithmic structures.

What this document offers is an opportunity to step back from any particular implementation to explore the larger questions raised by the design of embedded systems for the practice of pervasive computing. A central theme of this work is the idea of an "ecosystem" of devices and tools for pervasive computing – a wide collection of artifacts with at least some degree of mutual compatibility that can be "mixed-and-matched" into pervasive computing projects. Increasingly, the toy-chest of the amateur or student-level pervasive computing enthusiast is taking on just this form, with a growing catalog of inexpensive sensors, actuators, controllers, and specialized materials.

At the same time, the design of the prototype systems of Chapter 2 suggests another element of this ecosystem approach: namely, creating artifacts whose design is tailored to certain genres of projects. For instance, the learning sensor[2] does not separate out a controller and sensor; nor, indeed, does it separate out the controller's hardware and software. Instead, the sensor is designed as a bundled object which combines a standard programming pattern with the hardware components

[2] Described in more detail in Chapter 2, a learning sensor is a small computational device with an integrated sensor and outputs for controlling lights and motors, among other things.

to realize that pattern. The learning sensor may thus be characterized as an intermediate device, partway between an "atomic" element (such as a temperature sensor) and a highly elaborate finished artifact (such as a commercial device like the ambient orb). The learning sensor is "molecular" in that it combines smaller components; but it can itself be incorporated into still larger projects. This approach suggests that artifacts should be designed not as stand-alone, closed systems, but rather with an eye toward their interaction with a wide range of other devices. For this reason, this work represents something of a contrast with other work in tangible programming.

An important attribute of programmable ambient devices will be their ability to get input from the environment, and to act on that input. The crucial departure from previous work is that we envision performing operations on these input sources. To explain: after (say) twisting a computational tack (Rototack[61]), or dragging a robotic wheeled dome across a surface (Curlybot[55]), or manipulating the limbs of a robotic creature (Topobo[42]), the device in question would take that example piece of data and encapsulate it such that it can be used in a textual program. The gesture could be interpreted as one in a class of motions (e.g. rotation, hinge open, push-pull, linear). Once classified, a program writer could then ask to extend the action (say, have the robotic dome complete a circle as best as it can, given a small part of it; or create a spiral by "decaying" copies of the original motion, as illustrated in [55]).

There is an analogous scenario in handwriting recognition: once a system recognizes the letter "A", it can perform a number of transformations on it: change its case or font, make it bold, and so on. Such a system could borrow a few cues from the Programming by Demonstration (PBE) community[93], [17], especially since things that have been hard to turn into generic procedures (e.g. gestures), are incidentally well suited for use as input data. This idea was explored a bit with the IO Brush[46], where a video camera equipped brush allowed for the input of complex textures for use as paint in a drawing program, though it did not operationalize them. Ambient programming addresses being able to operationalize inputs, treat them as inputs to procedures and incorporate them into data structures within the textual language.

1.2 Research Question

The research question that provides focus for this work is:

> **Can a programming platform be created through which the behavior of ambient artifacts can be modified with fully symbolic abstractions within the physical context of use?**

Traditionally, tangible programming systems have been criticized for being intuitive but too simple, limited by the range and power of expressive statements one might make. On the other hand, programming languages have been criticized for being sophisticated but (especially for beginners) hard to learn. This research question addresses the existing inability of users to customize the behavior of a large number of electronic devices; it also addresses the current state of affairs where physical computing artifacts can only be programmed from a desktop computer. In answering this question, this document makes its most significant contribution, which goes beyond prior work in this field: that is demonstrating how to profitably blend the expressive power of text with the immediacy of tangible programming.

1.3 Research Approach

To reiterate, then, programming is traditionally conceived (and taught) as a highly structured, abstract, sedentary, and time-intensive activity. By contrast, the notion of ambient programming reconceives the practice of programming as (at least partially) informal, opportunistic, physically active, and playful. Along the lines of previous work in tangible programming, ambient programming places a greater emphasis on the interweaving of abstract coding with tangible materials and construction. It advances the discussion, however, with its emphasis on **full-fledged** programming through physical means.

1.4 Chapter Guide

The next chapter (Chapter 2) presents a number of prototype systems created during this research that exemplify the goals and styles of ambient programming. It includes a description of

each system, the various ways it is programmed, and how it fits into a larger ecosystem of tiny programmable artifacts.

In order to give this work some context, Chapter 3 discusses related research and how the notion of ambient programming fits within the space already explored by existing work. In particular, it maps out the territory with specific examples and shows how the prototype systems break new ground.

Chapter 4 goes into technical detail about the prototypes' construction, providing enough information to illustrate major design decisions, as well as the capabilities and limitations of the resulting implementation.

Chapter 5 distills the different ideas presented by the different prototypes into a vision for ambient programming. In so doing, it highlights some implications for democratized pervasive computing and for children's educational activities.

Chapter 6 finishes this document with a catalog of ongoing work, done mostly by others who have incorporated these prototypes into their projects. It also discusses the various ways these systems are being extended and appropriated by designers, and outlines a variety of potential directions for future development.

Chapter 2

Prototype Systems

Traditionally, programming is conceived (and taught) as a highly structured, abstract, sedentary, and time-intensive activity. By contrast, the notion of **ambient** programming suggests a reconception of the practice of programming as (at least partially) informal, opportunistic, physically active, and playful. Ambient programming places a greater emphasis on the interweaving of abstract, symbolic coding with tangible materials and construction.

This chapter presents several prototype systems that exemplify the goals and styles of ambient programming, and provides some of the points needed to map out this space. In so doing, it demonstrates the notion of ambient programming as seen through a number of lenses and illustrates several recurring themes of ambient programming:

- Programming instructions are embedded inside or on the surfaces of day-to-day, physical objects
- Environmental cues are collected and used as bits of programs
- Informal means (such as rapid drawing by hand) of "low-tech" program construction and alteration are emphasized
- Physical activity and construction are interwoven with program composition.

There are some fascinating scenarios that are suggested by this approach to programming. One could make a truly "ubiquitous" program by writing it on a desktop computer, printing it as

barcodes or color patches on sheets of paper, and pasting the sheets on all sorts of surfaces. Casual passersby can see and read in programs thus scattered through an environment.

One could also acquire a collection of procedures for later use/modification in this manner. With crayons, fabric paint, or markers, one could even write, by hand, short programs - transcribing a sequence of bar codes on walls, clothing, bags, or a favorite book cover. The current prototype systems and underlying language allow entire procedures and lists of data to be passed between computational artifacts: for instance, a coin-sized device could send data and procedures to a computational wall, in order to (for instance) take advantage of the wall's more powerful computational resources.

The first two systems introduced in this chapter (Mimeolight and Learning Sensor), are characterized as "customizable" systems: they are able to capture and replay environmental cues, and respond to different sensor values respectively, but are not taken to be general purpose in the sense that their primary behavior can be changed. As the chapter continues, however, the systems become increasingly sophisticated, and the languages become more expressive and symbolic. The prototype systems map out different points in the space of ambient programming, and together form the ecosystem of interoperable objects, where simpler objects are used to gather input for running programs in more sophisticated ones.

Being able to write a program on the floor, whistle it to a curtain, capture bits of it for transformation in other artifacts, print it out on paper, share it with friends, print it on a bag, or physically pass it over others' computational artifacts enables us to tastefully extend the traditional landscape of programming into unexpected, informal settings.

But first, a note about Scheme, the end-user programming language used to describe the behavior of the prototype systems. Scheme [1], [28] is a small yet powerful high-level programming functional language that nicely provides constructs like tail recursion, block structure and lexical scoping. It has a very simple and consistent structure, making it relatively easy to implement on tiny embedded systems. The syntax is based on parenthesized lists in which a single prefix operator is followed by its arguments, and in which compound expressions are simply sequences of

such nested lists. Lists, the primary representation of Scheme programs, are also a primitive data structure in the language, allowing Scheme programs to easily create and evaluate pieces of Scheme expressions dynamically.

The language is also remarkable in that a variety of object types have "first-class" status, (e.g. procedures, like data, can be given names and employed freely as arguments, returned values, or as list elements). The language structure (and interpreted nature) lends itself very well to partitioning pieces of program code on various tangible surfaces. Thus, though tangible programming blocks have a simple structure, they can be combined and passed as arguments to other pieces of programs embedded in yet other tangible media.

These properties make Scheme one of very few languages particularly well suited to programming ambient devices, as will be made apparent in the following sections.

2.1 Mimeolight

Figure 2.1: At left, Mimeolight sewn onto a leather wristband; closeup at right, showing holes for connecting to power, and other devices (e.g. lights, buzzers, motors).

Mimeolight, as its name implies, is a computational object that records and replays patterns of light. The patterns can be generated either by flashing a source of light, or by creating varying levels of light and shadow (e.g. by waving one's hands in front of it). Mimeolight is a single, self-contained unit, consisting of a light-emitting diode (LED) on the top and a microcontroller

chip underneath. The LED is also used to detect changes in light levels, eliminating the need for a separate light sensor. The microchip provides the computational power to record and replay light patterns, as well as store ambient light levels. The combination yields a "smart LED" that allows its user to create custom patterns in a "language of light."

Mimeolight is generally sewn onto fabric using conductive thread, which simultaneously secures it and provides connections to a battery and other output devices; in the example above, it is sewn into a leather wristband and connected to a 3 Volt battery, also sewn into the strap.

2.1.1 Record and playback light patterns

In its normal mode, the mimeolight ceaselessly plays back a pre-recorded flashing pattern, but can also be "reprogrammed" with new patterns in a novel fashion: rather than write a sequence of patterns using text-based instructions, the wearer "shows" the mimeolight what she wants.

To reprogram it, she rapidly waves her hands at the mimeolight 3 times within a second to get its attention (Figure 2.1). It indicates its readiness to accept new patterns by blinking 4 times and then turning off. Subsequently, as she waves her hands over it, she is able to record a new sequence, up to a full minute: cast a shadow to input "light off"; allow light to input "light on". When she is done, she holds her hand steady for 3 seconds, signifying the end of the program. At that point mimeolight stops recording, and immediately starts playing back the just-recorded sequence. She can reprogram it at any time to change the pattern. Although she does not have direct access to it, the program running on mimeolight has this general form:

```
(define (playback pattern)
    (if (program-preamble-received)
        (set! pattern (record-new-pattern))
        (replay pattern))
    (playback pattern))
```

While playing back a pattern, mimeolight checks to see if a programming preamble has been received. This is true if it detects rapidly changing light levels. If this is the case, it saves the new pattern. In any case, it proceeds to play the recorded pattern back.

2.1.2 Optional settings

Mimeolight also has a hole marked *H* that is used to set some playback options. When this hole is unattached to power or ground, mimeolight will repeat its program only once until power is turned off and then back on. This feature was used for the cellphone cozy project in Figure 2.2. The cozy houses two mimeolights (hidden behind the flowers). A magnetic snap was used as a power switch: when closed it completes the circuit, enabling the mimeolights to turn on. Because the *H* hole is left untied, these mimeolights will playback the preprogrammed patterns only once. Unlike the previous program which runs continuously (by recursively calling itself), this one invokes a sleep instruction, which turns off mimeolight until the next power cycle, basically until the snaps are opened then closed again. The program is modified internally to become:

```
(define (playback-once pattern)
    (playback pattern)
    (sleep))
```

If, instead the *H* hole was connected to high, mimeolight would have endlessly repeated playback. If the *H* hole was connected to low, it would also continuously repeat playback, but would also now ignore new programming instructions.

Figure 2.2: The cellphone cozy that flashes briefly when closed. At right, the location of mimeolights, snaps, and battery.

2.1.3 Self-programming

Because mimeolights are programmed with flashing lights, they can be arranged to reprogram each other. A single mimeolight can be "meta-programmed" with a flashing pattern using traditional means (flashlight, handwaving, etc). As long as the pattern itself contains instructions to program another mimeolight, the flashing patterns from one will cause the other to start recording. To prevent runaway reprogramming, the user sets one of the mimeolights to be "read-only" by tying its H hole low. Figure 2.3 shows two t-shirts where the mimeolights on one reprogram those on the other.

Finally, mimeolight also has two extra holes (refer back to Figure 2.1) that can be connected to other output devices (e.g. lights, small motors, buzzers, and beepers). The hole marked O is high when the built-in light is lit, while the one marked $\bar{x}$ does the opposite, allowing a replay of the inverse of the programmed pattern. With these extra holes, the end-user creates on-off patterns using light, and is then able to reproduce those patterns as (for instance) motion or sound.

2.2 Learning Sensor

Learning sensors were designed to capture a large proportion of what, in our experience, beginning programmers tend to do with embedded computation: namely perform some action in

Figure 2.3: A pair of mimeolight-equipped t-shirts that can reprogram each other.

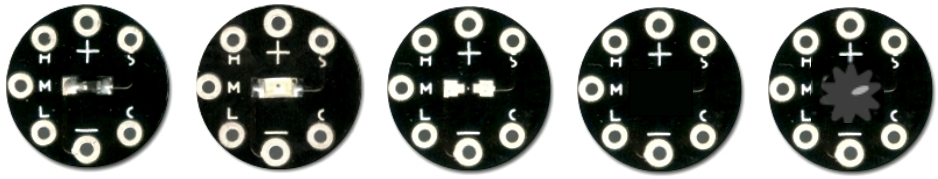

Figure 2.4: The five flavors of learning sensors from left to right: temperature, light, touch, custom, and counter. The two holes at the top and bottom (+ and −) provide power. The three holes on the left are outputs for lights, motors, buzzers, mimeolights, and other learning sensors. The "calibrate" input port is on the lower right; this is connected to "" momentarily to set the sensor's trip point. The upper hole on the right is an input port that can read values from switches or other learning sensors.

reaction to a sensor variable in the ambient environment. For instance, one fairly common program for beginners is to turn on a light when a pre-set temperature has been reached; to move a car when a light is shone on it; or to move pieces of kinetic artwork and plays sounds when the user presses a button. These projects, as skillfully crafted as they are, are effectively using the CPU as a transducer: creating sound from light, or motion from temperature.

A learning sensor, then, is a small circuit board with a microcontroller underneath and a sensor on top. It can be conditioned to learn about the ambient environment and respond when conditions change. The conditioned value is termed "the trip point", and the sensor's response depends on whether sensed ambient values are higher than, at, or lower than this trip point. It is essentially a distillation of many beginner embedded electronic projects which generally consist of processing unit, memory, sensing, and a sense-react cycle program; however, the program being run by a learning sensor cannot be changed.

Learning sensors come in five "flavors": the first four analog versions are light-sensitive, temperature-sensitive, touch-sensitive, and a custom version that is connected to many other types of sensors (e.g. pressure, acceleration, humidity, infrared). The fifth sensor incorporates a digital counter; it has an on-board dial with 10 ticks that allows the user to select 1 of 10 positions (counts). Unlike the other four, it reacts to digital pulses (on/off) rather than analog sensor values.

2.2.1 Parameter setting

Each learning sensor has three outputs that can be attached to other electronic components like lights, buzzers, motors, and other small computational artifacts. The holes are marked H, M, and L, which stand for "HIGH", "MEDIUM", and "LOW", respectively.

Only one of the outputs will be active at any given time, depending on the trip point and current ambient conditions:

- H turns on when the level rises above the trip point.
- M turns on when the level is close to the trip point.
- L turns on when the level falls below the trip point.

In the case of the digital counter, the appropriate outputs will be activated when the number of pulses it receives is greater than, equal to, or less than, the trip point value.

The user sets the trip point by briefly connecting the "calibrate" (C) port to the minus point, effectively "pressing a calibrate button". The ambient condition (light, temperature, etc.) currently being measured by the sensor is taken to be the trip point. When the calibrate button is pressed, the ambient condition (light, temperature, etc.) currently being measured by the sensor is taken to be the trip point. The "calibrate button" can in fact be implemented in a wide variety of ways, depending on the project: it could be a push switch that connects only when pushed, or a tilt switch that connects when upside down, or a light sensor that activates when sufficient light hits it, and so on. Once the calibrate button has been pushed, the learning sensor will afterward turn on the appropriate output in accordance to the rules above. Figure 2.6 show this calibration action.

The program running on the four analog sensors – not directly accessible to the user – looks like this (in Scheme syntax):

```
(if (calibrate-switch-pressed) (set! trip-point
  (read-current-sensor-value))
```

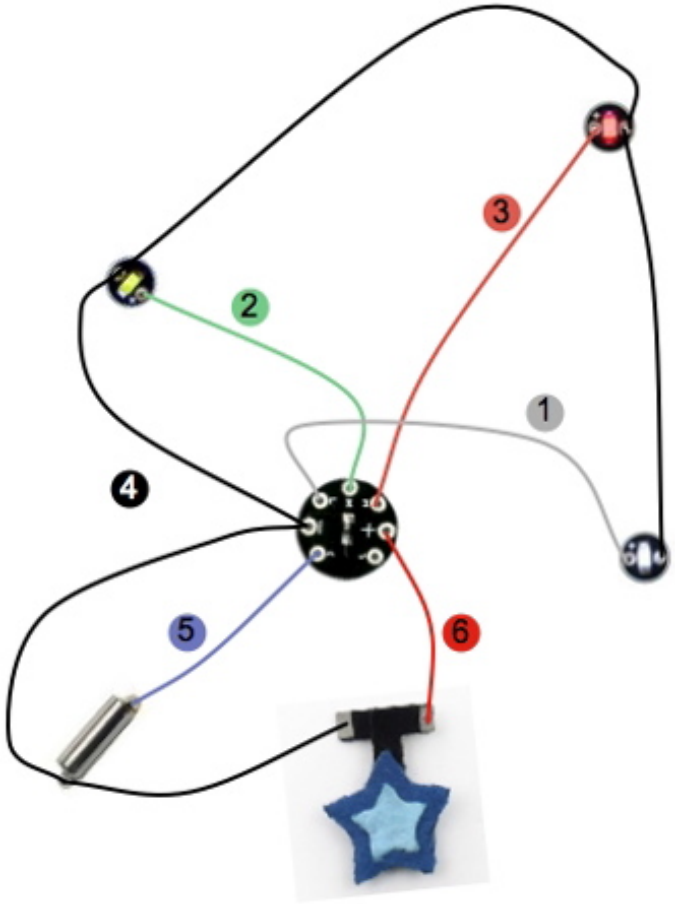

Figure 2.5: A schematic of an example learning sensor system, connected to red, green, and white lights (1, 2 and 3). The trip point is set by briefly turning the tilt switch (5) upside down. Power is supplied by a 3V battery housed in the star-shaped felt battery holder.

```
(let ((current-value (read-sensor))) (cond
  ((> current-value trip-point) (set-high-output))
  ((= current-value trip-point) (set-medium-output))
  ((< current-value trip-point) (set-low-output))
 )))
```

In other words, when the calibrate switch is pressed, the program reads in the current ambient sensor value and uses that as the basis for comparison in the next clause. When not in "calibration" mode, the program continuously monitors the sensor and turns on the appropriate outputs in response.

Likewise, the program running on digital counter sensor is similar:

```
(set! pulses-needed (read-dial-value))
(let ((current-count (read-pulses)))
  (cond
    ((> current-count pulses-needed) (set-high-output))
    ((= current-count pulses-needed) (set-medium-output))
    ((< current-count pulses-needed) (set-low-output))
  )))
```

When the calibrate switch is pressed, the learning sensor reads in the value of a small dial at the center of the device (this value will be a number from 1 to 10) and uses that as the basis for comparison in the next clause. When not in "calibration" mode, it checks for the number of pulses received and turns on the appropriate outputs in response. See Figure 2.8 for an example of how one would use the digital counter sensor.

One example application of learning sensors is demonstrated with a "smart coaster". This coaster monitors the temperature of drinks placed on it, giving a visual indication to the end-user about how hot or cold the drink is. For example, when monitoring a hot beverage (e.g. a cup of warm sake), a red light indicates that it is too hot, a green light indicates that the temperature is just right, and a white light suggests the user get a new cup.

Figure 2.6: Using a temperature learning sensor to monitor a cup of sake. Tilting the coaster activates a tilt switch which causes it to acclimatize to the cup's temperature.

It can also be calibrated to monitor cold drinks; in this case the white light could indicate "too cold", white "just right", and red "tepid". The user would set the calibration point by doing the following:

- Fill a cup with the "right-temperature" beverage of choice.
- Let it sit on the coaster for a minute.
- Remove the cup and tilt the smart coaster so all three lights start flashing - this recalibrates the sensor.
- Place the smart coaster back on the table.
- Place beverage back on coaster.
- The coaster will remember this setting for future drinks.

2.2.2 Complexity by combination

Learning sensors can also be connected to each other – in a cascade – to create more complex computational sensor scenarios. For instance, if the output of a light-sensitive learning sensor is connected to the sensor input (S) of a temperature-sensitive version (a light-temperature cascade), the end result is a logical AND of two conditions: light and temperature have to be at certain levels in order for the LED to turn on. By physically cascading learning sensors, one can assemble each sensor's small program into a larger one.

Again, the internal running program cannot be changed, but the fragment below is a textual representation of the new (combined) behavior:

```
(if (and (< light light-low-trip-point)
         (> temperature temp-high-trip-point))
    (led-on)
    (led-off)
)
```

Although a logical AND is one result of such a cascade, one can create fairly complex programs in a variety of ways: by using the different outputs (H; M; and L); by cascading different combinations of learning sensors; by changing which outputs are connected to which inputs; and by summing the outputs of several learning sensors.

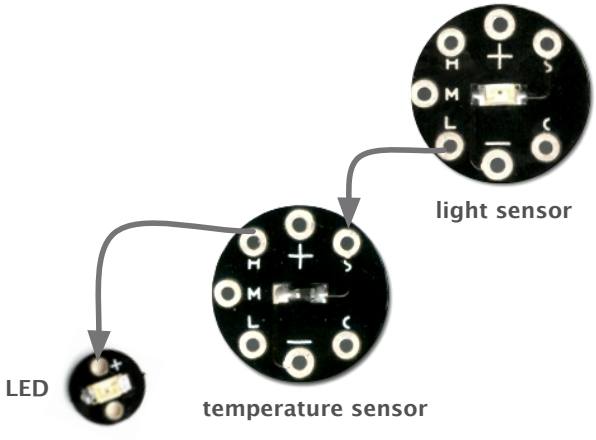

Figure 2.7: A light-temperature cascade: the "LOW" (L) output of the light sensor is connected to "SENSOR" (S) input of the temperature sensor. Results indicator LED is subsequently connected to the "HIGH" (H) output of the second sensor. As programmed, the LED will only turn on when it is dark (L from light sensor) and the temperature is above 65°F (H from temperature sensor).

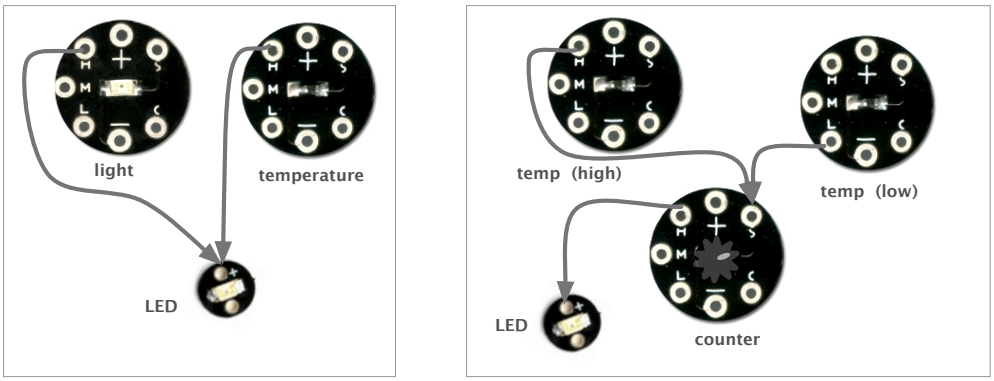

Figure 2.8: Other combinations. At left, an OR combination that lights up if either the temperature or light conditions are satisfied. At right, a cascade that turns on only if ambient temperatures have swung from one extreme (high) to the other (low) 3 times.

By combining learning sensors, one can create projects that respond under a variety of conditions. Take the temperature tapestry of Figure 2.9 for example. During the day, none of the lights on the tapestry is lit, and the tapestry is "dormant". Only at night (when it is dark) does the tapestry become "active", and the various LEDs are turned on/off to indicate ambient temperatures. In this case, the green light comes on when the surrounding temperature is around 65°F.

Figure 2.9: The cascade example used in an ambient temperature tapestry. The tapestry comes on only at night, and has different lights to indicate outside temperatures. In this example, the green lights lights are on because the conditions are "just right". To recalibrate, the user lifts the bottom-left corner of the tapestry, which engages a tilt switch. Power is supplied by the 3V battery in the butterfly-shaped holder.

This ability to cascade sensors also enables the end-user to implement some rudimentary memory into a project. The temperature-counter cascade in Figure 2.8 is one example that counts the "oscillations" between high and low temperature and lights up only if ambient temperature conditions have swung from one extreme (high) to the other (low) 3 times.

2.3 Pendantif

The Pendantif is a color-changing "gemstone" that can be incorporated into a necklace worn around the neck, or into a bracelet worn around the wrist. Although most colored gemstones exhibit colors which never change, there are a few, such as alexandrite or garnet, whose colors appear to

Figure 2.10: At left, the pendant being worn; center, a close-up of the object glowing in blue; at right, glowing in orange. As currently programmed, it takes 5 minutes for a gradual, time-varying gradation between the two colors.

change with the type of incident light. For example, one variety of alexandrite appears emerald-green in natural sunlight and raspberry-red in artificial light; some garnets appear blue-green in daylight and purple in incandescent light.

In this spirit, Pendantif is a gemstone with dynamically customizable colors. It runs a program that gradually changes between two endpoint colors, incorporating intermediate combinations of colors between the two points:

```
(if (switch-pressed?)
    (begin
        (set! endpoint-1 (button-get-color))
        (wait-until-switch-pressed-again)
        (set! endpoint-2 (button-get-color)) )
    (fade-between-two-colors endpoint-1 endpoint-2))
```

On top it features a bright 3-color LED for displaying different colors; underneath it has a tilt switch used to measure shaking, a switch for general input, and a microcontroller to provide computation. The multi-color LED is also used to detect colors, and to transmit programs to other computational devices.

In order to set endpoint colors for the program, the wearer brings her Pendantif to an everyday object and presses the switch twice. The first time she presses, it records the hue of the object as the first endpoint color. She then takes the Pendantif to another object and presses the same switch to specify the other endpoint. Figure 2.11 shows how to choose endpoint colors.

While selecting the two endpoint colors, she can (optionally) shake her pendant to alter the speed of transition between the two endpoint colors: a gentle shake means the transition should be slow, (about five minutes), and a vigorous shake means the program should run faster.

2.3.1 As ambient program reader

Pendantif's default program can be altered and augmented by swiping it over a variety of barcode-bearing surfaces. The barcodes are usually printed on cards, but we have experimented with stencil printing on walls and clothing. In this way, Pendantif functions as an "ambient program reader", able to read programs from all sorts of serendipitous sources.

The cards embody a special-purpose "iconic programming dialect" in which the textual elements are encoded underneath a pictorial representation, endowing Pendantif with the ability to read and execute simple programs. To read in an expression, the cards are aligned like in Figure2.12, and Pendantif is swept over the collection.

In this case, the programmer wishes to create a sequence of color transitions, so she first lays out the `list-a` card, followed by several color cards: (e.g. `red`, `purple`, `green`, as shown in Figure 2.12). Swiping Pendantif over these four cards produces the expression that binds the name `list-a` to the list containing these three colors:

Figure 2.11: Choosing one endpoint color for the Pendantif (worn as a bracelet). The bracelet on the left (initially yellow) is changed to blue by placing it in front of a blue object and flexing one's wrist to activate the pick-up functionality. (Flicking one's wrist closes a built-in switch.) Blue now becomes one of the endpoint colors of the pendant.

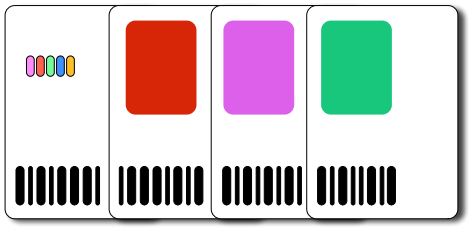

Figure 2.12: Laying down a list of color cards. The card on the left is the `list-a` card; the three other cards represent the colors to be added to the list. (There are 3 other dedicated list cards: `list-b`, `list-c`, `list-d`).

```
(define list-a (list 'red 'purple 'green))
```

Next, she lays down a `cycle-through` card, followed by the same `list-a` card in the previous step, and swipes her Pendantif again (left picture of Figure 2.13). The effect here is for Pendantif to use the named list as the argument to a built-in procedure that runs through a sequence of colors. These actions create a program to cycle through the list of colors:

```
(define (cycle-through color-list)
    (if (not (empty? color-list))
        (begin (fade-to (car color-list)
               (cycle-through (cdr color-list))))))

(cycle-through list-a)
```

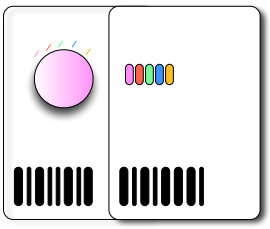

Figure 2.13: Using the list card to activate a light sequence.

If she wanted to alter the program to advance to the next color only when she presses a switch, she would insert a `(wait-until-pressed)` card between the `cycle-through` and `list-a` cards:

2.3.2 As ambient input device

Pendantif is also an "ambient input device", used to pick up environmental cues (like color) and transmit those cues to running programs in other ambient media.

Figure 2.15 shows this scenario in action: in the first photograph, the user employs the Pendantif to retrieve a selected (reddish) color from a plastic box. In the second photograph, the

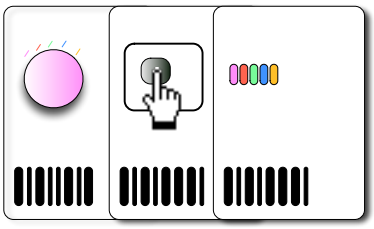

Figure 2.14: Cycling colors in a list by pressing a switch.

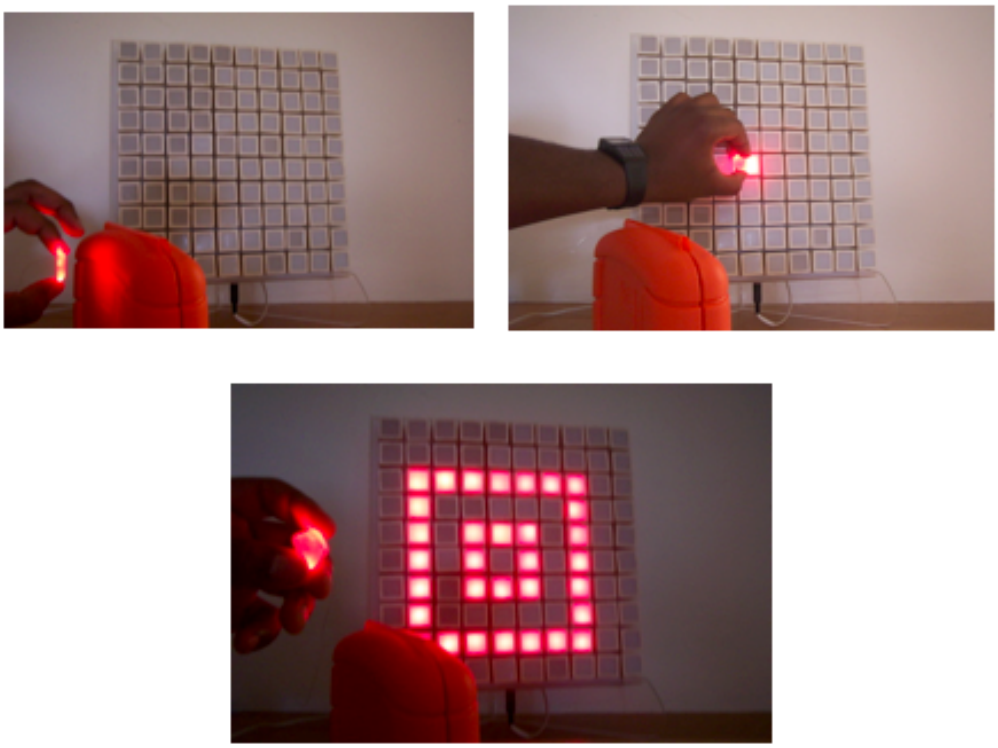

Figure 2.15: Using the Pendantif as an ambient input device. Upper left: a reddish color is retrieved from a plastic box. Upper right: that color is transmitted to a single SmartTile. Bottom: the array of tiles, acting as a cellular automaton, propagates the color over the entire surface.

Pendantif's color is transferred to a single SmartTile[1] in a large array of tiles. Once the chosen tile has changed color, the entire array of tiles acts as a cellular automaton to propagate that color in "waves" over the surface of tiles.

2.4 Schemer

Figure 2.16: Schemer, with lights sewn onto a leather wristband. Flashing patterns of lights are customized by beaming programs to it from a computer screen, swiping the bracelet over barcoded surfaces, or by whistling tunes to it.

So far we have discussed three computational artifacts that can read in data and programs from the environment in a variety of ways. Mimeolight is a program-by-example system that records and plays back patterns of light. It can also store patterns of light that are encoded as programs for other devices, though it does not have the ability to interpret those patterns. Learning sensors are customizable, combinable sensors that have the fixed behavior of setting one of three outputs high depending on how much sensor values deviate from a trip point. While the trip point is set manually by the user, the fundamental behavior cannot be changed. Learning Sensors can be cascaded and combined to produce more complex systems. Pendantif is a color-changing "gemstone" whose

[1] SmartTiles, discussed more in depth in [31], are small, touch-sensitive, independently programmable tile objects that can be combined to cover various sorts of planar surfaces. Each SmartTile contains a microcontroller that dictates the rules by which it communicates with neighboring tiles; red and green LEDs to indicate an internal state value; and a piezoelectric disk to make the tile touch-sensitive. When assembled on a grid that provides power and communication facilities, the tiles collectively enact user-customizable cellular automaton programs.

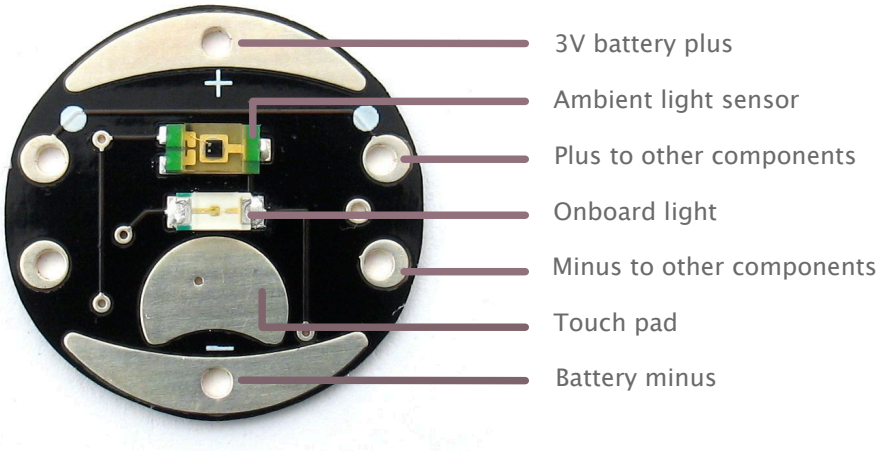

Figure 2.17: A closeup showing ambient light sensor, onboard light, and holes to connect power, inputs (switches, sensors, etc.) and outputs (lights, buzzers, etc.).

default program is to cycle between two endpoint colors. The colors are determined when the user places it in front of everyday objects and presses the "pickup this color" switch. Pendantif also reads in short programs embedded onto specially formatted barcoded cards. Although the language is symbolic, the space of available programs is small, the user cannot create new procedures or data structures, and abstraction is limited to putting colors in a list and iterating through them.

The prototype system in this section, Schemer, is the most powerful and featureful of all the systems in this dissertation. It is programmed with an expressive, fully symbolic language, and it reads in programs from a variety of sources: the computer screen, barcoded surfaces, colored pieces, melodies, and from other computational devices. Programs from these various sources can be combined such that (i.e.) higher order procedures are written in text at the desktop, and smaller snippets of code and data are expressed using tangible means. This section introduces schemer and its capabilities as a standalone system; the three succeeding sections incorporate it into several artifacts: a programmable stuffed duck, a toy car that can read and react to barcodes strewn on the floor, and a train that reads programs embedded as color on the tracks.

Schemer is a small, button-sized general-purpose computational device. An "ambient program reader", it can be programmed in a number of ways. In one instance, one can program it by placing it directly in front of a computer screen (i.e., the reader does not use a wire-based

connection). Specially timed flashes within a small area on the screen convey programming bits that are received by the onboard light sensor, similar in principle to the calendar wristwatches by Timex Corporation [18].

Schemer can also receive data and procedures from other computational devices: e.g. it can receive a procedure "beamed" from a Mimeolight or Pendantif, or sensor information from Learning Sensors. Alternatively, one can put programs into Schemer by sweeping it physically across appropriately marked materials such as paper or cloth, in the manner of a barcode reader. Finally, if equipped with an audio sensor, Schemer can read programs that have been encoded in simple melodies. Connected to the appropriate sensors, it can also react to ambient environmental values like temperature, light, sound intensity, and touch, among others.

Once it has read in a program, it can be used to activate a variety of output devices such as lights, motors, memory cards, music players, and digital picture frames; it can also communicate wirelessly with other computational devices.

One might want to think of Schemer overall, therefore, as a kind of "computational button", attached to something like a sleeve, curtain or toy car, able to read in data, procedures, or entire programs from various serendipitous sources.

The other systems presented earlier share some ideas that eventually lead up to Schemer, but Schemer is the culmination of these ideas into one project, and sits at the heart of this thesis. It is a full-fledged programmable object, and programs can be sent to it in a variety of forms. It also works with other prototype systems like the Learning sensor and Mimeolight.

2.4.1 Schemer ecosystem

In addition to Schemer, the end-user has access to a growing collection of input devices to collect data and new programs from the environment, as as well as output devices that produce motion and sound, display pictures, communicate with other devices, record sensor data, and display internal program state.

Each component of the Schemer ecosystem has a microcontroller for local computation and

for communication with other members of the system (Figure 2.19). They are all connected using the same pair of wires, using a signal and power distribution technique further discussed in Chapter 4 - Technical Implementation.

Each component has a unique identification number; for lightboards, there are five such IDs, represented by the dots at the top.

There are a few procedures that affect the state of the lights, turning them on in sequence, or manipulating them individually. Generally, their names suggest the direction of the flashing sequence and they take a single integer argument to determine the speed of the sequencing. For instance, the procedure `(right 2)` tells Schemer to turn on the left most light (ID 1) for about 1 second, turn it off, then go on to do the same with the other lights, finishing on the right (ID 5). If the expression was `(out 2)`, the pattern would start from the middle (ID 3) and spread out in both directions. The lights can also be individually controlled with expressions like `(lightboard-1 lit)` or `(lightboard-2 unlit)`. The section on Language facility goes into further details about these and other built-in procedures. There are also 5 each of other output types like motors and beepers, and they will turn on and off in sequence just like the lightboards. The growing set of output devices will be detailed in Chapter 4.

Some components like the music player and the digital picture frame do not have specific IDs. Instead, they cycle through 5 prerecorded songs or pictures that correspond to which lightboards are lit. To load these songs or images, one plugs their memory cards into a computer, and drags

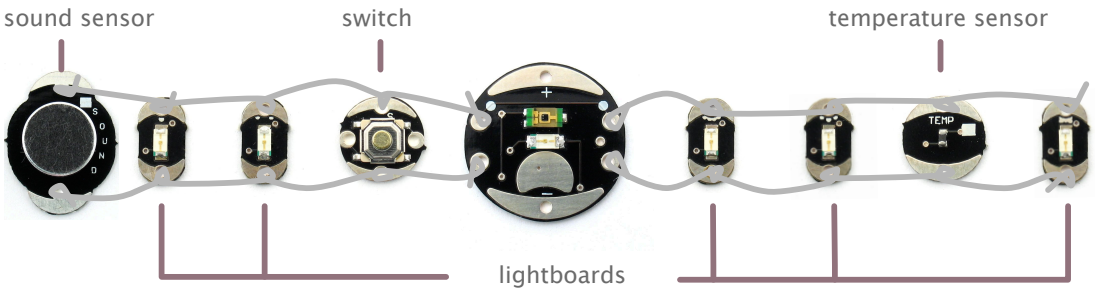

Figure 2.18: Picture of Schemer connected with some components: microphone, switch, temperature sensor and lightboards.

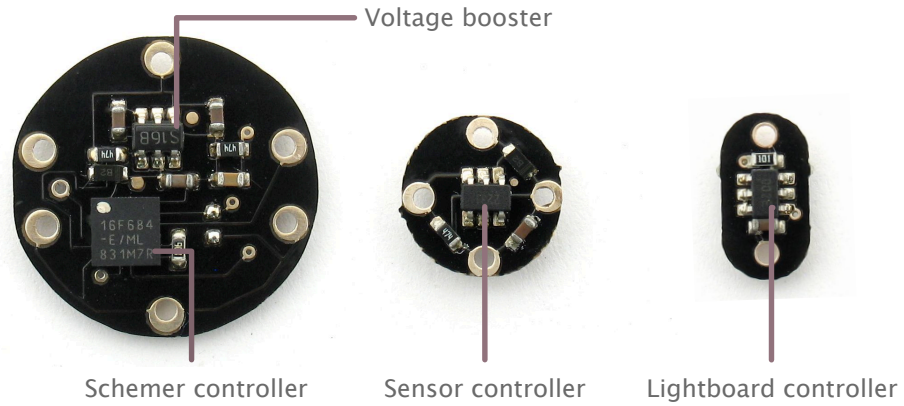

Figure 2.19: Closeup picture of the controllers underneath Schemer, sensors, and lights.

the files onto the memory card, taking care to name the files “song-1”, “song-2”, “picture-3”, etc. as desired.

Schemer uses a variety of sensors to sense the environment and to read in new programs. They are identified by the shape at the top: circle and square. Although only one of each shape can be used at any time, they could be the same type (e.g. both light sensors) or a mix (e.g. one circle temperature, and one square accelerometer).

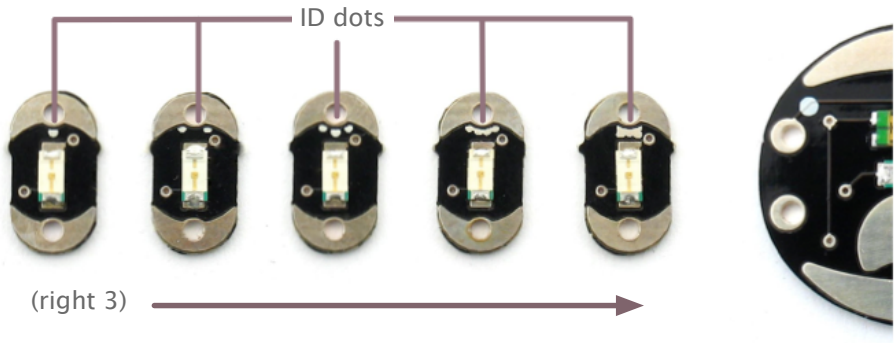

Figure 2.20: Closeup picture of lightboards, indicating ID dots (1-5). The board with ID 1 is also considered the left most light within the programming context.

2.4.2 Programming interfaces

There are five ways to program Schemer, each with varying levels of expression and complexity. The most expressive is the computer-based interface where the vast majority of Schemer programs are written. The entry methods that rely on tangible means (barcodes, color, musical notes, and the customization card), are typically used to gather short expressions and data for running programs. The methods are:

(1) web-based development environment on a computer

(2) barcodes printed on various surfaces like paper, cloth, or walls

(3) colored pieces of felt, paper, and cloth

(4) music notes and melodies

(5) physical sliders embedded into a wall or business card

2.4.3 Web-based development environment

With the desktop and mobile interfaces,, a programmer can send expressions to Schemer directly from a web browser. Specially timed flashes within a small area on the screen convey programming bits that are received by the onboard light sensor. Since the interfaces use visible light, there is no need for extra wires or special hardware.

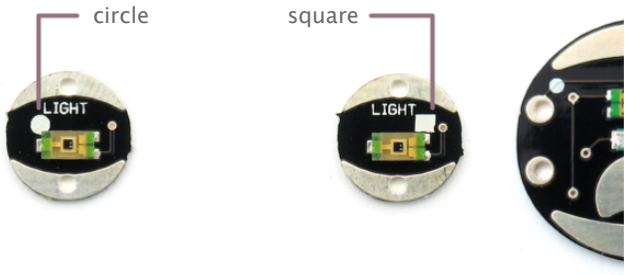

Figure 2.21: Closeup picture of sensors, with identifying shapes. They can be identified in textual programs as the `sensor-circle` and `sensor-square` variables that return the integer values (0-100) currently being measured by the sensors.

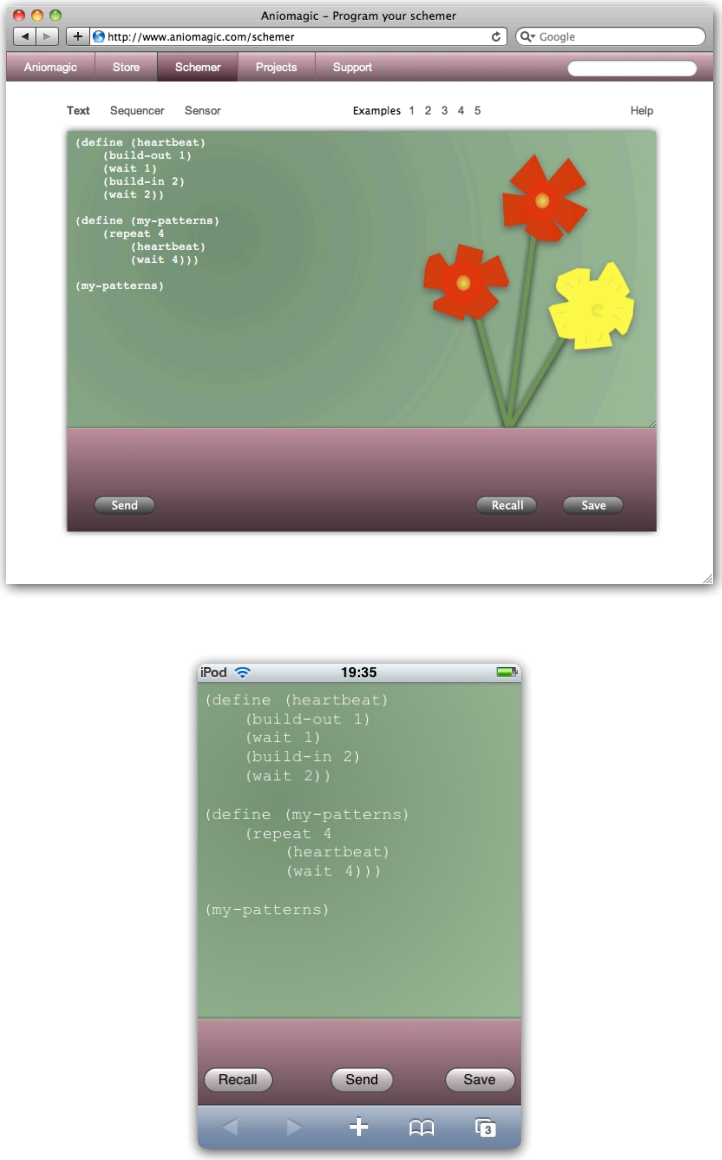

Figure 2.22: Two computer-based interfaces to programming schemer: top, on a desktop, and bottom, on a mobile device. Both versions run entirely within a web browser.

To send expressions to Schemer, the user (referring to Figure 2.23) would:

- Go to the Schemer programming web page: www.aniomagic.com/schemer
- Write a program, or use the graphical pulldown menus.
- Press Schemer's touch pad until onboard light flashes thrice.

- Put Schemer's ambient light sensor in front of the yellow flower.
- Press "Send" on the screen.
- Hold still until the web browser finishes sending the program.

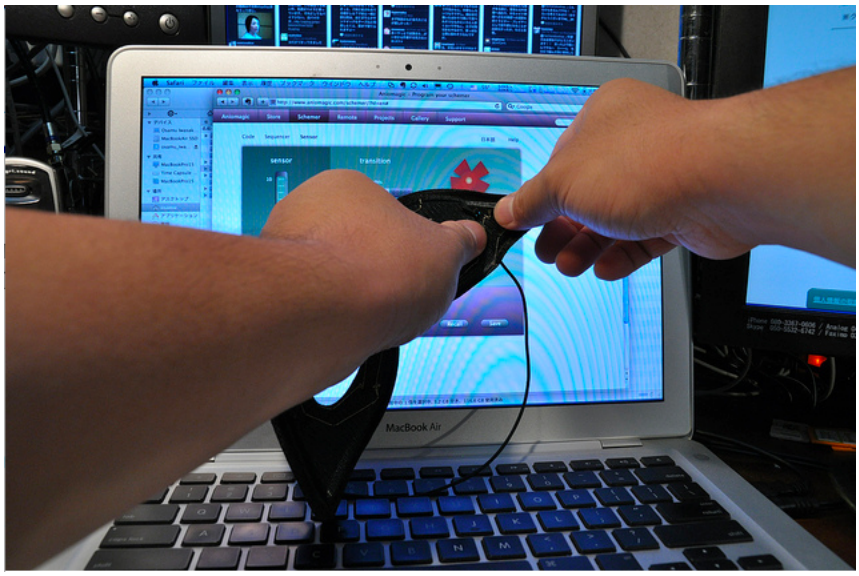

Figure 2.23: Beaming a program to Schemer from the browser-based desktop interface. Picture courtesy of MechaRoboShop.

There are two major ways of programming Schemer from the web-based environment. One method employs a menu-driven programming interface, from which a user can choose from a range of pre-programmed actions, how fast to perform them, and what to do in response to different sensor levels. This interface also allows programmers to display short strings of text which are visible only when Schemer is swung through the air. Although very simple to use – and appropriate for creating a variety of simple programs – the range of algorithms expressible through this interface is distinctly limited. A view of the menu-based interface can be seen in Figure 2.24.

This menu-driven system offers beginners a way to customize Schemer behavior without writing any text. More important – and certainly more in keeping with the notion of of **ambient** programming – is that the menus and sliders can be physically instantiated as hardware sliders and buttons embedded in (e.g.) the walls of a room. See Figure 2.30 for an illustration of the

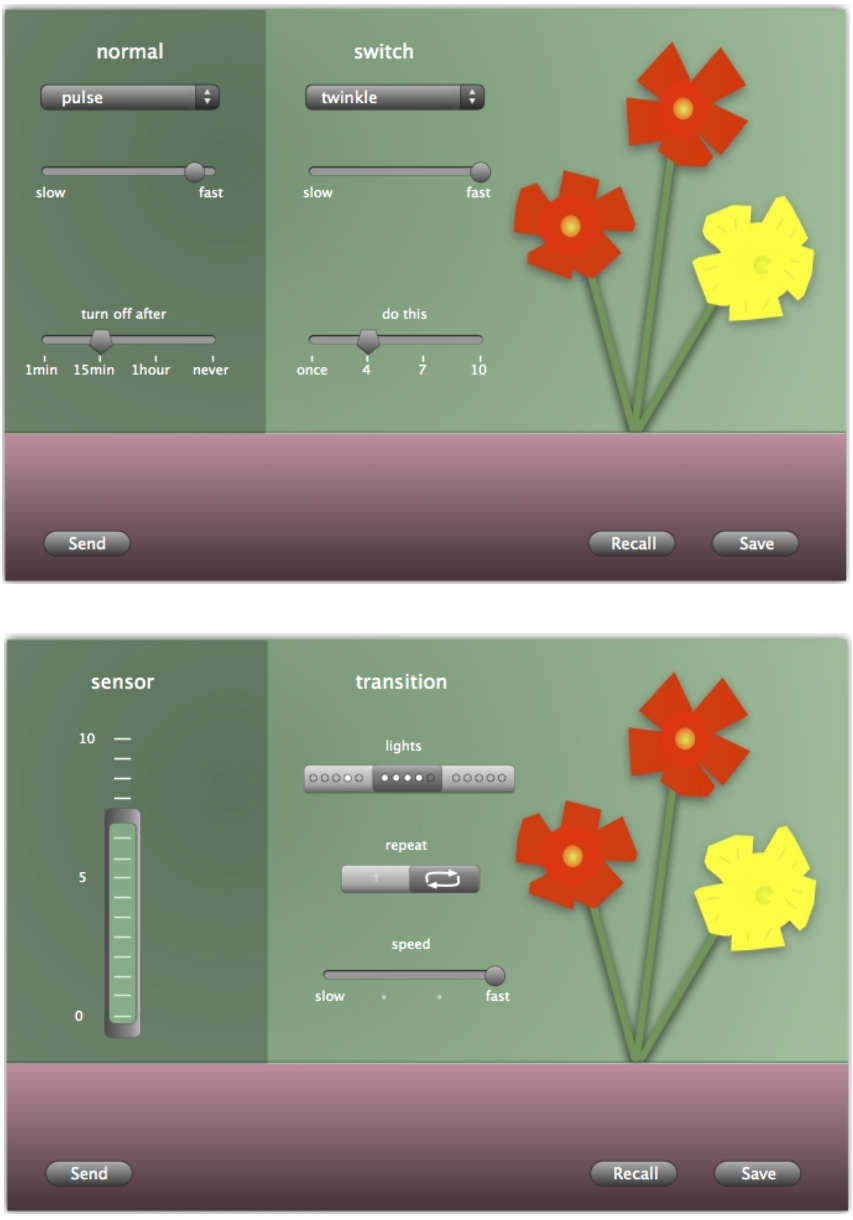

Figure 2.24: Two views of the menu-driven interface for programming Schemer. The top picture shows the drop-down menus from which built-in actions can be selected; the sliders are parameters to those actions. The bottom picture allows customization of how Schemer is to react to various sensor inputs.

"programming card".

Having presented the graphical programming environment, we now unveil the textual programming interface. Unlike most Scheme environments, where results of expressions are merely displayed on the screen, these expressions are created to be sent off to Schemer, whose built-in

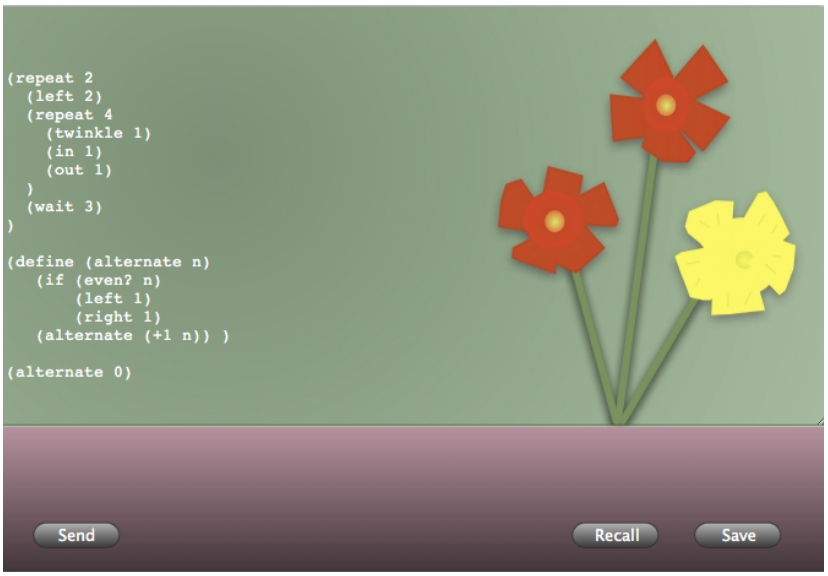

Figure 2.25: A screen view of the text interface for writing programs.

interpreter evaluates them and (optionally) performs some actions. With the textual programming interface, one can write short expressions as well as complete, self-sufficient programs and have them run on Schemer.

In a similar vein, one can write pieces of programs that cannot be sufficiently or efficiently expressed by assembling tangible objects or with barcode. After writing the program text, the flashing patterns from the screen are subsequently downloaded by a light-pattern recorder (Mimeolight). In turn, putting Mimeolights into the environment allows one to embedded "little nuggets" of complex programs in tangible media.

These nuggets can be numbers, lists, expressions, or entire programs, in fact, any Scheme object. There will be further discussion of this ability to write, store, retrieve and manipulate various type of objects, but first, on to the other means of programming.

2.4.4 Programming with barcodes

The second way to program Schemer is to lay down specially-encoded cards and swipe Schemer over them. The barcodes could also be painted on a desk, wall, or a piece of clothing. Any surface would do, as long as the barcodes look the same.

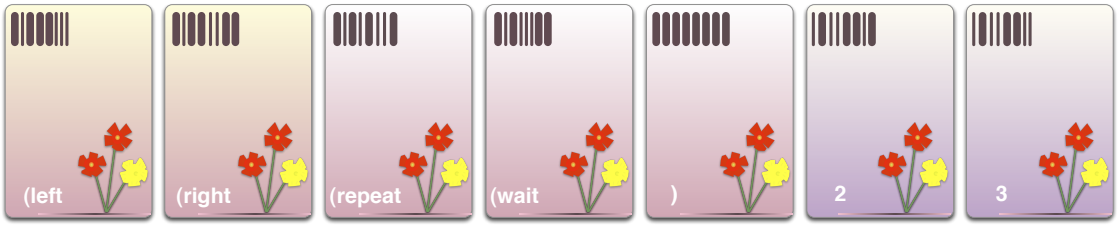

Figure 2.26: Some Schemer instructions on barcoded cards (the actual patterns read by Schemer are the bars at the top; the text are included for convenience). They are subtly colored to indicate their type: for example, the first two cards (`left` and `right`) are "light actions" and directly result in externally visible actions.

For example, to create the program shown in Figure 2.27, the appropriate cards are first arranged in a linear fashion so all the barcodes can be read in one single swipe, then Schemer is passed over them. If the program is too long to read in reliably, one would read in (say) the first 5 cards, then swipe over the last 4.

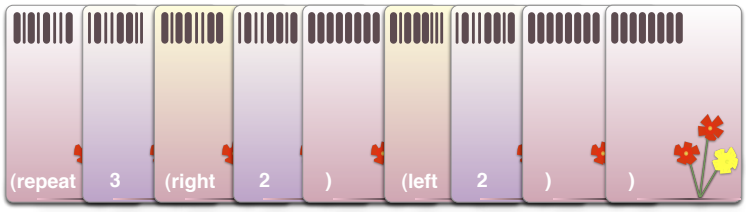

Figure 2.27: An example Schemer program on barcoded cards. After the barcodes are properly aligned, the program is read in by sweeping Schemer over them.

There are two minor issues with this method. First, the set of built-in procedures that one can call upon is significantly smaller than what is available from the computer version. In addition, there is no way to create or call user-defined procedures. Nevertheless, it does allow one to quickly create simple programs without a computer.

2.4.5 Programming with color

The third way to program Schemer is through its color-sensitive sensor. With it, Schemer can recognize the 9 distinct colors in Figure 2.28, and can perform certain actions when it encounters these colors, as in the example below:

Figure 2.28: The 9 colors of felt patches recognizable by Schemer.

```
(let ((color (button-read)))
     (cond  ((= color 'brown) ((left 2) (right 2)))
            ((= color 'blue ) (out 2))
            ((= color 'white) (twinkle 1))))
```

Here, it would fade the lights left and right when it encounters a brown object, fade them outwards for a blue one, and fade them randomly for a white one. The `(button-read)` procedure evaluates colors in the environment and returns a symbol.

One can write other types of programs using color. As with bar codes, these colors are mapped to specific built-in procedures. When Schemer is held close to an object, it interprets its color, searches for a matching Schemer expression and evaluates it. Thus one could layout a program as an array of colored pieces of felt or paper (in the same manner as the barcoded cards), and swipe Schemer over them.

Unlike barcodes however, colors can be named to other scheme entities (procedures, numbers, lists, etc.) by using the `define` special form in the web-based development environment:

```
(define (yellow)
    (if switch-pressed?
        (left 2)
        (right 2)))
```

The 9 color names (`black`, `brown`, `blue`, `purple`, `red` , `pink`, `green`, `yellow`, `white`) have special reserved meaning to Schemer and are evaluated when encountered in the everyday envi-

ronment. After reading this definition, Schemer will run the `yellow` procedure whenever it sees a yellow object.

Because the Scheme language endows a variety of object types with first-class status, colors can be employed freely as arguments, returned values, or as list elements. Again, the Language facility section has more details but a brief example suffices here:

```
(define (fade-or-twinkle speed)
    (if switch-pressed?
        (left speed)
        (right speed)))

(define blue fade-or-twinkle)
(define pink 1)
(define white 8)
```

Swiping Schemer over these patches is equivalent to writing `(fade-or-twinkle 1)` or `(fade-or-twinkle 8)` within the web-based environment on screen. As it happens, the lights on the bracelet flash very quickly when Schemer first sees a blue object followed by pink one (left of Figure 2.29); and they slowly change if it sees a blue, then white object (right of Figure 2.29). The direction depends on if a switch is pressed.

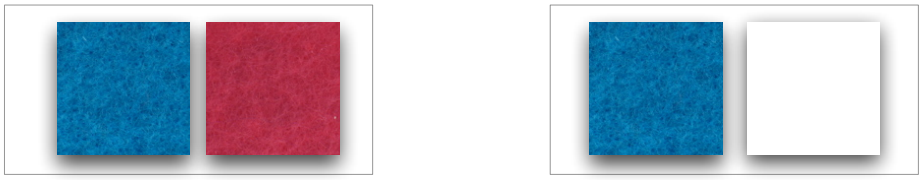

Figure 2.29: Passing arguments to the `blue` Schemer procedure; "written" in felt.

2.4.6 Programming with melody

Schemer's sound sensor endows it with the ability to "hear" programs and data that have been encoded as simple combination of notes. In its simplest form, it recognizes a 3-note combination and calls a predefined procedure when it hears that melody.

The tones are represented as two frequency ranges: a low note around 500Hz, and a high note around 1200Hz, and are represented internally as 0, and 1, respectively. The length of the notes and the pauses between them is not significant, only the frequencies are. Although the number of recognizable tones and the length of the melodies can be arbitrarily long, only 4 whistle patterns can be mapped to Schemer actions.

Returning to the bracelet as the usage scenario, one can change flashing patterns and speeds by whistling at it. Table2.1 shows the available mappings of whistles to procedures.

pattern	meaning
low-low-low	`heartbeat`
low-low-high	`fade`
low-high-high	`twinkle`
high-high-high	unmapped by default

Table 2.1: List of available melodies.

As currently implemented, whistling to Schemer merely invokes a corresponding internal entity, and does not have the symbolic power of the previous methods. However, the end-user can change the mappings from the web-browser programming environment by use of the `(melody-map)` command, as in

```
(melody-map low-low-low nod-if-prime)
```

Numbers can also be remapped so one can whistle numbers as arguments, as in:

```
(melody-map low-low-low 1)
(melody-map low-low-high 8)
```

2.4.7 Programming with physical sliders and buttons

The Schemer programming card is a physical, interactive card used to pass new arguments to running Schemer programs. It has buttons and sliders used to set parameters, and an LED that beams those settings to Schemer. To use it, one holds it in front of Schemer's ambient light sensor, and presses "Send". In fact, it is designed to be the physical instantiation of the computer-based

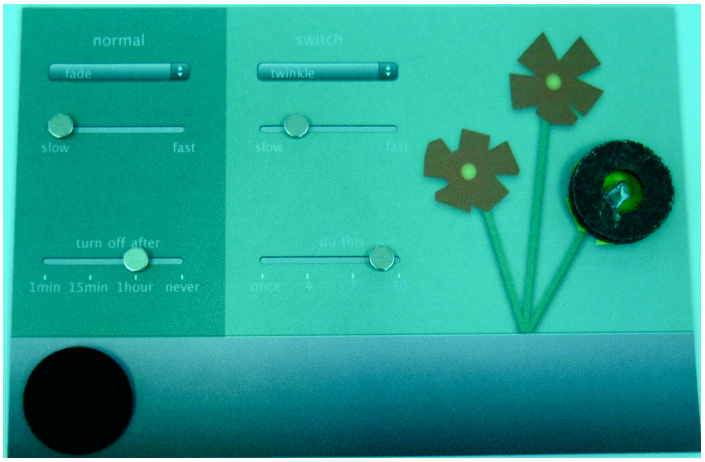

Figure 2.30: Programming card, used to change parameters to running programs. To send programs, one would move the sliders, place schemer in front of the LED towards the top right of the card, and press the round button at the bottom left.

menu-driven system, and the user interacts with it in very much the same manner. The card is meant to be carried in a pocket or embedded into a wall or desktop surface.

The idea is to write higher-order procedures in the textual language, then change parameters to those programs by moving physical sliders and pushing physical buttons. For example, if one first writes this high-order procedure that takes other procedures as arguments:

```
(define (sequencer pattern speed reps)
    (repeat reps (pattern speed)))
```

then the actual procedure being used to control lights (e.g. `left` or `twinkle`), its speed, and repetitions can be changed from the card: in this case, twinkle lights 3 times, with speed 1. This expression sent from the card: `(sequencer twinkle 3 1)` becomes `(repeat 3 (twinkle 1))` after Schemer evaluates it.

2.5 Language facility

The previous sections introduced the five different interfaces by which programs and data may be read into Schemer. Although some aspects of the language have been mentioned, this section expands on how the language enables **ambient** programming.

To run user programs, Schemer makes use of a built-in, rudimentary Scheme interpreter that executes bytecodes produced by the web browser system, represented in barcodes, colors, musical notes, or generated from the physical programming card. Like other Scheme interpreters that run on general purpose computers, this one runs a read-eval-print loop, with one difference: no text is "printed" out. Instead, to generate an action (e.g. spin a motor, turn on a light, or play a music note), one would need to explicitly call the appropriate procedure, some of which are introduced below.

There are 6 built-in procedures for making output patterns, and they have the form `(<name> <n>)`, where `<n>` is a number from 1-10. The numbers affect how long each light is lit, so with an expression like `(left 1)`, each of the five lightboards will flash in sequence within 1 second. This same commands causes appropriate actions in other output devices: the beepers would play notes in sequence, and motors would spin one after the other. Some of these procedures have already been encountered, but here is a complete list:

`left`	`right`
`in`	`out`
`center`	`twinkle`

If a programmer wants more specific control over which components are turned on or off, he can make use of expressions like: `(output-1 on)` or `(output-2 off)`; these are synonyms for the expressions `(lightboard-1 lit)` and `(lightboard-2 unlit)` previously mentioned.

Programmers make use of the `sensor` keyword, to get sensor input, and Schemer will use the following conditions to determine what value put in this variable:

- With no sensor connected, Schemer uses its ambient light sensor.
- If only one type is connected, Schemer will use it.
- If both are, Schemer will use the formula: circle - square.

A computational piece can thus behave in different ways, while the program remains the same: to make (say) a light-sensitive artwork respond to sound, simply attach a microphone; and

to make it respond to the **difference** in temperature (say by tilting in the direction of higher temperature), attach one square and one circle temperature sensor. The sensors can be attached and removed dynamically, even while programs are running.

The Schemer interpreter is designed in such a way that pieces of expressions can be gathered from different sources. For instance, take `(twinkle (+ 1 2 3))`. Some part, (say) `(twinkle (+` is written at the desktop; then the numbers (`1 2 3`) are input from barcode or bits of felt in the physical environment; and the final parenthesis is provided by whistling the "closing-parenthesis" tune. When the expression is complete, Schemer performs the action once, and goes back to waiting for more input. Schemer can also indicate errors in parsing or semantics (e.g. a number was given where a procedure or list was expected, by rapidly blinking its onboard LED until the user holds down the touchpad, causing it to "throw away" the bad input and go back to waiting for a new input.

This first example deals with reading in numbers. We start by writing two procedures: the first flashes the lights in a back and forth motion, and the second simulates a heartbeat (flash the center light, pause, then flicker lights outwards).

```
(define (back-and-forth)
    (left 1)
    (right 1))

(define (heartbeat)
    (center 1)
    (wait 1)
    (out 1))
```

Now let us use them in a new procedure that takes a number `n` so that it does a heartbeat pattern (indicating yes) if a number is prime, and the back-and-forth (indicating no) if it is not:

```
(define (nod-if-prime n)
    (if (prime? n)
        (heartbeat)
        (back-and-forth)))
```

Now if we write `(nod-if-prime 7)` and `(nod-if-prime 8)`, Schemer will respond with a

`heartbeat` (yes) and a `back-and-forth` (no) respectively. The last piece of this example is the recursive `(my-prime-indicator)` procedure, which is simply:

```
(define (my-prime-indicator)
    (nod-if-prime (button-read))
    (my-prime-indicator))
```

Where is this all going? Well the Schemer language includes a special built-in procedure named `(button-read)`, which pauses a running program and waits to read in a representation of a Schemer object (which can be a number, procedure, symbol, list, color, sound tone, or text string). By including `(button-read)`, one can now go about reading numbers in the environment [2] and have Schemer respond accordingly.

This ability to gather elements of a program from different sources – procedures, data, lists – is extremely powerful and forms the core around which the notion of **ambient** programming is built, and the next three prototypes expand on this idea. They incorporate the Schemer system into other forms: a programmable stuffed duck, a car that moves in response to barcodes it reads, and a train that performs different actions depending on what colors it sees on a track.

2.6 Duckie

Duckie is a hand-crocheted stuffed duck containing a variety of computational elements: Schemer, lightboards, a speaker, and motor actuators. Because it is based on Schemer, Duckie can read programs from a variety of sources, though we focus on its faculty for reading from a computer screen and for interpreting barcodes.

To program Duckie from the screen, the user first holds its beak closed for a few several seconds (to place the bird in "program" mode) and then holds the bird's face up to a computer screen running the "Duck programmer", a slight variation of the stock textual programming environment. The screen flashes a new program to Duckie; the duck reads the pattern directly from the computer screen, and is then ready to run the new instructions. In this manner, there is no need for special

[2] Alas, the numbers have to be encoded in a form Schemer can read: as barcodes, colors of felt, or sensor input. It would be nice if it could actually recognize numerals.

Figure 2.31: Duckie (the toy is approximately 10.5 inches in height, from toe to head). It can be programmed to flap its wings, flash the lights on its head, and sing songs.

equipment beyond the toy itself (e.g., no special connectors, infrared light sources, or RFID tags); the toy simply reads, optically, what it has to do. Duckie understands a "Duck language" that makes him flap his wings (`flap-wings 3`), flash the lights on his head (`flash-headlights 3`), or sing a song (`sing-song`).

Suppose we want to tell Duckie that we would like him to occasionally (at random) flash the lights on his head and flap his wings. Sometimes the `flash-and-flap` pattern should be done once (followed by a pause), and sometimes twice in succession (again followed by a pause). We would write:

```
(define (coin-flip)
    (flash-and-flap (+ 1 (random 2)))
    (coin-flip))
```

The effect of this procedure is to repeatedly tell Duckie to execute the `flash-and-flap` pattern either once or twice. `coin-flip` includes a final call to itself (a tail-recursive call) so that, once Duckie begins to run this procedure, it will continue to do so indefinitely until its beak is squeezed again, returning the toy to "program mode" once more.

Once the `coin-flip` program has been written, we then squeeze Duckie's beak, telling him

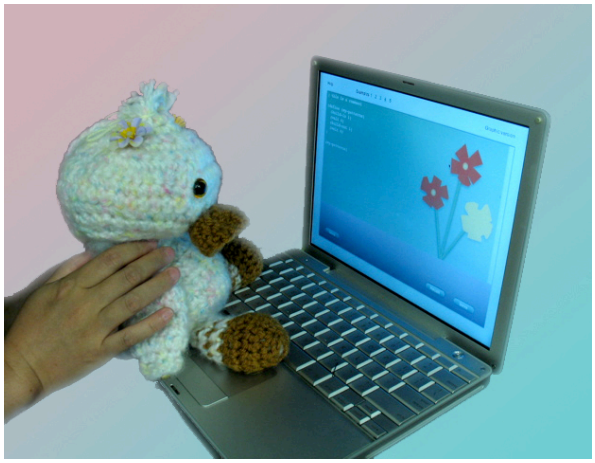

Figure 2.32: Duckie is placed in front of the computer screen, running the "Duck language"; the new program is now flashed to the toy via a pattern of lights emanating from the flower forms seen toward the right of the screen.

to (in effect) "watch the screen"; we then place Duckie up to the screen, where the new program is translated into a pattern of flashing lights. Now that the new program has been "read in" by Duckie, he will run the program, flashing and flapping, sometimes once and sometimes twice, with pauses in between. The program running on the screen may thus be thought of as a "Duck Language interpreter", translating new textual programs into flashing-light-patterns.

An alternative method of programming Duckie is the more informal, "by-hand" scenario. In this case, the user can (in effect) mimic the pattern of screen flashes in the first method by passing a series of black-and-white stripes (essentially, a hand-written barcode) in front of the duck's beak.

Figure 2.33 shows (at left) a simple handheld device, made out of wood, which permits a strip of paper to be pulled through it. (The small hole in the device is lit from behind by a bright button-activated LED.) To employ the device, the user first writes the code to be read in bar form on a paper strip; she then pulls the strip of paper through it in front of Duckie's beak while activating the LED light, as shown in Figure 2.34. The overall result is basically the same as in the "screen-programming" scenario; the toy reads the program optically much as it does when facing the computer screen.

If the program is slightly modified (e.g. `(+ 1 (random 2))` is replaced with `(button-read)`),

Figure 2.33: At left: a small wooden handheld device that can be used to program Duckie "by hand". The device has a small hole (lit by LED), and the user pulls a strip of paper with a barcode program through the device to create a "flash pattern" much like that on the screen in Figure 2.32 earlier. At right: the device placed beside Duckie to indicate comparative size.

then Duckie will respond by flapping his wings a certain number of times that matches the number he reads from the environment. One could also make him flap his wings only when he sees a prime number, using an adaptation of the `(nod-if-prime)` program seen earlier:

```
(define (respond-to-numbers)
    (if (prime? (button-read))
        (flap-wings 2))
    (respond-to-numbers))
```

2.7 Code Road

Code Road is a computational construction kit, whereby one assembles sheets of paper - each with its own instruction encoded with barcodes - for a small robotic car to read. The barcodes may be preprinted on pieces of paper, or drawn with a felt-tip marker by hand; indeed the barcodes can be made by covering parts of a floor or table with (e.g.) dark masking tape. The program-reading car carries Schemer in its undercarriage, which enables it to recognize barcodes as it whizzes along. When the car moves over a particular piece of code, it evaluates the expressions that it represents (e.g., `turn-right`, or `pause`) and performs the appropriate action. Figure 2.35 (at left) shows a photograph of the car about to read an instruction, and (at right) shows the program reader embedded in the car's undercarriage.

Figure 2.34: A new program is communicated "by hand" to Duckie, much as in Figure 2.32 earlier. Here, however, the program has simply been written on a strip of paper using a black marker pen.

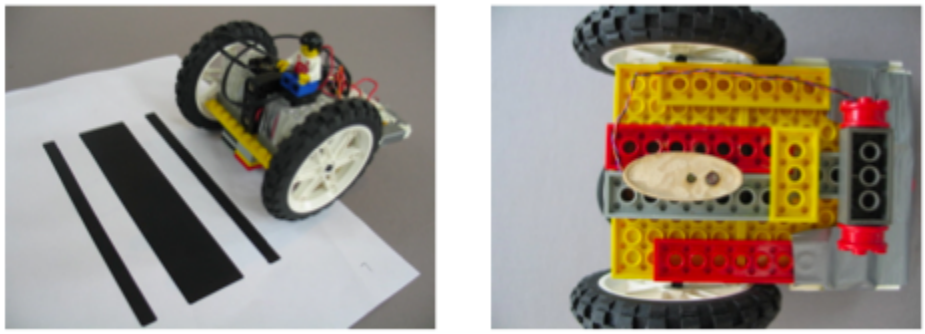

Figure 2.35: At left, the ambient program-reading car about to encounter a `turn-left` pattern. At right, a view of the underside of the car, showing the program reader.

Using this starting point, we can now create programs by composing patterns of markers on the surface over which the car will move. Specifically, certain patterns of bars have meaning for the car's behavior: one pattern (of three narrow bars) denotes "turn right immediately", while another (a narrow, wide, and narrow bar in turn) denotes "turn left immediately". (See Figure 2.36.) Thus, by placing the car in front of a piece of surface with three narrow bars, one is "programming" the car with a direction to turn right immediately upon passing over that surface. By composing sequences of patterns, one can create larger programs that indicate how the car should move over time. Figure 2.37 depicts a particular program (the same one), both in diagrammatic form and in a still from a

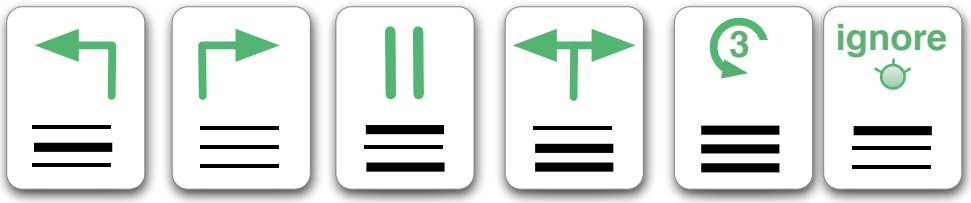

Figure 2.36: Six readable instruction patterns (the actual patterns read by the car are the bars at bottom; the symbols above are included for convenience). From left to right: `left turn`, `right turn`, `pause`, `flip-a-coin`, `repeat`, and `ignore`.

video of the working prototype. The car, once started, will first pass over the `turn-right` pattern; then two subsequent `turn-left` patterns; and finally pause briefly after passing over the farthest card toward the top.

Figure 2.37: At left, a simple Code Road program, laid out for the car to traverse: a right turn, followed by two left turns, and (finally) a pause. At right, the program "written out" on the floor in front of the actual car.

Although the available set of initial commands has been kept to a minimum for simplicity, the kit supports constructs for probability, iteration, sensor reading, and higher order procedures.

The first four cards in Figure 2.36 are standalone, single-instruction cards that cause an immediate action in the car. The `turn-left` and `turn-right` cards cause the car to make a corresponding 90° turn, and the `pause` card causes it to stop for 3 seconds before continuing its previous course. The fourth card, `flip-a-coin` tells the car to either turn left or right with a

probability of 0.5 for either choice.

In addition to these simple sequences, one can write more complex programs using the cards `repeat-three-times` and `ignore-next-when-dark`, higher-order procedures that take as argument the next instruction the car reads.

First, `repeat-three-times` tells the car to "repeat three times the move specified by the next instruction"; thus, this instruction followed by a `turn-right` card causes the car to make a three-fold right turn rather than a single right turn, as shown in Figure2.38. The procedure `(repeat-three-times)` is internally defined as:

```
(define (repeat-three-times)
    (let ((procedure-to-run (button-read)))
    (repeat 3 (procedure-to-run))))
```

Figure 2.38: A `repeat-three-times` card, followed by a `turn-right` card. If a car traveling towards the right of the page rolls over these two cards, it would make three consecutive right turns on the spot and end up travelling towards the top of the page.

Next, the `ignore-next-when-dark` card tells the car to conditionally ignore the next instruction, depending on the ambient light levels surrounding the car: if the car finds itself in the dark, it ignores the next instruction; if in a lit environment, it executes the instruction.[3] This procedure looks like:

```
(define (ignore-next-in-dark)
    if ( (< sensor-circle 60)
        (button-read)))
```

Effectively, if the level of light shining on the sensor is less than 60 (an empirically derived

[3] This procedure does not actually call the next instruction, it just immediately returns without "eating up" the instruction.

number), the procedure still reads in the next instruction, but does not execute or assign it, effectively "throwing it away."

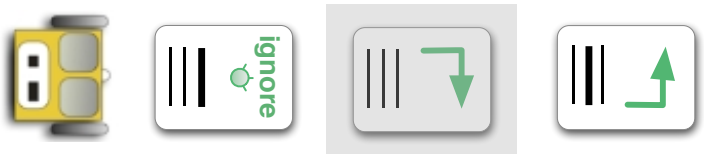

Figure 2.39: An `ignore-next-when-dark` card, followed by a `turn-right` card, and a `turn-left` card. In the dark, the car will ignore the adjacent card (`turn-right`, grayed out), and upon encountering the next card (`turn-left`), turn left towards the top of the page. If the area around the car is sufficiently lit, it would have taken the right turn towards the bottom of the page.

Code Road illustrates a recurring theme in this style of programming: namely, that programs can be symbolic artifacts placed in one's physical environment in all sorts of interesting ways. In this case, the "program" for the car's behavior is represented as cards spread about the floor, constructed in informal, moment-to-moment ways. One might alter the programs by physically messing about with cards upon the floor, changing positions and putting down new cards on the fly.

Indeed, one can simply draw out (using a felt-tip marker) patterns of the type shown in the figures above; so one need not have a computer (or keyboard) to "write a program" in this case, since one can actually write out a readable sequence of cards by hand. The key point for now is that ambient programming implies a type of programming activity that looks and feels quite different from the traditional method of desktop composition.

2.8 Code Tracks

Code Tracks is a computationally enhanced construction kit consisting of programmable trains, tracks, and small pieces of colored felt. The trains, equipped with color-reading sensors in their undercarriage, can distinguish among a variety of colors as they pass over the tracks, and subsequently take action based on the colors they sense. Some of the actions can be to move forward, stop, choo-choo, toggle a headlamp on and off, or trigger a track change. To use the system, the

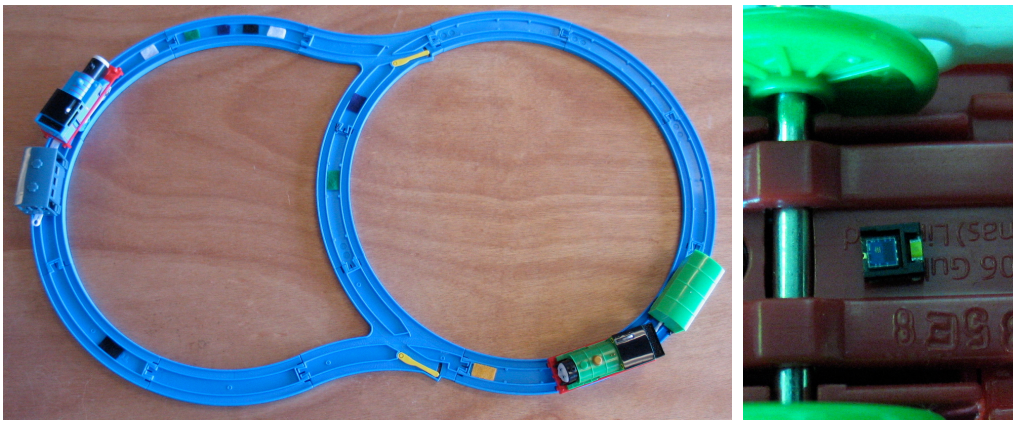

Figure 2.40: At left, two trains on a section of track, with pieces of felt as instructions. As the train goes over the pieces, it recognizes the colors and takes different actions depending on the color. At right, underneath the engine, showing the color sensor.

programmer first assembles the tracks (some of which are straight, curved, or branches), puts down pieces of colored felt on the assembled track, and finally places the trains on the tracks. The pieces of felt can be placed and removed as desired while the train runs on the tracks.

Trains have three different tiers of functionality: first they carry out a single primitive instruction per piece of felt; second they provide a means for combining and abstracting procedures on the track; third, they can read in more expressive programs from the computer screen than can be expressed exclusively in felt.

Figure 2.41 shows a train about to read two expressions from the track. As it passes over the green piece of felt `(toggle-headlamp)`, it toggles the state of its headlamp which becomes lit if previously off, and vice versa. As it continues and passes over the purple piece `(choo-choo)`, the train emits a "choo choo" sound. A complete list of actions is provided in Table2.2.

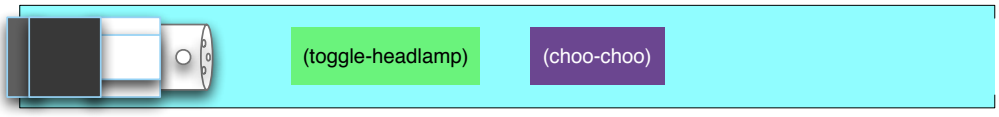

Figure 2.41: A train about to read two expressions from the track.

Some procedures take arguments, so some colors can be combined in a few simple ways. For

instance, by preceding the purple piece `(choo-choo)` with a brown `(repeat)` piece, one would hear "choo-choo choo-choo". Repeats can be nested, so if one was to place two `(repeat)` pieces before the `(choo-choo)`, one would hear the "choo-choo" sound 4 times; i.e. `(repeat (repeat (choo-choo)))`.

Other actions can also be repeated: one would write `(repeat (pause))` to pause for longer than three seconds; `(repeat (toggle-headlamp))`; to rapidly flash the headlamp, and `(repeat (branch))` to take the next two branches.

There are two colors that cause the trains to ignore the immediately succeeding instruction. The first, `ignore-next-when-dark` (red) causes the train to conditionally ignore the next instruction it encounters depending on the level of ambient light: if the level of light is below a (predetermined) threshold, skip that next instruction, otherwise perform it. This causes the trains to only perform certain actions depending on environmental factors, useful for, say, taking an alternate path only in the dark.

The second, `ignore-next-until-fourth` is multifaceted and depends on several implicit and explicit variables: initially, each time this instruction is encountered, an internal (implied) counter is incremented and the next instruction is ignored, up until the fourth pass. On the fourth encounter, the heretofore ignored instruction is now executed, and the count reset to zero. This has the effect of giving the train some memory, where it will only perform certain actions every 4 times it goes past a particular point. Its use is demonstrated in Figure 2.42.

One can also create new procedures on the track by combining existing procedures and

color	meaning
black	stop the train for 3 seconds, then resume moving
purple	emit a choo-choo sound
green	toggle the onboard headlamp
yellow	choose an alternate path (branch)
brown	repeat a subsequent instruction
red	ignore the next instruction if the train is in the dark
pink	ignore the next instruction based on internal counter

Table 2.2: Matching colors to default train actions.

giving the collection a name. To do so, one makes use of an additional color (white), in the manner described in Figure 2.43. The first color after `white` is the new name of the procedure being defined, and subsequent colors until the closing `white` are the actions to be performed by this newly defined procedure.

The arrangement at the top of Figure 2.43, read from left to right as the train passes over it, yields the expression: `WHITE PINK GREEN GREEN WHITE`, which if typed out more conventionally would look like:

```
(define (flash-lights) ((toggle-headlamp) (toggle-headlamp)))
```

After the train runs over and reads this, the programmer removes the pieces of felt used for procedure definition and need only use the single color piece; in this case the train will now call that procedure whenever it encounters the color pink.

Newly defined procedures can themselves be used in the definitions of new ones (Figure 2.44), including those that reuse the same color:

The color white is reserved for procedure definitions and cannot be redefined or changed in the manners just described.

The train language allows one to remove all bindings to a given color for user-defined pro-

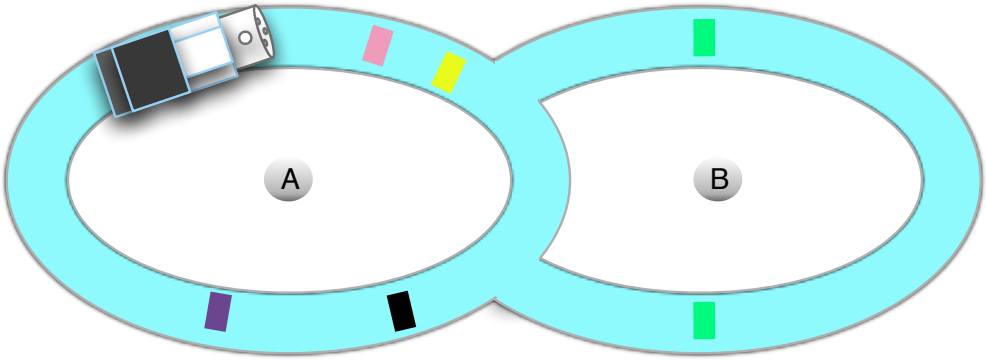

Figure 2.42: The train will circle in loop A three times, each time incrementing its internal counter when it encounters the `ignore-next-until-fourth` command (pink), and therefore ignore the `branch` instruction (yellow). On the fourth pass, it takes the branch into loop B, resets the counter and returns to loop A.

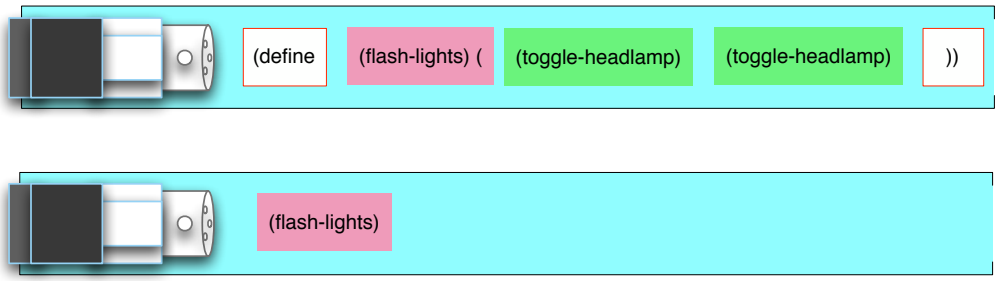

Figure 2.43: Top, the train being told to toggle its headlamp when it see pink. Bottom, calling the (new) pink procedure; support pieces (white, green) are no longer needed.

cedures. To do so, one puts down two white pieces, followed by the color being undefined, as in Figure 2.45. This "unravels" all definitions and returns the color to its original built-in meaning.

Because the trains use Schemer, one can also write programs at the desktop, and have them run on the tracks. There is a much larger repertoire of commands, as well as control structures that are simply not available in felt form. The Code Tracks system allows programmers to blend the expressiveness of text with the informality and immediacy of manipulating felt pieces.

The point of these examples is to show that one can write Schemer programs that read in integers, lists, procedures, or other types of objects from a variety of places. The only constraint (consistent with all Scheme programming) is that these read in objects must be used appropriately within the surrounding program – e.g., one cannot try to take the numeric sum of a number and a procedure.

In summary, Schemer allows programmers to write programs in a traditional desktop setting, augment or replace them "in situ" with programs scattered throughout the environment (from sheets of barcode, pieces of felt, specially coded stockings, other computational devices, music, etc.) It also allows them to opportunistically gather input for running programs using these same means.

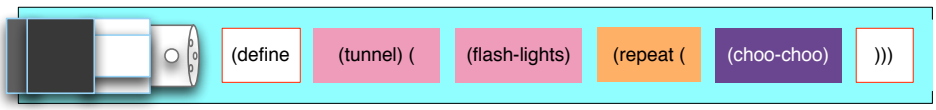

Figure 2.44: The color pink is being used in a new definition of itself.

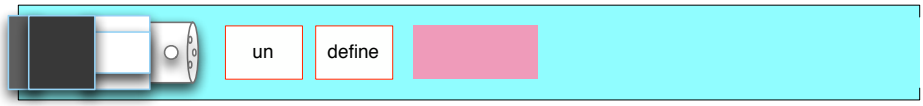

Figure 2.45: The color pink being "undefined", restoring its original built-in meaning.

These features represent a significant departure from previous work, which have not explored this potentially fruitful avenue.

In principle, any owner of a toy such as Duckie, or Code Train could have easy access to a range of programming environments for the toy, available over the World Wide Web (as it happens, the current language system is available over the Web, though in an early stage, at: www.aniomagic.com/schemer). This development environment is also available to those with mobile devices, since it runs within a web browser via Javascript. The low-level details of how this all works is explained in Chapter 4 - Technical Implementation, but now we turn our attention to discussing some similarities and differences among the various prototype systems.

2.9 Discussion of Prototypes

Having presented the prototypes in the previous sections, the time has come to discuss their attributes, comparing and contrasting them in order to distill the core ideas that each contributes to the concept of ambient programming. We start with a brief summary of each one.

Mimeolight is a programming-by-demonstration artifact, allowing its user to reprogram it by demonstrating the patterns of lights she wants, without ever writing a line of program text. It can also be used as a holder for programs to be transmitted to other computational items.

Learning Sensors allow the user to customize parameters to a running sensor pattern, turning on outputs depending on ambient temperatures, light, etc. They can be combined and cascaded to form more sophisticated systems, and can also be combined with full-fledged computers for greater functionality.

Pendantif gently fades between two endpoint colors for its default program. The colors are picked up from the environment and shaking it determines the speed of the program. It can also

read new programs from appropriately encoded cards, and transmit its color to other devices.

Schemer is a tiny general-purpose computational device, programmable with a fully symbolic language which can be input from a variety of sources. Expressions and entire programs are represented by flashes from the screen, barcodes on paper, colored pieces of felt, and short melodies.

The first of three Schemer-powered systems, Duckie, is a customizable stuffed toy able to perform different actions depending on what it reads from the computer screen, or from barcodes. The other two, Code Road and Code Tracks (also powered by Schemer), integrate programming with tangible construction. Programs are distributed in the environment as barcoded paper or colored pieces of felt. The expressiveness of train programs can be extended by writing them on the screen, to be run on the track.

These prototypes show how we may extend the traditional landscape and nature of programming. One might (e.g.) use Schemer to read in a program directly from the screen on a handheld computer or laptop. Or one might sweep a bracelet along a surface on which an appropriately encoded program has been written. One might also record environmental data like sequences of light flashes, and use that as input to a cellular automaton running on a wall.

Together, they also illustrate the idea of embedding "low-tech" programming instructions in physical objects, and demonstrate the feasibility of mixing textual programming, demonstrative programming, physical parameter setting and informal (hand-drawn) programming.

The upshot of this design is that one might read in programs and data from all sorts of serendipitous sources - a friend's laptop, a "program mural" on a wall, a printed-out pattern on stockings - to customize a computational wearable or robot. The placement of programs for such a device can be made informal, creative, and practically ubiquitous.

It is worth noting that the programming medium of these systems (except Mimeolight and Learning Sensor) is a true symbolic language, with sequences written and composed by the user; it is not (e.g.) a programming-by-example system [17] (with all the problems of interpretation that that notion usually entails[101]), nor is it a system in which notions such as control structures, data or procedure abstraction are inherently problematic[125]. Indeed, if the programmable car

trains were further augmented with a means of writing out symbols on the surface underneath it (admittedly, a technically ambitious step at this point), the system would clearly be capable of any computation performed by a Turing machine.

In the case of Code Road and Code Tracks systems, the programs are for the most part externally visible. Here, the car and train act like the program counter in a general purpose computer, in that one can actually trace program progression and see the operations being carried out (branch if lit, pause, etc).

By blending textual, pictorial, and demonstrative elements of procedures and data that can be read into an ambient device, it should be possible to create expressive programs in a way that dovetails nicely with the very notion of ambient, embedded computation: it is "everywhere", portable, interactive, and situated in the user's physical context.

System	primary interface	primary use	secondary	editable
Mimeolight	hand waving (light)	prog. by example	storage	no
Learning Sensor	electrical hookup	smart sensor	none	no
Pendantif	color, cards	color changing gem	storage	no
Schemer	screen(light), sound	programmable button	none	yes
Duckie	barcodes, screen	programmable toy	none	yes
Code Road	cards	programmable car	none	yes
Code Tracks	colors, screen	programmable train	none	yes

Table 2.3: Summary of prototypes and their capabilities.

Although these prototypes are meant to be used individually, there are some cases where using them in combination yields some interesting, complex and fascinating behavior.

For instance, Learning sensors can be connected to Mimeolights to create dynamic patterns in response to changes in (say) temperature or light. The dancing grasshopper of Figure 2.46 is one such desktop toy that performs a pre-programmed chirp and "dance" when it experiences a certain temperature range. It consists of a Mimeolight (which provides the pattern), and a learning sensor (which tells it when to dance). Power is provided by the solar cell on its back. The dance pattern is programmed by waving a hand at the grasshopper in the manner previously described, and movement is provided by a motor connected to the Mimeolight's output. The temperature-

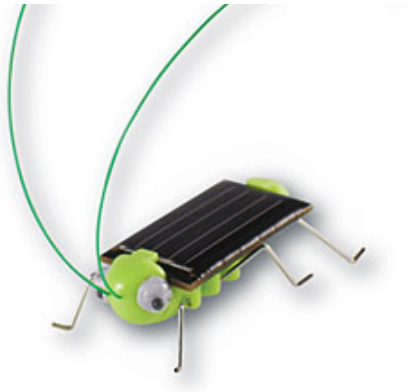

Figure 2.46: A dancing grasshopper that puts on a preprogrammed performance only when there is light and the temperature is "just right." It is a modified version of the "Frightened Grasshopper" commercially available from [80]. Photo © Educational Innovations[80].

sensitive learning sensor has been calibrated to turn on when the ambient temperature is at a certain level, and to turn off when it is too hot or too cold.

In effect, the grasshopper "program" can be thought of as distributed programs, each running on a different device, but communicating. The end-user has effectively constructed a complex program (from two smaller programs) whose final function depends on how the two pieces are physically put together.

Mimeolight also opens up other fascinating scenarios, ones in which "little nuggets" of complex programs can be embedded in tangible media. It combines Mimeolight's primary ability to record and playback light patterns, with fact that the Schemer browser-based environment sends new programs as pulses of lights. One can write relatively long (or complex) procedures at the desktop – the kind that would be hard to write with barcodes – and store them within a number Mimeolights. Subsequently, armed with a library of code in tangible form, one can send these procedures to others' Schemers, at a later time, without needing a computer. Programs can also be sent to Duckie, Code Road, and Code Tracks in the same way.

Imagine another scenario for participatory artwork, where the artist embeds Mimeolights into the gallery's wall, so that visitors' computational bracelets and t-shirts react (light up, beep, or

shake) when they are held close to the artist's embedded programs. All visitors would have to do is hold their artifact in front of Mimeolights; the interaction is pretty much the same as with the screen. If instead, the Mimeolights were equipped with beepers, and t-shirts with sound sensors, the patterns and programs can be transmitted with audible sound.

These prototypes illustrate a larger theme in programming: namely, that programs can be symbolic artifacts placed in one's physical environment in all sorts of interesting ways. In the examples, the "program" is represented in various ways: as cards spread about the floor for the car, as pieces of felt on train tracks, as color patterns on wallpaper, as a sequence of musical notes, flashing lights, or a tactile pattern of textures on a wall.

The key point for now is that ambient programming implies a type of programming activity that looks and feels quite different from the traditional method of desktop composition. Programs may (depending on the example) be placed around a room, drawn by hand, scrawled onto a wall, changed by whistling particular tunes, and so forth. We will return to the ramifications of these ideas in chapter 5.

Chapter 3

Related Work

The prototype systems from Chapter 2 build upon – and have been influenced by – a rich tradition of research to make computation tangible. No matter the motivating problem or particular details of implementation, this style of computation emphasizes (to varying degrees) a number of concepts: tangibility and materiality of the interface, physical embodiment of data, and the embedding of the interface in real spaces. Traditionally, the intent has been educational with a particular focus on children and novice programmers, e.g. to make programming more accessible, or to make powerful ideas like feedback and recursion more salient. In the past decade, the intent has broadened (if not shifted) towards the creation of toolkits and techniques that a wider audience can use to create their own personal embedded-computing projects.

Much of the early work in this area derived from a hobby culture in robotics; often, small computers were used to control model creatures or vehicles built from materials such as Lego. One prominent example – designed for use by children – is the MIT Media Lab "cricket" device [96]. It is a small, easily programmable computational device that can be incorporated into tangible objects (e.g., Lego constructions) and connected to a range of sensors and actuators to endow those objects with interesting user-specified behaviors. Another example of this genre – arguably more powerful and complex – is the Arduino microprocessor system[3], aimed at an audience of hobbyists or professional designers. The Arduino system and its offshoots such as the LilyPad Arduino[9] are again small computers for incorporation into physical objects (the LilyPad is particularly suited for adding computation to textiles). Generally, these systems may be characterized as a suite of devices

centered on a microprocessor, accompanied by a variety of compatible sensors and actuators.

Without a doubt, these kind of devices – and the end-users who design with them – are sparking novel ideas in embedded and pervasive computing and represent a profound advance in democratizing pervasive computing. However, there are two key factors that suggest that they represent only the first steps in exploring what promises to be a much larger territory of democratized design for pervasive computing. First, they are still conceived (and structured) along the lines of traditional (if highly shrunken) desktop computers: a central processor is augmented by ports to which sensors and actuators may be connected. Though this notion is very powerful, and will continue to be so, it is not the only way of approaching hobbyist- or novice-friendly pervasive computing. Indeed, hobbyist-created projects cataloged on websites like [3], [81], and [68] suggest that there might more fruitful avenues of approaching hobbyist- or novice-friendly pervasive computing. Second, there is nothing particularly pervasive about the way these devices are programmed: they still require connection to a desktop computer from which programs are written, debugged and downloaded. In other words, there is a decoupling between the embedded system being programmed and the programming activity itself. By contrast, the notion of ambient programming reconceives programming as an opportunistic, physically active, and playful activity, situated within the very same physical context as the pervasive systems being built.

Such notions have surfaced in a variety of projects focusing on (e.g.) tangible interfaces and educational computing: Schweikardt and Gross's roBlocks[124], Raffle et al.'s Topobo [42], Buechley's Boda Blocks[8], Watanabe et al.'s ActiveCube[57], and Wyeth's Electronic Blocks[149] are salient examples. (See [125] for a good survey.)

Tangible programming – that is, composing and executing programs in physical space as opposed to writing them entirely within a computer – is an idea dating as far back as the late 1960s when Radia Perlman created the Slot Machine device[113] to enable young children to write physical programs for a robotic turtle. Since then, there have been a variety of tangible programming systems that in one way or another blend programming with physical activity.

The related work discussed (and/or briefly introduced) in this chapter, in one way or an-

other, attempts to eliminate the distance between the computational and physical world by making behavior directly manipulable, and we see these notions repeatedly. There are several reasons for the recurring popularity of tangible programming systems: they are easy to understand and manipulate; they are more inviting and provide better support for active collaboration; and they make computing concepts more intuitive. See [106] for a comprehensive discussion about the arguments for tangible programming and computational construction kits.

Some work features program-by-example systems, "programmable" in the sense that they record and replay actions performed by the user. These include Topobo, a 3D constructive assembly with kinetic memory; a light that can play back a sequence of light and dark [59]; and Curlybot[55], a wheeled dome that remembers how it was dragged across a surface.

Others construct algorithmic structures from physical blocks. In other words, individual blocks represent tokens in a programming language that are then connected together to produce a program, which is subsequently downloaded to the computational object. This style is represented by AlgoBlocks[135], a set of a system of cubical aluminium blocks that when connected together generated small programs to be executed on a desktop computer; Programming Bricks[98], stackable lego-like bricks into which one can also slide "parameter cards" to change arguments to most procedures; and Tern[73], 3-dimensional wooden puzzle pieces to manipulate a robot's behavior. Rather than use embedded microcontrollers to determine program's structure, Tern uses a camera to take a snapshot of the arranged blocks and then compiles the text symbolized by the icons.

Other research projects are robots whose behaviors emerge from their embodied algorithmic structures. In other words, their "programming" emerges from the combination of sensors, actuators and logic gates. Prominent examples include Electronic Blocks[148] and roBlocks[124], each a set of sensor, logic and action blocks that can be combined to build constructions that (e.g.) drive forward in the presence of light or illuminate when touched; and Polybot[54] a collection of (mostly) homogenous motorized sections that are chained together to build crawling, walking robots.

Other types of robots change their behavior depending on cues in the environment. Examples of this kind include Johansson's Sniff, a stuffed toy dog that reacts with sound and motion to tagged

objects; and Tieben et. al's Actdress[56] a modified Roomba (automated floor vacuuming robot) whose behavior is modified by the tags attached to it (e.g. a shy tag causes the robot to hide under furniture, and a "curious" tag encourages it to go out into open areas). In both cases, the robot contains an Radio Frequency IDentification (RFID) reader, and the objects to be detected contain RFID tags that help the robot identify which tagged object it has encountered.

This chapter discusses five projects that are representative of the different styles of tangible and embodied interaction. In many cases the projects and styles overlap, and although they present different approaches to tangible programming, the ideas inherent in them are not necessarily exclusive. At the end, there is a brief summary of some other projects that fall into some of these categories, which space precludes us from discussing in detail.

3.1 Tortis Button box - tangible programming

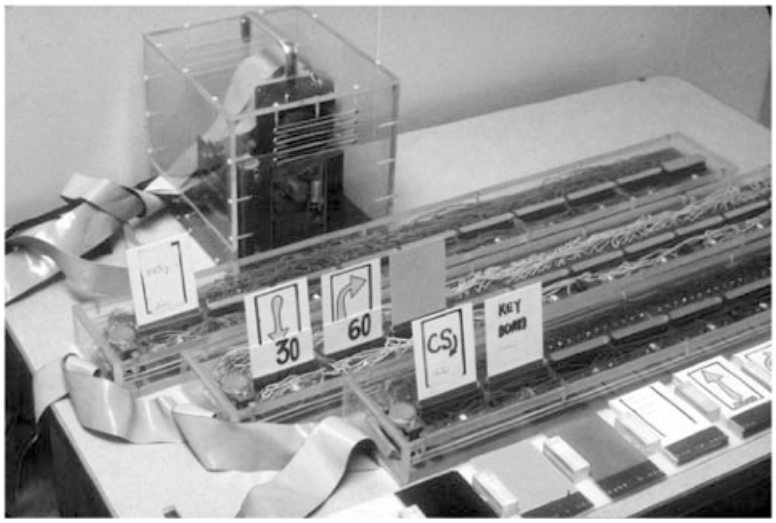

Figure 3.1: Radia Perlman's Slot Machine, used to generate programs for a floor turtle. Photo © Robert W. Lawler.

The Button Box and TORTIS slot machine were two tangible programming systems built to enable young children control a robotic turtle. The Logo turtle, first built in 1969, was a basketball-sized, dome-shaped robot that moved across the floor in response to simple commands like FORWARD, BACKWARD, LEFT, and RIGHT. It was tethered to a desktop computer at

which a user would type in the instructions before sending them. Often fitted with a pen, it was used to draw pictures on large sheets of paper placed on the floor. Children learned to write programs that made the turtle draw basic shapes (e.g. triangles, rectangles, and other n-sided polygons), and complex shapes mostly by repeatedly drawing basic shapes, rotating the turtle slightly before each repetition.

The Button Box had some simple controls that directly controlled the robot's actions and a few to record and replay sequences of actions. It had presented the user with four groups of buttons to directly control a turtle's actions: a number box from which one could select a number from 1-10; an action box from which one could select an action (FORWARD, BACKWARD, RIGHT, LEFT, PENUP, "toot your horn", etc.); a memory box with buttons for "start remembering", "do it", and "forget it"; and a four procedure box (YELLOW, RED, GREEN and BLUE) where the actions could be recorded. To generate the same program as above, a child would press the "start remembering button", then the YELLOW button (meaning store the following into the YELLOW function), and then proceed to press the FORWARD, 1, and RIGHT buttons four times. At the end, she would press the "do it" button causing the turtle to begin running the YELLOW procedure.

This system still had a two major shortcomings[113]: programs could not be modified once written, and children found the concept of procedure calling difficult. The next system, Slot Machine, was created in response. It consisted of four color-coded racks of slots in which cards could be inserted, side-by-side. Each card represented a command with a fixed parameter (e.g. forward 10). Other racks (procedures) could be invoked with a colored card. On the left side of each rack was a "Do it" button. When this button was pressed, the turtle performed the actions pictured on each card in the rack, in sequence from left to right, and lights under each slot would indicate which card's action was being executed.

The Slot Machine design was rather sophisticated in that it enabled a number of key abilities: one could manipulate a program directly, by adding, rearranging, and removing cards by hand; one could build abstractions by calling other procedures (for instance if a blue card was encountered, program execution continued at the beginning of the blue rack and returned to the original rack

after the last card on the blue rack was completed); this idea also enabled one to write tail recursive programs (e.g. a yellow card on a yellow rack called itself over and over again.

3.2 Crickets - general purpose embedded computers

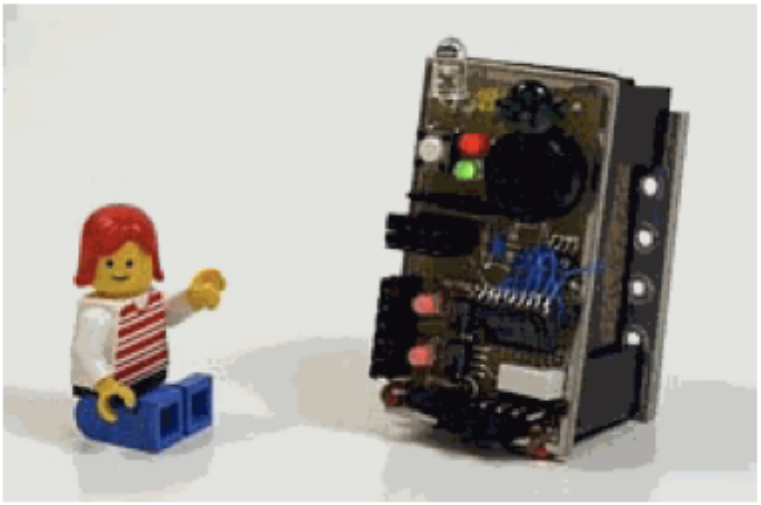

Figure 3.2: MIT Cricket

Crickets[51], are a class of computational bricks the size of a 9-volt battery that can be programmed using the Logo language[110] and can be combined with a variety of sensors and actuators to make dynamic constructions. Unlike the floor turtles however, crickets contain computation and storage and can run programs independent of the computer that was used to write programs.

Generally, they can control two motors, read two analog sensors, beep and play musical notes, send commands to other peripherals via a serial bus, and communicate with other Crickets via infrared light. To program them, one writes Logo programs on a personal computer in the same manner as the floor turtle, except the programs are downloaded to a Cricket using an infrared link rather than a physical wire. The user can also interactively send commands which are immediately executed by the Cricket, an extremely useful feature for debugging and incremental testing.

The Crickets have been used to create robots and other varieties of interactive kinetic structures, ensembles of small general-purpose communicating computers, and telemetry stations that

periodically record and transmit environmental data. A number of commercial products, like Lego Mindstorms[90], Handyboard[95] and Arduino[3] are variations of this theme of programmable brick (central processor) to which programs are downloaded.

3.3 roBlocks - distributed computational construction kit

Figure 3.3: A robot built with roBlocks construction pieces.

"roBlocks" [126], is a computational construction kit with which children can easily and quickly build fascinating, programmable kinetic structures. Traditional robot building generally involves a single centralized computational resource (like the Cricket above), which is then connected to simple sensors and actuators. However, roBlock modules have predefined behaviors that govern how they react when connected to power and to other blocks. Essentially, users program their robots by snapping the blocks - tiny bits of computation - together and as a result, the resulting program (what the robot does) emerges from the structure itself (what the robot is).

The kit has four categories of blocks: Sensor, Actuator, Operator, and Utility blocks. Sensor blocks incorporate a specific transducer and the computation to connect that sensor input to the roBlocks network. There are five sensor blocks: light, sound, touch, motion and distance. Actuator blocks produce the externally visible state of the robot: motion (translation, rotation, extension),

sound, and light. Operator blocks are the logic blocks that invert, sum, enhance or inhibit signals between sensor and actuator modules. Utility blocks provide power to the robot, communication to a PC via wire or wireless, or provide convenient (passive) routing of signals.

Figure 3.4: A catalog of roBlocks modules: sensors, actuators, logic and utility blocks.

These blocks can be snapped together, creating both a physical robot and the distributed system that controls it. For instance, one can create a simple robot that moves according to the amount of light in the environment by snapping a light sensor block to a motor block; in this case, the more light, the faster it moves, giving the impression of accelerating towards a light source. More complex constructions can be created with the logic and operator blocks which allow the user to selectively inhibit signal flow, invert sensor values, and manually set thresholds. Placing an invert logic block between the light sensor block and the motor block produces a robot that appears to run away from light.

Robotics continues to provide a compelling domain for experimenting with the design of complex systems, and construction kits like roBlocks sidestep a lot of the experience and expertise traditionally required to build robots with interesting behaviors. This work finds itself well situated among projects like Electronic Blocks[148], Polybot[54], and Braitenberg Bricks[6].

In addition to a younger audience, roBlocks is also aimed at undergraduate robotics and

control system education, where dealing with the low-level details of construction and electronics distracts students from being able to fully understand fundamental control theory ideas like distributed control or closed- and open-loop feedback. By constructing with the kits, students can concentrate on high-level theory and see it in practice, without getting bogged down with construction and programming details.

The idea of snapping blocks together also lends itself well to making the behavior of a robot physically apparent: for example, a robot with a closed-feedback loop will actually have a cycle in it where input from the sensor is both directly connected to an actuator and also "fed back" through some logic or operator blocks.

3.4 Topobo - kinetic programming by example

Figure 3.5: Manipulating Topobo. The two large blue blocks that form the body are the Actives that contain motors and computation to record motion.

"Topobo" [43], a unique example of physical programming by demonstration, is a lego-like computational construction kit that can record and play back motion. Certain pieces ("actives")

have kinetic memory that remember how they have been moved and twisted, and play back that motion. They have motors, sensors and memory to record and play back how they have been repositioned. Sensor modules ("backpacks") add sensor feedback and variable controls directly to the actives. Some backpacks are equipped with ambient sensors, such as light sensors (which allow the creatures to react to varying light conditions), and others have dials, buttons and levers that allow one to manually modify the speed, amplitude, timing, and direction of a recorded motion. One can also record several short snippets of motion and recombine them to produce different sequences of motion.

Technically then, Topobo is a distributed system comprised of individual elements each with their own internal parameters (e.g. speed) that define their behavior. Each Active has embedded motors and electronics to manage power distribution, motor control, and a custom distributed peer-to-peer network. To record a motion, one presses a button and manipulates the construction as desired. Pressing the button again stops recording and commences a looping playback mode. The recording can be saved to one of four nonvolatile memory banks to be used and recombined later.

With Topobo, users can gain insights into kinematic principles like relative motion, shifting centers of gravity, distributed coordination, controlled asynchrony, resonance, inverse kinematics, etc. that underlie why their creations exhibit certain behaviors. Using the backpacks, the kit also makes tangible some of the benefits of symbolic abstraction, feedback and behavior modulation.

In the evaluation of design tools, the learning curve and the potential complexity and sophistication of models created with such a tool are often brought up. Topobo appears to have multiple entry points for different learners, with many levels of complexity that can engage beginners who just want to produce motion, and more advanced users who want to manipulate the recorded motion.

Figure 3.6: Sniff reading an RFID tag.

3.5 Sniff - reactive toy

Sniff is a stuffed toy dog with the ability to "sniff" a variety of objects and react with sound and motion. Sniff's design incorporates an RFID reader in its large nose, which reads in the information contained in RFID tags embedded into everyday objects.

Sniff was designed as a personal toy that enables simple, playful interactions for young children, with a conceptual focus on social interaction: according to the author, "the dog is designed as a companion in daily situations as well as in play and games, alone or together with other children, and including one or more Sniffs." There are two contexts of use envisioned for the toy: playing melodies, and exhibiting different emotions and behaviors.

In the first case, Sniff is programmed to play back a small part of a well-know children's song when he sniffs the tags embedded into small pebbles. Entirely new variations in melody and rhythm can be created as the children pass Sniff over over the pebbles in different playful ways. With more than one Sniff toy, children can also experiment with harmony, singing-in-the-round, and accompaniment. This scenario is reminiscent of Neurosmith's Music Blocks[102], a set of five colored blocks (and base system) that play different phrases of well-known classical music pieces depending on how the blocks are arranged in the base.

The second case involves triggering different emotional and gestural responses when passed over stickers that have been embedded around a a child's physical space: on appliances, furniture, toys, and on other people. This way, Sniff can develop a fondness for some items, dislike and ambivalence about others, becoming a participant in a child's daily activities. Sniff can also sniff stickers placed in books, yelping and trembling with excitement, or whimpering and shivering in fear as he "reads" along with the child.

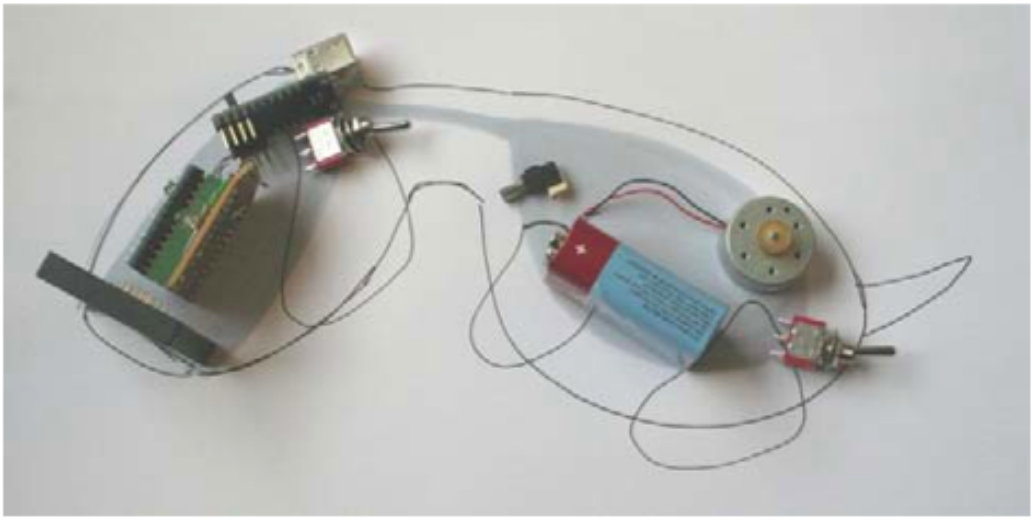

Figure 3.7: Underlying the soft plush exterior are (from left to right) the RFID reader, loudspeaker and amplifier board[78], Arduino[3] protyping board for computation, 9 volt battery, and vibrating motor.

Interaction with Sniff is not programming in the same vein as the other related work: its behaviors are already preprogrammed by the designer, and the user cannot create new ones. Nonetheless, the dog performs selected built-in procedures when it senses different tags, and this use of everyday items to influence the behavior of a computational object is part of the overall theme of ambient programming. There are other newer projects that are moving in this direction of full-fledged programming environments and ecosystems. For instance, Jay Silver's Twinkle[131], shares some resonance with ambient programming in that music (and programs) can be composed using crayons. The device contains a color sensor that can distinguish the different colors that represent different notes; by drawing and subsequently passing the device over it, one can compose musical

phrases and short programs.

There is a lot of additional related work that space precludes us from discussing here in detail; instead, we provide in Table 3.1 a non-exhaustive summary of related projects that are similar in style to the five systems highlighted in this chapter.

Project	Summary	Similar to
Programming Bricks[98]	tangible programming elements	Tortis
AlgoBlocks [135]	stackable blocks	Tortis
Tern[73]	program composition via computer vision	Tortis
ActiveCube[57]	input elements to create virtual 3D structures	Tortis
Tangicons[40]	tangible bricks used in interactive games	Tortis
BlinkM[136]	programmable multi-color lights	Crickets
Rototack[61]	programmable rotational tack	Crickets
Arduino	general purpose embedded computer	Crickets
Electronic popables[118]	programmable sheets of paper	Crickets
Curlybot[55]	record and playback motion	Topobo
Playback LED[59]	record and playback LED patterns	Topobo
Electronic Blocks[148]	sensor and actuation blocks	roBlocks
Braitenberg Creatures[38]	sensor and actuation pieces	roBlocks
Polybot[54]	linked homogeneous robots	roBlocks
Boda Blocks[8]	stackable 3D cellular automata cells	roBlocks
Actdress[56]	RFID tags to customize behavior	Sniff
Audiopad[45]	tag-based interface for musical performance	Sniff
Brio Trains[67]	RFID-sensing trains	Sniff

Table 3.1: Summarizing additional related work.

Traditionally, tangible programming systems have been criticized for being intuitive but too simple, limited by the range and power of expressive statements one might make. On the other hand, programming languages have been criticized for being sophisticated but (especially for beginners) hard to learn. Ambient programming is conceptualized as a new style of programming that blends the expressive power of text with the immediacy of tangible programming. It has been influenced tremendously by earlier work, but it moves the discussion forward in two major ways:

First, ambient programming emphasizes the idea of **full-fledged** programming through physical means. Artifacts like Schemer (and derivatives like the train and duck) can be programmed with a fully symbolic language without giving up the opportunistic nature of collecting nuggets of complex programs from tangible media. Since parts of programs can be easier to write on the

computer, and parts of it might be better input from whistled notes, the choice is left up to the end-user to decide where and how the programming activity takes place.

Second, one radically different notion about ambient programming is that it is not tied to a particular tangible object. Where other projects build example systems that are configurable in a particular way, the prototype systems from Chapter 2 suggest an ecology of inter-related systems. A Mimeolight can hold Schemer programs, or a Pendantif's colors might influence the running program on Schemer. They point to a different kind and style of programming that has been anticipated - but not yet articulated - by existing work.

There is a much wider ranging discussion in Chapter 5 of how this work breaks new ground. Before this discussion however, we will explain the technical implementation of our prototype systems.

Chapter 4

Technical Implementation

In this chapter, we focus on the technical implementation of the prototype systems introduced earlier in Chapter 2. We start with an overview of the prototype systems, touch on some of the design goals and associated constraints, and then proceed with an in-depth discussion of the various pieces of the ecosystem, also discussing how these items can be used with each other. Enough technical detail is provided to illustrate major design decisions, as well as the capabilities and limitations of the resulting implementation. For brevity, this chapter focuses exclusively on the implementation of Mimeolight, Learning Sensor, Pendantif, and Schemer, but not on the systems which incorporate them. In other words, there is extensive discussion of how Schemer works, but none about how Duckie or Code Road work beyond the description already given in Chapter 2.

The prototypes systems are conceived as small, inexpensive, and widely useful programmable physical objects, to be put into clothing, worn as jewelry, and otherwise embedded into a variety physical settings. Couched within these terms, the design of the prototype systems has been conducted with an eye toward a particular set of design goals. These goals are (in our view) representative of the larger genre of computationally-enhanced construction kits, and the prototypes fulfill these goals to varying degrees. Briefly, ambient computing artifacts should be:

- programmable (and re-programmable) by their users via a variety of means
- interactive, so that users may affect - for instance, via waving - the behaviors of constructions in action

- robust, suitable for wearing
- low-maintenance and low-cost
- simple in design, requiring relatively modest components

These general guidelines affect the design of the prototype systems, and we return to them throughout the design experience to inform the implementation. Under these guidelines, we find that each individual system is remarkably simple, in some cases requiring only a microcontroller and an LED, and even in more complex cases (e.g. Code Road) consisting of no more than 10 components.

4.1 Mimeolight

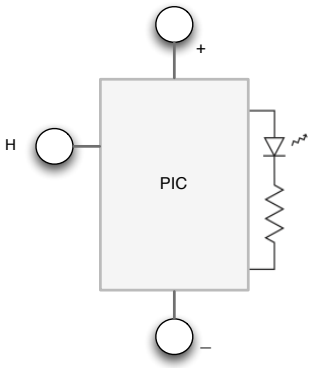

Figure 4.1: Schematic of Mimeolight showing how the LED is connected to pins on the microcontroller. The circles are the sewable tabs that connect the device to power and for customizing some behavior.

Mimeolight is a relatively simple circuit which couples a PIC microcontroller (12F675) with a light-emitting diode (LED). The PIC has 128 bytes of non-volatile memory from which it recalls prerecorded data. Data is stored as on-off patterns (corresponding to 1 or zero in memory), and are played back at regular 100ms intervals. As it marches through its memory, the PIC turns on the onboard LED for 100ms if it encounters a 1 and turns it off for the same amount of time if it encounters a 0. During the 100ms time period, while appearing to be lit continuously, the LED

actually flashes rapidly at a 50% duty cycle. The first half of the cycle provides illumination, and the second half provides Mimeolight with the opportunity to sample ambient light conditions.

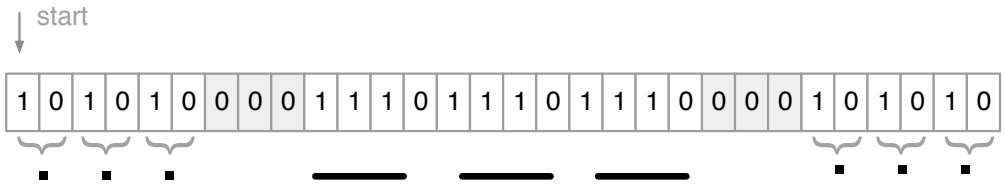

Figure 4.2: A Morse-code SOS pattern stored in memory. Each slot represents 100ms, where a 1 turns on the LED, and 0 turns it off.

To explain, an LED is a photoelectric device, emitting light when a voltage is put across it(forward biased), with positive on the anode, and negative on the cathode. On the other hand, it can act as a light-dependent capacitor when reverse biased. To do so, two pins on the controller are connected to the LED so it can reverse the voltage on both sides of the LED. To "sense", the microcontroller connects the LED anode to 0V, while simultaneously connecting its cathode to 3V. The LED becomes "charged up" and full of electrons, akin to a regular capacitor. Subsequently, the pin connected to the cathode is switched to input mode to monitor the charge on the LED. The speed at which the LED "discharges" depends on how much incident light hits it: the stronger the light, the faster the discharge. The microcontroller measures this time to determine whether it is in light or shadow. Figure 4.3 illustrates how this is done. Using an LED in this way, as popularized in [22], reduces the system's size and cost because there is no separate light sensor.

So, during one half of the on/off duty cycle, the light illuminates, and during the second half, it monitors ambient conditions. Mimeolight keeps track of the ambient conditions, and when it realizes that the ambient light is changing rapidly (e.g. someone is waving their hands at it or flickering a flashlight), it goes into recording mode. In this mode, it samples the amount of incident light (again every 100ms) and stores a 1 or 0 depending on the direction of change (i.e. if it senses a reduction in light, it stores a 0). Storing the **direction** of the light levels, rather than whether it passes over a given threshold, allows Mimeolight to operate in all manner of lighting conditions. While recording, it also uses the 50% duty cycle technique to turn the LED on and off, so the user

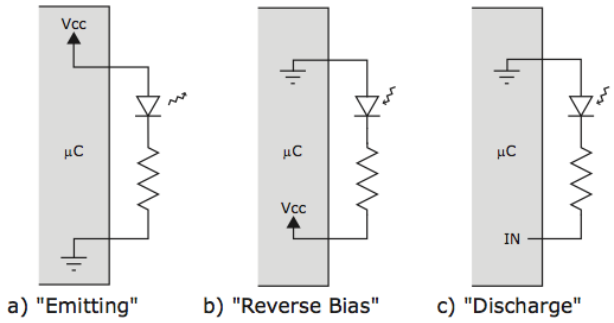

Figure 4.3: The light emission and detection cycle, taken from [22]. In (a), the LED is forward-biased which turns it on. In (b) it is reverse-biased which "charges" up any internal capacitance, and in (c) that capacitance is measured as a function of incident light.

can also see what is being programmed. If it detects no changes in light levels for about 3 seconds (or if it runs out of memory), it stops recording and proceeds to endlessly play back that pattern.

4.2 Learning Sensor

Learning Sensor combines a PIC microcontroller (also a 12F675) with one of 4 analog sensors; in the case of the counter version, it uses an on-board dial with 10 ticks. Although the methods of sensing are different, the sensed values are all translated into the same "sensor" value. In some cases (e.g. temperature), the sensor translation is direct, whereas in the counter case, each tick measured is multiplied by 10 to get an equivalent, scaled sensor value.

The temperature and custom sensor configurations use the PIC's internal analog-to-digital converter (A/D) to measure a voltage divider. Essentially the voltage divider consists of as few as two resistors, where one resistor is the sensor, and the other is a fixed value. In the case of the temperature sensor, both resistors have a nominal value of $4.7k\Omega$ at room temperature, which means the measured voltage is about half the source voltage. For the touch-sensitive version, one resistor is fixed at $1M\Omega$, and the other varies with the amount of resistance in a human finger. These voltage changes – corresponding to ambient temperature or galvanic skin response – are measured by the PIC's A/D converter to produce an 8-bit sensor value. The light-sensitive version

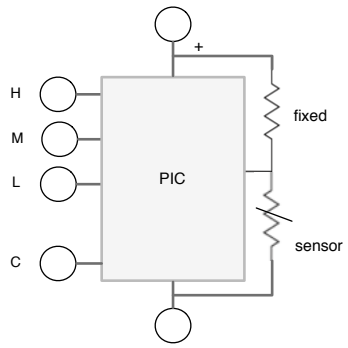

Figure 4.4: Schematic of the analog Learning Sensors. Although the specific sensor and fixed resistor values are different for the temperature, custom, and touch versions, the notion is the same: a variable resistor (sensor) is generally the bottom of the voltage divider network, and the fixed resistor the top.

uses an LED as a light sensor, along the same lines as the method employed by Mimeolight. Here, the time it takes for the LED to discharge is converted to a sensor value. In the counter version, the fixed resistance is provided by the dial, while the "variable resistance" is derived from the number of high-low-high pulses it receives.

For all cases then, the resulting program is the same: compare a desired sensor value (a "trip point") with currently sensed values. To condition the sensor to the ambient environment (to set the trip point), the user briefly connects the "calibrate" (C) port to the minus point. This value is stored as a single byte in EEPROM memory, so the sensor will "remember" this point even when power is removed. From then on, the sensor's response depends on whether sensed ambient values are higher than, at, or lower than this trip point by 10 points. In other words, if the trip point temperature is set to 65°F, the corresponding output would be turned on if the sensed value swung to 55°F or 75°F. There is some hysteresis around the switching points to prevent the outputs from swinging too often: if the sensed value was headed up and the H output switched on, the sensed value would need to swing down by (say) 3°F before it is switched off.

The outputs from any Learning Sensor are only "asserted" when high. What this means is, if an output is turned on, that output will provide current and voltage on that pin to turn on an LED, motor. If an output is turned off, however, it "floats", meaning the pin does not

apply a high or low voltage on that pin. This way, any number of Learning Sensor outputs can be connected together without causing electrical damage: if one any one is high (even if all others are low), the result will be a high. Outputs can also be connected to the sensor pins (S) of other Learning Sensors; by applying different voltages (high or low) to a point in the voltage divider, one sensor can dynamically influence the measured value on another sensor (the counter version counts the high-to-low transitions on its S input). This is the mechanism through which sensors can be cascaded.

4.3 Pendantif

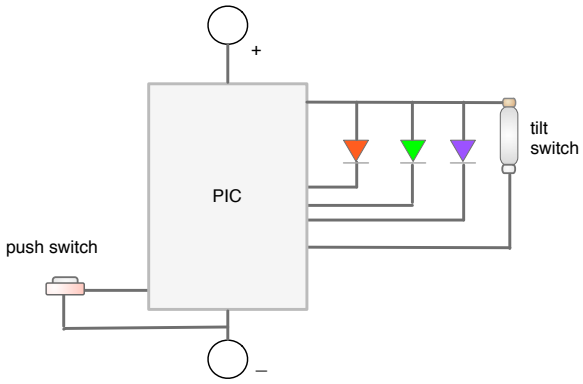

Figure 4.5: Pendantif Schematic showing the connections of the three colored LEDs, the tilt switch, and the push switch.

The Pendantif combines a microcontroller (again a PIC 12F675) with a tri-color LED, a light sensor, tilt sensor, and push switch. The whole assembly is mounted behind a glass pendant that also serves to diffuse the light. The LED consists of 3 separate color elements (red, green, blue) that can be independently powered to create different colors. By turning on the red and blue elements, one creates purple, for instance. By varying the timing of how long each element is lit (flickering as fast as 1kHz), one can also have finer control over the brightness and hue of the light emitted. The default program gradually changes between two endpoint colors, stored as a pair of 3-byte numbers (red-green-blue) in the PIC's non-volatile memory. Each byte corresponds to how long a particular element is lit. For instance, if the PIC first reads (say) 100, 0, and 100, it turns on the red and

blue elements 100% of the time, and leaves the green unlit (0% on); if it then reads (say) 0, 100, 0, the observed color gradually changes from the first (purple), to the second (green). The transition speed is controlled by a third variable, also stored in non-volatile memory.

To input a new pair of endpoint colors, the wearer places Pendantif in front of an everyday object (like a sweater), and then briefly presses and releases the push switch. This action causes Pendantif to sense the sweater's color and use that as the first endpoint color. He then puts Pendantif in front of another object, and repeats the switch press and release. This causes Pendantif to store this next input color as the second endpoint color. After this second press, Pendantif begins to run its regular program of transitioning from one enpoint color to the next, and back. If the user shakes Pendantif between presses (i.e. after capturing the first color, but before the next), the speed of shaking will be used to affect the speed of transitions: the faster the shake, the faster the program runs. This data is captured by the built in tilt sensor.

To detect the color of objects placed in front of it, Pendantif turns on 1 of the 3 output elements, and measures the amount of reflected light picked up by the light sensor. To illustrate, when the blue LED is lit, a blue object would reflect most of that light; however when a red LED is lit, a blue object would be dark. By turning on the different colors in sequence and detecting the amount of light received, Pendantif can estimate the color of the object.

To reprogram the pendant with new, expressive programs, one holds down the push switch for about 3 seconds, which informs Pendantif to look for rapidly changing patterns of light. These patterns can be transmitted from a computer screen, a user's flashlight, or by passing the object over a variety of barcode-bearing surfaces. Each pattern of codes is mapped to a specific procedure (out of 8) that the Pendant understands, and by arranging them on a surface, one essentially "writes" a program comprised of barcodes. There are two widths of barcodes, thick and thin, and white space between them delimits the bars. As pendantif passes over the cards, its light sensor is able to detect whether it is over a bar (dark), or a white section (light). Pendantif subsequenly translates these patterns of dark and light into tokens in its programming language.

4.4 Schemer

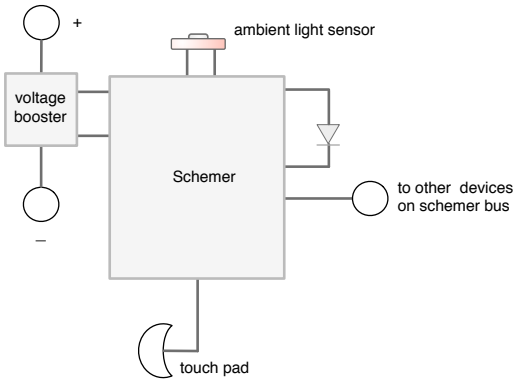

Figure 4.6: Schematic of Schemer; the voltage booster converts 3 Volts to a higher voltage needed for transmission on the bus. Also featured are the ambient light sensor (top), onboard LED (at right), and the metal touch pad (bottom). The hole on the right is output schemer bus that provides power and communication to other devices.

Unlike the more highly-specialized prototype systems (Mimeolight, Learning Sensor, or Pendantif), Schemer can be thought of a more general-purpose computer with connections to actuators, and to sensors through which it gathers data (some of which are bits of programs scattered around the environment). Once it has read in a program, that program can be used to activate lights, motors, or buzzers; it may also be employed to read other input besides programs (such as sensor values), or data gathered by a Mimeolight. When physically connected to peripheral devices, it communicates with them over a single-wire bus that provides power and data; otherwise it sends data wirelessly via light pulses.

The Schemer ecosystem consists of four identifiable parts: the Schemer computational board itself to which programs are downloaded and run; the source of the programs (e.g. the computer screen or barcoded card); and the sensors through which the programs can be input, and actuators through which their effects are often observed.

The design of Schemer called for a slightly more powerful PIC microcontroller than the ones in Mimeolight and Learning Sensor (16F684) because of the need for additional memory space for storing the interpreter and user programs, and for more pins to interface with accompanying

components like the light sensor, LED, touch pad, and voltage booster. Nonetheless, the resources on the chip are still quite modest, even by embedded system design standards, due to the need for low cost and small size.

As an aside, the voltage booster is used to provide a higher voltage on the bus and has several advantages: a higher voltage (6 Volts) means less I^2R power losses due to the resistance in the conductive thread typically used to connect components together; it also means brighter lights, more reliable communication over a noisy channel like conductive thread, and a better operating range for sensors. As all the prototype systems are meant to be powered by commonly available 3-volt coin-cell batteries; a voltage doubler allows the system to squeeze as much useful energy from the battery as possible. With it, for instance, a bracelet can flash continuously for 1 day.

The details for the communicating system is discussed further along in this chapter, but first, a brief overview of some of the various devices that can be connected to Schemer. There are primarily two classes of peripherals: output devices that produce motion and sound, display pictures, and display internal program state; and input devices that collect and record sensor data and new programs from the environment.

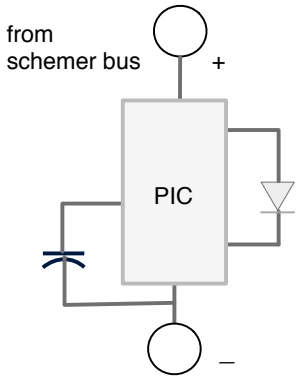

Figure 4.7: Schematic of a lightboard; the holes on top and bottom are connected to the Schemer bus that provides power and data communication to it. It turns the LED on and off in accordance to from Schemer. The capacitor on the left provides temporary energy to the chip while the bus is low.

Actuators, like the lightboard in Figure 4.7, generally consist of a tiny 6-pin PIC microcontroller (10F200) and the dedicated output device (in this case an LED). Voltage on the communica-

tion line is rectified with the chip's internal diode and filtered with the external capacitor to provide stable chip power. The other actuators are variations on this theme: for instance, rather than use an LED output that shines light of a particular color, the motorboard incorporates a motor capable of providing motion to mechanical automata, and chirper houses a speaker capable of playing a tone or musical motif. Regardless of the actual output, the behavior is programmatically similar: each actuator has a specific ID (1-5) which causes it to shine, shake or play a sound according to user programs. As previously mentioned, there are also other output devices like the music player and picture frame that do not have specific IDs. Instead, they cycle through 5 prerecorded songs or pictures corresponding to which actuator is currently active. The prototype picture frame is a slightly-modified version of a commercially available digital frame: its "next" and "previous" buttons are electrically connected to Schemer via a custom translation board.

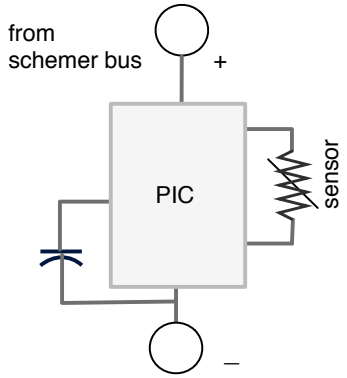

Figure 4.8: Schematic of a temperature sensor; the holes on top and bottom are connected to the Schemer bus that provides power and data communication to it. It translates analog values from the temperature-sensor (at right) chip to messages bound for Schemer. The capacitor on the left provides temporary energy to the chip while the bus is low.

Similarly, sensors, like the temperature sensor in Figure 4.8, feature a tiny 6-pin PIC with the same diode and capacitor rectifier arrangement. Instead of an output device however, the microcontroller (10F222) is coupled with a particular sensor; in this case, the microcontroller continually monitors the analog values produced by the temperature sensor and reports these values as messages to Schemer. Again, though the actual sensor might be different, the end-result (from a programmer's perspective) is the same. The sound sensor returns the amplitude of sound, and

the color sensor produces a different analog value for one of 9 colors it recognizes: e.g. it returns "0" when it encounters a black object, and "30" when it encounters a purple one.

4.4.1 Interpreter and Bytecodes

The main program that runs on Schemer is implemented as a virtual machine, which tightly integrates download and interpretation of user program instructions, task management, scheduling, and communication with other devices. Each Schemer "leaves the factory" with the virtual machine installed into the microcontroller's onboard 2-kilobyte flash memory, which is not accessible to the end-programmer, analogous to a standard computer's operating system kernel. User-customizable behavior is specified as a series of bytecodes stored in 128 bytes of onboard EEPROM, and program state is stored in 64 bytes of internal RAM. This system shares some similarities with other work like PIC/BIT[63], Armpit[105], Cricket Logo[96], PICBasicPro[76], and Basic Stamp[111], which all feature a tiny interpreter embedded into a microcontroller for which users write programs in a higher-level language.

There is a corresponding compiler - resident on the handheld or desktop computer - which parses the text of a user program, and produces the series of bytecode instructions which are then "flashed" to Schemer. The compiler essentially massages user text into a form more easily understood by the interpreter in Schemer; it also performs some syntax and semantic checking on the program before sending it on to Schemer. Incidentally, barcodes (or whistles, or color) are those same "compiled" instructions – merely in another form – but the user would need to consult a table that shows which barcodes, whistles, or colors correspond to which instructions.

Generally, Schemer's bytecode interpreter starts up by fetching, decoding, and executing instructions in its nonvolatile EEPROM memory. If it gets to the end of a program (or if the space was empty to begin with), it stops and waits for new expressions to evaluate. The interpreter can also be stopped and told to input new programs when the user touches the touch pad for 2 seconds.

While executing user-specified instructions, Schemer monitors its light sensor, and watches for a timed sequence of light-dark-light transitions that may indicate the beginning of a new pro-

gramming session. It also continually monitors the communication line, pinging for messages from other devices. Some of these messages can be used as input for running programs (say the current temperature level), or (when put in programming-mode) as bytecodes for new programs.

If the correct sequence of flashing lights or other variation in sensor data is received, the interpreter stops executing user code and remains in programming mode until the end of sequence pattern is received. In this mode, it opportunistically checks all available sensors for sequences of transitions that might contain new data or program. This is how one can, for instance, read in programs that have been encoded as flashing lights, as sound (using an external sound sensor), or as color (using the color sensor). After the end-of-sequence pattern is received – usually a closing brace ')' – control is then passed back to the virtual machine which starts executing these new instructions.

Light from the onboard light sensor is coupled to a resistor-capacitor network that makes the detector sensitive to small, low frequency (less than 200Hz) changes in light levels, yet able to filter out high frequency (85kHz) variations – the kind that occur in front of a computer screen. The ambient light sensor can also be powered down to conserve energy when Schemer enters its sleep mode. The other sensors do their own filtering and tracking on board; for instance the sound sensor performs some simple processing on the audio coming from the microphone, and produces messages that contain the analog voltage values in the same format as the built-in light sensor.

Schemer then keeps track of trends in the change of light (or sound, or temperature) levels, switching its internal state at local crests or troughs, independent of the actual amplitude of the signal. It uses the length of time between any transition (light-to-dark, or dark-to-light) to determine whether the bit received is marked as a 1, 0, or spurious (bits that don't fit within a window of time; see Figure 5). Specifically, a "1" is a change in trend of about 100mS, and a "0" is a change in trend of about 300mS. These turn out to be good values that take into account the maximum speed at which some LCD screens switch pixels on and off (CRTs being generally faster). They also turn out to be good values for Code Road as it passes over barcoded cards, and for Duckie when the barcodes are passed in front of it by hand. These bits are re-assembled in memory, and

when Schemer determines that the sender has stopped sending data, it passes control back to the Scheme interpreter to evaluate the newly acquired information.

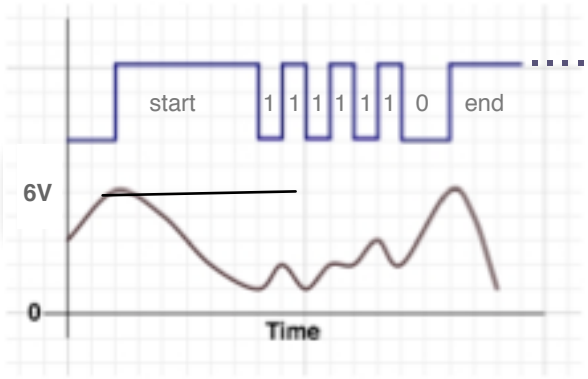

Figure 4.9: Turning trends into pulses: the bottom signal represents ambient sensor levels, while the top signal represents internal recognition of high-to-low or low-to-high trends. The sensor levels could be changes in light, sound, acceleration, etc. and Schemer extracts meaning from the timing between the transitions.

Our use of Scheme was encouraged by the existence of BIT[26]] and PICBit[63], two implementations for microcontrollers with very small memory resources (2 kilobytes in the case of PICBIT). Strictly speaking though, these are not interpreters per se: they are compilers – running on a PC – that take user programs written in Scheme, and generate low-level machine code that run directly on the microcontroller. Nevertheless, there are some good ideas about implementation that we borrowed from those projects.

The small size of user-accessible memory and the low bandwidth method of flashing the screen means programs must be short – it takes about 1 minute to transmit 64 **bytes**. As a result, Schemer implements only a skeletal Scheme environment, with many standard language elements omitted. Nevertheless, one can still write programs that read in integers, lists, procedures, or other types of objects. The only constraint (consistent with all Scheme programming) is that these read-in objects must be used appropriately within the surrounding program: e.g., one cannot try to take the numeric sum of a number and a procedure.

Such constraints on bandwidth and memory space require a number of compromises in the implementation of the interpreter. This had a particular and ever-present influence on the choices

made for what functions should be built in as high-level primitives, and how low-level other primitives could be.

Schemer-equipped artifacts are envisioned, unlike (say) the Crickets or Arduino, to be self-contained ensembles whose behavior is determined by a domain-constrained language. This notion of context-appropriate language is a recurring theme in ambient programming. For instance, a robotic car need only know about moving directions; a bracelet need only know to turn on different lights, change colors, or sample environmental data. By choosing the appropriate high level task-specific primitives for a language, programs can be short, while still allowing one to express ideas with the minimum amount of effort. In turn, powerful programs can be represented with short barcodes or melodies. Rather than force programmers to write programs thinking of registers, variables and ports, ambient programming allows them to think in terms of "turn the lights on when the switch is pressed."

Schemer comes with a number of such primitives built in; some of them are simple, doing things like turn lights on and off in a sequence like `(left 2)`, and some of them are rather sophisticated, like `(button-read)` which can evaluate environmental cues (e.g. light, color, melody) and produce representative Scheme symbols from those cues. For sure, there is a loss of flexibility when the primitives of a language are this specific, but the trade off is worth it, considering the types of programs one might write for ambient computing artifacts.

Figure 4.10: Picture of a bracelet with Schemer, four lights, and a switch.

Take the bracelet in Figure 4.10 for example; in addition to the centerpiece Schemer, it has four lightboards (two on either side), and a switch (furthest right). If one were to write a program to flash the lights in sequence, from left to right, using something like an Arduino, one would need

to write something along the lines of:

```
#define LEDS 5
int ledPins[LEDS] = {10, 11, 12, 13, 15};

void setup() {
  for (int i=0; i<LEDS; i++)
      pinMode(ledPins[i], OUTPUT);
}

void loop()
{
  for (int i=0; i<LEDS; i++) {
      digitalWrite(ledPins[i], HIGH);
      delay(1000);
      digitalWrite(ledPins[i], LOW);
      delay(1000);
  }
}
```

Creating this same effect with Schemer (a device built for flashing lights, among other things), the program is remarkably terse:

```
(left 2)
```

More important though, this program can be represented with a few barcodes, musical notes, or as a particular color. Of course, one could also "merely" call upon a pre-written library procedure in Arduino that does the same thing, but the crucial point is that in the tradition of full blown programming systems, this program would take up almost 500 bytes (a whole host of library functions would need to be transmitted to the device, in addition to the single procedure call), whereas the Schemer version takes up exactly 7 bytes (3 for the procedure call, 4 for the argument). This is one reason why there needs to be a reappraisal of how we (re)program ambient devices, and the choices we make for the prototype systems are always in favor of high level task-specific primitives that permit a degree of immediacy and informality in programming that is difficult to achieve by other means.

The interpreter's counterpart, the bytecode generator, is a Javascript-based parser and pre-compiler that runs in any graphical Web browser. It parses program text, checks for errors, and

gives user feedback if it finds problems with the expressions that have been typed in. If no errors are found, it creates a bytecode stream corresponding to the program and sends them to Schemer using specially timed flashes.

The Schemer system (the embedded interpreter and the Web-based bytecode generator) aggressively compacts object representation. Most procedures are encoded as 3-bit objects, and other symbols (numbers, colors, variables) as 4-bit objects. There are exceptions, but this approach ensures that the most frequently used symbols are represented compactly, which in turn keeps the bytecode count short for typical programs.

4.4.2 Communication bus

In most embedded systems, peripheral devices (e.g. LEDs, motors, and sensors) are connected to the central computational element via dedicated wires. In other words, one would need a pair of wires to connect to (say) one LED, and other pair for (say) a temperature sensor. Any additional devices would need their own wires, all of which need to be connected to different pins on the controller. The major disadvantages of this approach (which the Schemer bus attempts to address) are that the controller board ends up being large, and wiring up even simple projects becomes a complex undertaking. This situation becomes particularly acute within the context of sewing components into clothing where twisting and turning of fabric causes various strands of conductive thread to touch and short out in unpredictable ways. There is also the issue of having to track down and fix a multitude of break points.

The alternative is a system where all the components are connected using the same pair of wires, which can drastically reduce the size and complexity of wiring. There is a long history of this style of connectivity, some of which date back to the earlier days of computing (i.e. the data bus), where different devices (e.g. memory, disk drives, video cards, etc) were all connected to the central processing unit (CPU) over the same set of data and address lines. In the simplest case, all devices used the same 8 lines for data, another 8 for addressing, and one to tell it when to read or write. More recently, this notion of bus connectivity has been implemented with only two wires

(e.g i2c, 1-wire bus, Picocricket).

The Schemer bus system follows this practice, and connects all compatible components (sensors, actuators, data loggers, etc.) using the same pair of wires. As is done in similar systems, each device is preassigned a particular identification number, and Schemer interrogates each in turn. To further simplify wiring, the data line also provides power to the peripheral devices in a "charge-float-pulldown" cycle which we'll describe shortly.

In the current implementation, all members of the Schemer bus are participants in a global network where they all share a single communication line. This approach reduces the number of connections any component has to make, which further reduces pin-count. The current network topology borrows its architecture from the classic token-ring network (with a few changes that better suit the limited computational resources of these devices).

Before we continue, we offer a brief summary of the traditional token-ring network, highlighting where the Schemer bus differs in implementation. In this class of network, a token, a special bit pattern, travels around the circle of attached nodes. To send a message, a node takes possession of the token, attaches a message to it, and then lets it continue to travel around the network. Possession of the token grants the right to transmit. If a node receiving the token has no information to send, it passes the token to the next node unmodified. Each node can hold the token for a maximum period of time. If the node in possession of the token has information to transmit, it seizes the token, alters 1 bit of the token, appends its message, and sends this to the next node on the ring.

The information frame circulates the ring until it reaches the intended destination node, which copies the information for further processing. The information frame continues to circle the ring and is finally removed when it reaches the originating node. This node can check the returning frame to see whether the frame was seen and subsequently copied by the destination node.

While the information frame is circling the ring, no token is on the network, which means that other nodes wanting to transmit must wait, preventing the occurrence of collisions unlike in CSMA/CD networks (such as Ethernet). Token-passing networks are deterministic, which means

that it is possible to calculate the maximum time that will pass before any particular node will be capable of transmitting. Having a deterministic topology was one of the earlier goals of the Schemer bus, as programs rely on getting timely data from sensors (as does the act of collecting new programs); the elimination of collisions (and, by extension, collision detection software), makes this network topology more suitable for small microcontrollers.

Reproducing this protocol on the bus, each component is assigned a unique identification number. Actually, each **class** of component is assigned this ID. For instance, lightboards are numbered 1-5, and a "circle" sensor is numbered 10. All actuators, whether lights, motors or buzzers share the same ID and will "turn on" whenever they hear the message "ID**n** turn on", where n is its identification number. Similarly all circle sensors will respond when they hear "CIRCLE respond", regardless of their type (light, temperature, microphone), so it is important for the user to have only one such class of sensor in their design.

To communicate with items on the bus, Schemer first issues a token to address 0x13 (the switch), giving that component the opportunity to send a message (which tell whether it was pressed or not). After the switch has sent its message (or after a timeout if no reply is forthcoming), Schemer then issues token to other components in sequential order. When the token has made the rounds, Schemer then broadcasts a synchronize message to all components on the bus, advancing the collective state machine. From a sensor or lightboard's point of view, reception of the synchronize message is an assurance that its recipients have gotten its message. In the example shown in Figure 4.11, Schemer issues the token (indicated by the arrow labeled 1) to switch (0x13), which has nothing to transmit (because it has not been pressed). After a timeout period, the controller issues the token (2) to the sensor at address 0x14. This sensor has something to transmit, and sends its message (3). The message may be intended for a specific device (as a request for information, or a reply to another component's request), or may be broadcast to all the devices on the bus. When it hears sensor 0x14, Schemer increments its count and issues the token to sensor 0x15. Under this mechanism, all the devices will have have communicated with Schemer and with each other within a constant time frame.

Alas, this system does suffer from some limitations, and it would take a drastically different design to resolve these three remaining sticking points: First, since each component has to be addressed in turn, the time it takes for all members to communicate their states is directly proportional to the number of unique devices on the bus. It currently takes about 100 milliseconds for Schemer to make the rounds of 5 actuators, 2 sensors and some other components. Extending the design to accommodate more actuators or sensors increases the lower limit for the same communication round.

Second, as each component is pre-assigned a unique number upon "leaving the factory", the user cannot (say) add 5 more distinct actuators to make a longer string of lights. As it stands, if she adds more lightboards with the same address, they will simply all light up like the original set according to their address.

Third - and perhaps the most glaring - is that all sensors and actuators must be coupled with a microcontroller in order to participate in the bus. In other words, the end-user cannot readily incorporate an off-the-shelf component of her choice, she would first have to connect to a microcontroller and program it. This problem can be mitigated by the provision of "blank" schemer controller boards to which she can bolt on a new sensor.

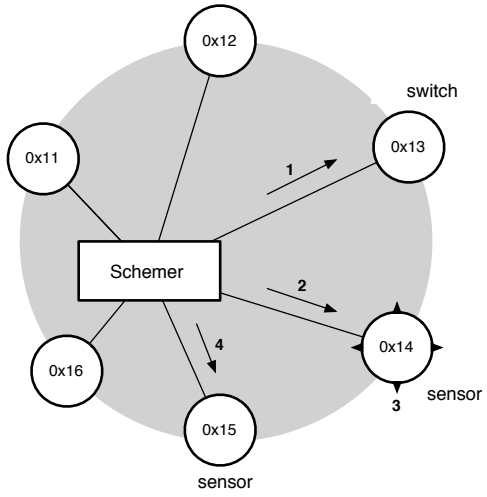

Figure 4.11: Example diagram of Schemer-bus messages; the lines represent Schemer conversing with particular devices; and the numbered arrows indicate the order, origin and destination of messages.

Messages sent on the bus consist of four pieces of information (each piece being a byte): the identification number of the device sending the message (sender address), the identification number of the device the message is being sent to (destination address), the message identifier, and an argument. The messages are lightweight, satisfying the requirement for low computational overhead, and are generic enough to exchange high-level instructions, including new programming code. In the current scheme, there can be up to 256 different entities on the bus. Of these, the following three addresses have special meanings: 0xF6 is Schemer's own address; messages with destination 0xFF are broadcast to all devices; the third (0xFE) is the address of an upcoming conduit which allows different Schemer buses to communicate over a distance, including over the Internet. Some of these messages can cause Schemer's virtual machine to fill its EEPROM with bits of new user programs. In order to determine whether the message is a "regular" or "programming" message, Schemer examines the most significant bits of the message identifier byte. If this bit is set, the remaining 7-bits in that message specify the EEPROM location to be changed, and the argument byte contains the bytecode of the instruction to be saved.

In summary, Schemer issues token messages, granting each device permission to send a message; after all devices have been thus queried, it sends the SYNCHRONIZE message which causes them to advance their internal state machine. Some devices can only receive messages (e.g. a lightboard responds to FADE ON), and some can respond with state or sensor data.

Without an external crystal, the internal oscillators in the microcontrollers that control each component are notoriously inaccurate, and can be affected by ambient temperature and voltage levels. In addition, they can differ in speed by as much as 10% due to the fabrication process. Indeed, although the addition of a crystal improves their accuracy to a level suitable for asynchronous serial communication such as RS232, some of the constraints imposed upon the design of the prototype systems (simplicity, low parts count, low-cost) necessitate the use of the internal oscillators by themselves. Use of a crystal would generally require 3 more components and consume two pins on the chip.

Standard communication protocols such as RS-232, IrDA, I2C, or USB either operate too

quickly or are too sensitive to variations in timing to be supported by these systems' internal clocks. Most communication protocols - including those mentioned above - require a minimum level of accuracy between the timing sources on the communicating devices. The RS-232 protocol, for instance, requires that the two communicating devices stay reasonably synchronized throughout the transmission of the entire byte (which can be ten or eleven bits long with control information). This requirement is satisfied with high-quality crystal oscillators; however the clocks in these system are inaccurate enough to make any connections using this protocol unreliable at best, and nonfunctional at worst.

Since the ability to remain synchronized for the duration of a character appears beyond the capability of the devices as designed, it was necessary to employ a protocol that could allow for re-synchronization every bit. One such is the return-to-zero pulse train: rather than represent ones and zeros as "low" and "high" levels of the line respectively, this scheme encodes each bit as a unique pulse whose length depends on whether a one or a zero is being sent, and separates each pulse with a short period during which the line is low.

When the line initially transitions from low to a high level, a listening device perceives this as the start of a new bit. The device then continuously examines the line until it returns to a low level. It then determines whether the pulse it just observed represents a one or a zero based on the length of the pulse. The length for a one is three times that for a zero, ensuring that the devices can understand each other, even if their timing sources differ by as much as 20%. However, if the pulse width is too long, the device assumes the communication was spurious or was terminated before completion. It then discards whatever bits it had accumulated since the end of a previous transmission, and returns to its main loop. In its quiescent state, the communication line is high. A device wishing to transmit pulls this line low between the pulses. It does this by providing a low impedance path to reference ground, absorbing current into its pin. Microcontrollers are typically much better at absorbing current that at providing it, and designing the communication line to be active-low ensures that a large number of devices will be able to share the same set of wires. Measurements show that the bus will be able to support up to 40 devices, more than enough for a

typical project.

Figure 4.12: Example of the low-level timing of a message signal. The sender of this message (device 21) is sending a message to the device with ID 3. Arrows mark the beginning of each field. Pulses representing a binary one are shaded for visibility.

The devices understand a message to be sequence of pulses that indicate a "break", a "one" bit, or a "zero" bit. A break is used to gain the attention of the receiver, and is very long compared to the pulses for a one or zero - long enough that most devices are able to catch a break signal by checking just once per cycle. Specifically, a break pulse lasts for 1000 microseconds while a one and zero take 120 and 40 microseconds respectively. The spaces between pulses are 40 microseconds long. A message is composed of a break signal plus the 32 bits used to encode the message. Figure 4.12 shows the voltage and time encoding of an example message. The devices switch from recognizing a space to recognizing a pulse around 0.7 Volts above the reference ground. This wide range of usable voltages, choice of timing, and ability to synchronize on each bit makes for a very forgiving communication system.

To power the devices shown in Figure 4.13, Schemer turns on its internal output transistors, which causes the line to be strongly tied high to 6 Volts, delivering power to all devices connected to it. The capacitors in each device charge through the diodes and provide stable power to each chip. To allow communications, Schemer then turns off those output transistors which allows the line to "float". This same line is now weakly tied high by internal pull-up resistors in each device. When a device wants to send information, it uses its I/O pin to pull the line low or goes into a high-impedance state which leaves the line high. By doing this within well-timed intervals, it is able to "send" data to the other chips sharing that same line. The diodes prevent the capacitors from discharging through the I/O pin while it is low. Note that the power available to each device is a diode drop lower than the bus voltage.

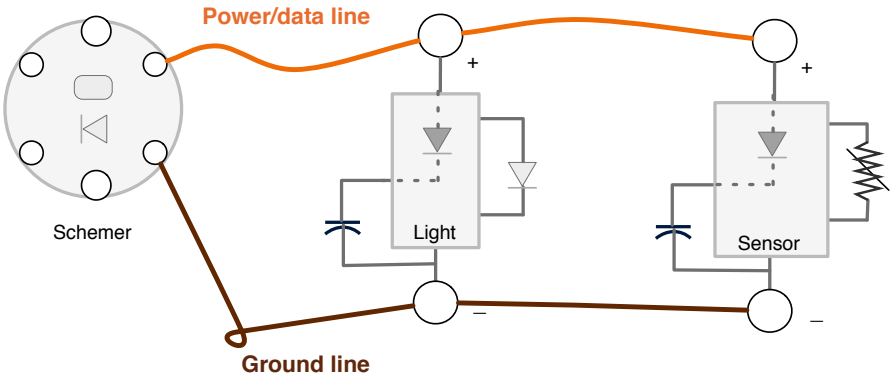

Figure 4.13: Diagram with Schemer and two devices, illustrating how the lightboard and sensor peripherals get power and data from Schemer via the single wire interface.

4.5 Current and Projected System Costs

Here is a good place to briefly provide some analysis from a cost perspective on the feasibility of our vision for ambient and wearable computing systems. Traditional craft artifacts like beads and buttons possess two particular attributes we sought to replicate with objects like Mimeolight and Learning Sensors: they are cheap and plentiful. These properties heavily influenced our design decisions, such as our focus on design simplicity, which further translated to low component count and modest computational resources. The resulting system cost of some of the prototype systems is profiled in Table 4.1.

All the pieces were made from off-the-shelf components available in medium quantities. Going to larger volumes of 50,000 or more would reduce prices by about 40%. It is also possible to trim these costs down significantly by using other types of microcontroller packages (e.g. the chip-bonded "epoxy blob" commonly found in greeting cards), or by combining the microcontroller and LED on a single die (some blinking LEDS are already packaged in this manner, though their functionality is fixed and they cannot be reprogrammed by the end-user). New advances in circuit substrates (e.g. circuits created on plastic or paper[118]) could pave the way towards circuits that may be cheaper the current copper-clad printed circuit boards. The chips arrive preprogrammed in bulk by the manufacturer; if programming is done "in-house" the chip costs would be 20% lower, although

Device	**sub-components**	cost per item at 1,000
Mimeolight		**1.22**
	12F675 microcontroller	0.76
	Blue LED	0.17
	Misc. components	0.01
	pcb	0.28
Temp. learning sensor		**1.19**
	12F675 microcontroller	0.77
	temperature sensor	0.11
	Misc. components	0.03
	pcb	0.28
Schemer		**3.24**
	16F684 microcontroller	1.32
	LM2766 voltage doubler	0.41
	Schottky diode	0.07
	Blue LED	0.17
	Light sensor	0.47
	Misc. components	0.08
	pcb	0.89
Lightboard		**1.00**
	10F200 microcontroller	0.63
	Blue LED	0.17
	Misc. components	0.02
	pcb	0.18
Switch		**1.29**
	10F200 microcontroller	0.63
	low-profile switch	0.31
	Misc. components	0.01
	pcb	0.34
Sound sensor		**5.13**
	10F222 microcontroller	0.83
	MCP6001UT opamp	0.24
	Microphone	0.57
	Schottky diode	0.07
	5 x 22μF capacitors	2.75
	Misc. components	0.05
	pcb	0.62

Table 4.1: Current cost per component of some prototype systems when produced in numbers of 1,000. Prices are quoted in USD and are current market prices as indicated by their respective suppliers. Labor, equipment, energy, and consumables are not included in the calculations.

the labor to first program them individually then place them on the boards would be significantly higher.

These cost projections suggest that it should be quite reasonable in the near future for one to speak of buying a "bucketful" of tiny wearable computational devices, in the way one can obtain beads and buttons from a craft store. Such a proposition becomes ever more feasible as component prices continue their relentless march downward.

The current prototypes, though at a very early stage, offer a very promising proof of concept. There are issues to work out, including limited program size, requirement for sufficient light, as well as adequate bandwidth for programming. These aside, together they hint at a much broader notion of "ambient computation" than has been traditionally purveyed.

Chapter 5

A Vision for Ambient Programming

Thanks to ambient computing, we are witnessing a proliferation of "interactive furnishings" – moderate-to-large-scale objects or furnishings with which one can interact in powerful or interesting ways. Indeed, we are seeing a wholesale re-imagining of the design of (among many other things) picture frames, vacuum cleaners, flower pots, window curtains, dresses, shoes, wallpaper, posters, children's mobiles, carpets, shelves, and so forth. There is a future being built, one where everything is endowed with interesting, beautiful, informative, or creativity-enhancing affordances generally associated with computation, and particularly with programming. This chapter describes our vision for this future, seen through the lens of ambient programming, and highlights some implications for democratized pervasive computing and for children's educational activities.

5.1 Themes of Ambient Programming

In the literature, there has been a traditional dichotomy between expressive, textual programming languages that can represent concepts like abstraction and recursion, versus tangible languages that of necessity, remain simple: for instance, there are only so many ways one can stack physical programming blocks before they get too unwieldy. It has also been difficult to do some things that we take for granted in a textual language, e.g. create a procedure from a set of primitives; pass parameters to procedures (see McNerney's Programming Bricks[98] for an elegant solution); or build anything but the simplest control structures. The prototype systems of Chapter 2 show how one can combine the best of both worlds, (i.e.) the expressivity of text with

the immediacy of tangible programming elements. They also illustrate some of the themes and goals of ambient programming that make it suitable for ambient computing. In this chapter, we elaborate on these themes, and further develop the case for ambient programming, using the different prototypes as occasional examples-to-think-with. We believe these systems are collectively poised to radically alter the appearance of ambient- (pervasive-, ubiquitous-) computing, end-user programming, computing for children, and the traditional debates around these topics.

5.1.1 Programmability, Reprogrammability, and Power

An essential element of the design of ambient computing artifacts is their controllability by their users. In most cases – since these artifacts will most likely derive their behavior from embedded computation – controllability is a rough synonym for programmability, and the range of expression can run the gamut from basic parameter setting to describing behavior with sophisticated abstractions. Traditionally though, (i.e. in the accepted way of programming computational artifacts), reprogramming even simple computational objects requires one to return to the desktop in order to write and download programs to them. This suffices as long as the user has the development environment and supporting hardware, but it leaves him stranded should he desire to change programs "in the field." Ironically, this scenario takes him out of the very physical context of use that ambient computing is intended for. We have seen novel exceptions to this tradition, but the prototype systems take the expressivity and power of tangible programming to a whole new level, showing how it can be done via a variety of input methods (e.g.) patterns of flashing lights, sheets of barcoded paper strewn about, pieces of felt arranged on a track, short melodious phrases, and so on.

However, the proper design of end-user programming systems remains remains a controversial subject (Nardi[101] has an excellent overview of the issues surrounding programming by end-users of computer applications). One of the major sources of contention is the issue of users' ability or willingness to learn the skills of programming. There is a widespread (although not universal; see [24]) perception that programming is a prohibitively difficult skill to learn. This thinking is

the motivation behind research in programming-by-example and other techniques developed to lower the barrier to program creation (e.g. the systems in [17]). Our belief is that many of the traditionally thorny issues in end-user programming (e.g., the ability of novices, especially children, to create complex programs, to understand large bodies of program text, etc.) lose some of their sting within the context of ambient computing artifacts. Artifacts such as Pendantif, Code Road, Schemer bracelet, or the dancing cricket rarely need a program of more than about a dozen lines to provide a great deal of sophistication and complexity.

More generally, since the artifacts that we imagine are envisioned as programmable physical objects, the types of programs written for them are likely to be tightly linked to a relatively constrained repertoire of physical actions. Consider what it might mean to write a program that "runs" within one of the "interactive furnishings" examples mentioned earlier, such as a mobile, or window-curtain, or carpet. Even very simple programs in these physical artifacts can give rise to profoundly interesting and complex behaviors because users can affect the programmed objects directly. For example, multiple Code Road users might interact with a running program in midstream by messing about the cards on the floor; Learning Sensor users can change, on-the-fly, how outputs are cascaded; a bracelet owner can whistle a new tune to interact with a running program. Further, the complex input of the surrounding environment might make a simple program behave in unpredictable ways: consider, e.g., how a "dancing cricket", "programmed window-curtain" or "picture frame" might react to different patterns of sun, shade, or wind, even if the artifact itself possessed only a simple program. In short, then, we believe that since ambient computational artifacts can be programmed with simple, short and understandable programs without sacrificing the potentiality of complex or interesting behaviors, the traditional controversies over end-user programming carry less weight in the context of programming physical objects.

5.1.2 Informality and Serendipity

The prototype systems illustrate another important theme in programming: namely, that programs can be symbolic artifacts placed in one's physical environment in all sorts of interesting

ways. In the Code Road, for example, the program for the car's behavior is represented as cards spread about the floor; one might imagine other types of scenarios in which a program is represented via color patterns on wallpaper, or a sequence of flashing lights, or the notes of a whistled tune, or a tactile pattern of textures on a wall, to name just a few. The idea of ambient programming suggests that programs can be constructed in informal, moment-to-moment ways. As already noted, one might alter a Code Road program by physically messing about with cards upon the floor, changing positions and putting down new cards. Going beyond this, one can simply draw out (using a felt-tip marker) patterns of the type shown in Figures 2.12 or 2.33; so one need not have a computer (or keyboard) to "write a program" in this case, but can actually write out a readable sequence of cards by hand. The key point is that ambient programming implies a type of programming activity that looks and feels quite different from the traditional method of desktop composition.

In principle, one might send symbolic information to a device such as Duckie, the Code Road's car or Code Track's train via (say) actual bar codes, or perhaps by some other means such as RFID tags placed within the environment. In these cases, naturally, the artifact would be equipped with a reader suitable for those alternatives. Such a design would be feasible, but it would sacrifice one of the benefits of ambient programming, which is precisely that the symbolic communication is not "invisible" or inaccessible to the user, but rather out in the open and available for casual inspection and alteration. In The Code Road, it is possible to write an instruction "by hand" on paper with a black felt marker. Indeed, one can alter an instruction in a similar way – for instance, by taking a set of three narrow bars (a "right turn" signal) and using a felt marker to thicken the central bar, one can change the instruction to a "left turn". This is the sort of playful, improvisatory relationship with code that is precisely rendered off-limits by means such as RFID tags.

A natural side effect of this sort of informal programming is that it can lead to all sorts of creative or serendipitous exploration of one's environment. Perhaps three strips of black tape stuck to the ground will serve as input to the car? (In fact, they will.) We discovered that on our laboratory floor, the prominent black borders between the tiles on the floor were also capable of being "read in" by the car – thus, if the car sped along over three tile borders, it would turn to the

right.

Figure 5.1: A photograph containing several embedded schemer programs. The programs are represented by the shades of light and shadow in the in bamboo forest. Photo © Rob Tilley[137].

Ambient programs also need not "look like code", (i.e.) highly structured text or plain black bars on white paper. Indeed, in a manner akin to steganography, one can embed programs into pictures, as is the case of the picture of the bamboo forest in Figure 5.1. The only caveat is that there has to be a "sensible" programmed pattern and that there is sufficient contrast between picture elements.

Indeed, along these sames lines of serendipity, Schemer programs need not even seem out of the ordinary. Take the railings from Figure 5.2, for instance. It is one of a number of temporary installations by the Paris-based art group Greyworld [66]. Essentially, the artists go around the world and (generally without permission) tune ordinary street railings so when one ran a stick along them, they would play songs like "The Girl from Ipanema", rather than the expected 'clack-clack-clack' sound. Since Schemer can also read in programs represented as simple melodies, one can imagine embedding programs in these railings, making them available only to those who happened to run a stick along the rails.

Ambient programs, spread and embedded in all sorts of ways in the everyday environment,

Figure 5.2: Specially tuned railings play a song when one runs a stick across them. The program is represented in the notes of the song. Photo © Greyworld[66].

are also open to different interpretations by the computational artifact that happens across them. In other words, how a particular set of symbols will be interpreted can vary depending on the interpreter placed within the artifact itself. For example, one car might choose to interpret three narrow bars as a right turn, another might choose to interpret it as a "pause" instruction; Duckie might choose to interpret is as "flap your wings twice and sing a song", or Pendantif might start a recursive color cycling program when passed over it, leading to a entirely different interpretation of the very same spatial pattern of symbols.

Ambient programming, by nature, then, has this element of opportunistic exploration to it - in contrast to the intense planning and organization associated with traditional programming.

5.1.3 The Ambient Ecosystem

Another important theme of ambient programming is the ability to mix and match otherwise simple components to create relatively complex systems. For instance, Learning sensors can be connected to each other – in a cascade – to create more complex computational sensor scenarios. To reiterate an earlier example, if the output of a light-sensitive learning sensor is connected to the sensor input (S) of a temperature-sensitive version (a light-temperature cascade), the end result is

a logical AND of two conditions: light and temperature have to be at certain levels in order for the attached actuator (e.g. LED or motor) to turn on. By physically cascading learning sensors, one can assemble each sensor's small program into a larger one, as shown in Figures 2.8 and 2.9 in Chapter 2. This "AND gate" represents one straightforward combination of learning sensors; but it is possible to create a wide range of sensor combinations. The distinct output ports (H, M, and L) can be cascaded in varying ways, connecting the output of one sensor to the input of another.

Learning sensors can also be connected to Mimeolights to create dynamic patterns in response to changes in sensor values. Rather than just turning actuators on or off, one can generate complex patterns of motion, sound and light by inserting a pre-programmed Mimeolight between the Learning Sensor and actuator, as exemplified by the dancing cricket of Figure 2.46. The creature performs a pre-programmed "chirp and dance" routine when it experiences a certain temperature range, which was programmed by waving one's hand over the integrated Mimeolight.

At the same time, Mimeolights and Learning Sensors may be employed as parts of still more elaborate projects that make use of fully-programmable components like Schemer. For example, with the Web programming environment, one can initially write and download different pieces of programs to a number of Mimeolights (in this use case, Mimeolights merely record the programs, and do not interpret them). Subsequently, the Mimeolights are cascaded with Learning Sensors and then embedded into the environment. The Learning Sensors determine which Mimeolights will be active, based on environmental cues like sound or temperature, or the number of "oscillations" of those cues. The end result is that the program a Schemer-equipped bracelet, or stuffed toy "sees" depends on what is happening in the environment it is placed in. There might be cases where one computational object may be unmovable (i.e. like a SmartTile installation embedded within a wall). In this case, devices like Mimeolight and Pendantif allow one to conveniently transfer programs from (say) a desktop screen to another ambient artifact. In other cases, a project might require physical means to change parameters to already running programs in Schemer-equipped artifacts. In this instance, one would write higher-order procedures in the textual language, then change parameters to those programs by moving physical sliders and pushing physical buttons on

the programming card mentioned earlier in Section 2.4.7.

The different prototype systems (Mimeolight, Learning Sensor, and Schemer) thus represent an **ecosystem** approach toward ambient, day-to-day programming, which enables one to read in programs from all sorts of serendipitous sources - a friend's laptop, a "program mural" on a wall, a printed-out pattern on paper - to customize (say) a computational wearable. Programs can thus show up in all sorts of creative and unexpected places in one's environment.

5.1.4 "Low-Tech" Ambient Code

One of the interesting things about ambient programming is the opportunity to place code symbols in unexpected places within one's surroundings and input those programs in all sorts of "low tech" ways. One can teach new flashing patterns to Mimeolight by hand waving, show new end-point colors to Pendantif, pass Duckie in front of a painting on the wall, whistle a "remind me in 30 minutes to go move my car" song to one's bracelet, or embed Code Road programs in floor tiles.

This is an intentionally "low-tech" manner of embedding programs in one's environment, and it makes something of a contrast – or complement – to (e.g.) standard practice in ambient computation or ubiquitous computing. Typically, in these styles of work, physical objects are augmented with computational devices, power sources, sensors, or actuators; and the result is that the environment is endowed with the capacity for executing computations (often outside the direct awareness of the nominal user, to whom the computation remains invisible). With ambient programming, by contrast, there is the possibility of placing programs – patterns of symbols – all about one's environment, without necessarily accompanying those symbols with embedded computational or digital devices. Programs, or portions of programs, may be written onto cards, scrawled as graffiti onto a wall, constructed as patterns of colors into sets of mosaic tiles, stitched onto curtains, or drawn on a favorite book cover. They may then be read and executed by a variety of computational devices at some later time.

Our choices for using "low tech" sources of input programs have come under repeated crit-

icism from researchers and users who feel that more "high tech" methods like Radio Frequency IDentification (RFID) or Bluetooth would be more appropriate. Without question, RFID technology is being used to great effect in the design of tangible, interactive artifacts (such as Sniff[84], Actdress[56], Brio Trains[67]), and for tracking (e.g. technical documents, passports). In fact, RFID has four features that would seem to make it suitable for our purposes of embedding code into everyday items: passive RFID tags do not require batteries; they are individually identifiable (each tag has a unique address); they can be detected over short distances (2-12 inches) without needing contact; and they are cheap enough to have many of them strewn about the environment. Bluetooth is a medium-range (2-30 feet) wireless technology primarily intended to connect general purpose computers to peripherals, though it is also used to interconnect "smart devices." In its favor are features like higher bandwidth and the ability to connect with many different devices from one place. Nonetheless, these technologies have drawbacks that make them unsuitable for ambient programming.

First, there are several important aspects of this more "ambient" style of programming that should be highlighted: programs must exist and be transmitted in many different ways, without being tied to a particular interface or technology; the programming materials should be low-tech and cheap enough for one to use a multitude of ways (including giving to and sharing with friends); one should be able to turn all sorts of new materials into program sources, even if it sometimes means fashioning a new sensor.

A wireless system like Bluetooth is designed for making connections between smart devices. The arguments in favor of it almost invariably assume that programming will be done exclusively on the screen of some device, which is precisely what ambient programming is NOT about. Programs can and should exist in a variety of ways (including smart phones and general purpose computers), but should not be tied to a particular technology.

As the prototypes demonstrate, a system whose primary input is temporal changes in a sensor value allows the same system to be programmed with flashing lights, barcodes, color, musical notes, magnetic flux, bumps, and so forth. Bluetooth and/or RFID would unnecessarily constrain

the flexibility of using different programming media. They also prevent end-programmers from easily creating new/duplicate instruction blocks, akin to a desktop programming environment that restricts the number of keywords one can use in a program (i.e. one could not just type new ones).

Within our vision of ambient programming, if the end-user programmer needs more pieces of syntax (e.g. 2 more branch conditionals), she can quickly cobble together a new command, using (for barcodes) a magic marker or (for, say, "braille"), plastic tabs, and so on. Ambient programming lends itself very well to creating and reprogramming with all sorts of informal physical materials. Since a program writer can see the barcodes and the colors, she can modify and create new ones on-the-fly with everyday items.

This sort of informality and spontaneity is difficult to accomplish with RFID or Bluetooth-based programming environments. In principle, one could get new RFID tags, but now one would need to to map those tags' 32-bit unique IDs to specific instructions. This would not be a trivial task, and – in the case of Schemer – require one to reprogram the virtual machine. It also goes against the grain of our notions about visibility: one can see three thin bars, or a collection of pieces of felt and associate them with a programmatic meaning; the same cannot be said about a collection of identical-looking RFID tags.

Because the sources of program and data can be low tech materials like paper and wood, they also have a number of significant advantages over high tech solutions. Paper does not require power, a Bluetooth radio does; optical sensors, tilt switches and bump sensors are cheap and tiny, RFID and Bluetooth are more expensive and bulky; a user can start swiping her new barcoded cards right away, with Bluetooth she would need to take intermediate steps like "pair devices" or install special software before doing any programming.

Thus, in our vision, an ambient program is a collection of low-tech symbols that can be easily created, modified, and extended with everyday materials and tools. In a deeper sense, ambient programming might be seen as a natural corollary to ambient computing: the advent of a plethora of small, embedded, and mobile computational devices facilitates the creative expansion of informal techniques for communicating symbolic information to those devices.

5.1.5 Programming as Playful Physical Activity and Construction

In most computer science curricula (and by extension, in most of industry), the activity of programming is associated not only with an explicit curriculum – of software engineering, language design, data structures, and the like – but also with what might be called an implicit curriculum. This implicit curriculum isn't taught directly, but it emerges as an unspoken, but pervasive, portrait of what programming looks and feels like. The programmer is, by assumption, a relatively sedentary creature who works exclusively at a computer keyboard (typically at a desk); the programmer is a model of intense concentration, focusing his or her complete attention on the complex demands of coding; the programmer makes frequent use of a variety of formal techniques and tools (debuggers, syntax checking, version maintenance systems, and the like) as a necessary element of their craft.

This portrait is in fact perfectly appropriate for a great deal of what programmers do, and for what they will undoubtedly continue to do in the foreseeable future. Nonetheless, it reflects only one plausible style of programming, much as symphonic composition (as opposed to, say, jazz improvisation, idly strumming a guitar, or humming a tune) reflects only one style of musical creativity. Ambient programming then allows us to alternatively imagine certain types of programming as small-scale; playfully woven into physical activity; and informal and experimental, rather than planful. In other words, we can begin to view programming as playful physical activity and construction, one better suited to the very idea of **ambient** computing. The prototype systems collectively illustrate several recurring themes of ambient programming:

- Embedding programming instructions inside or on the surfaces of day-to-day, physical objects;
- Emphasizing informal means (such as rapid drawing by hand) of "low-tech" program construction and alteration; and
- Blending physical activity and construction with program composition.

As we have seen in previous chapters, these notions are not new, and have surfaced in a variety

of places (e.g.) roBlocks, Tobopo, and Electronic Blocks. Artifacts like Code Road and Code Tracks extend this design space in that they provide explicit (though simple) symbolic programs, through their use of "low-tech" programming commands that can be (e.g.) drawn or painted by hand (as opposed to programming elements coupled with electronic blocks), and in their emphasis on "free-form" program creation that can take place incrementally and opportunistically over a large surface.

With Code Tracks, one constructs a program by first building a railway and then placing a sequence of felt pieces on the tracks with the intent of having the train pass over these pieces in turn. In this version of "programming", the activity of writing a program is akin to that of working in physical construction: the program is laid out spatially on the floor in much the same way that one might combine together (say) the elements of a model city or railroad. At the same time, this is not the only way of programming the train; more likely, one will find oneself rapidly slapping down instructions for the train it chugs along. (In testing the system, we have found ourselves racing ahead of the moving train to put down a pattern for it to "see".) In effect, the programming of the train is done energetically and playfully, as the program itself runs.

The style of work reflected in this sort of on-the-fly program construction could not be any further from the classical portrait of the stonily unmoving programmer. In ambient programming, the day-to-day environment (such as a large floor or table top) can readily become the expansive setting in which programming takes place.

5.2 Ambient Programming for Children

Historically, the term "educational computing" has meant, more or less, "software development for desktop machines." Within this desktop-oriented tradition, the great majority of educational computing artifacts can be classified within one of a dozen or so prominent genres (e.g., tutoring systems, simulation tools, design applications, games, programming languages). Despite the wide range in pedagogical philosophy suggested by these genres, they all share a basic, unshakable assumption of what "the computer" is – namely, a desktop box equipped with screen, keyboard

and (more recently) Internet connection. While this tradition of design has been (and continues to be) extremely productive, creating marvelous artifacts for children and adults, it is nonetheless inevitably constrained by its assumptions of the essential nature of computational media. Computers need not be boxes on desks, and ambient computing portends fascinating new possibilities for the design of educational computing artifacts. In many important respects, this re-examination of the field is already under way, with researchers exploring, for example, the possibilities of handheld and wearable devices [4, 148, 44, 9].

There are some really powerful concepts that can be taught through programming, but the manner of teaching in this powerful medium leaves a lot to be desired. In our exploratory work (and from reading the literature), we have found that children are more excited about topics that have personal meaning. From what we have been able to observe in the field, being able to customize how a bracelet or pendant behaves is a strong incentive for children to work and play within the domain. It also lends itself well to the socio-cultural aspects of child interaction: they learn tips, study how others made it, make special ones to give as gifts, etc. Rarely do we get this sort of engagement in the typical "computers in elementary school" scenarios; furthermore, there is a growing realization that the educational benefits of educational computing should transcend (say) writing reports, or browsing the Web.

Nonetheless, the question of whether (and if so, how) programming should be introduced to children continues to spark passionate debates among the educational computing research community. Proponents, on the one side, argue that children can be intellectually empowered by programming, and that important ideas in mathematics, software design, and engineering may be introduced to children using this medium. (See [109] and [119] for compelling arguments along these lines.) Opponents of children's programming, on the other side, argue that the medium is too difficult, too tedious, and perhaps too boring, for children to learn. Some go further to posit that programming is merely one of the many ways in which computer technology has largely proven to be a fruitless, (and perhaps even harmful) distraction to real education. (See [107] and [71] for instructive discourse from this point of view.)

Sidestepping some of the more thorny issues in these larger debates on educational computing, we find it more productive to take a middle ground that recognizes the potential of programming as a creative medium for children, while conceding that relatively few children seem to successfully appropriate this medium. Indeed, many prominent researchers have shared our enthusiasm for programming, but have also acknowledged the cognitive or motivational barriers that make the subject difficult for children. Some have responded by creating powerful programming environments intended in a variety of ways – graphical programming[48, 53, 71], programming by example[42, 55, 93], programming with spreadsheets [92], tangible programming elements[149], and so forth – to alleviate the difficulty of writing long or complex programs.

These innovations are themselves part of a larger tradition of research in end-user programming, where in this case, children represent a certain type of end-user – a type whose interests and cognitive abilities are mediated by age and experience. Good summaries of research in this area can be found in [17] and [101]: both sources deal with the subject of children's programming as a specific instance of end-user programming. A recent very brief account of the research issues surrounding end-user programming, also mentioning children's languages, can also be found in [100]. Two excellent recent discussions of programming language design specifically in an educational context can be found in [53] and [24].

Clearly the subject of children's programming is complex and controversial, but for our present purposes, it is worth focusing on one (usually unstated) assumption that has held sway in all previous traditional discussions of the subject: namely, that the act of programming itself is done at the desktop. That is, the notion of "children's programming" has virtually always been associated with the visual image of youngsters sitting before a desktop computer. We have seen some creative exceptions like Programming Blocks[149], or roBlocks[124], but still, most research has proceeded on the assumption that "programming" means working at a desk, in front of a screen. In most of these examples, the program itself is likewise something that runs on a desktop screen - e.g., a graphics program or interactive story; but even in those cases where the newly-created program runs in a setting away from the desktop (as in the Cricket programmable Lego brick and

its successors), the act of writing the program – the act of programming itself – takes place at the desktop.

The advent of ambient computing, with its plethora of computationally-enriched objects scattered throughout the day-to-day environment, suggests new and potentially quite powerful directions for children's programming. First, programming the behavior of a physical object – as in the programmable Lego brick – typically entails writing a program that is much shorter and simpler than the huge files encouraged by earlier media for children's programming. In part, this is because simple one-page programs can generate delightful effects when they control the behavior of physical objects. Moreover, programming physical objects meshes well with a large range of children's activities – activities that involve physical movement and play in a wide variety of settings.

One of the traditional barriers to end-user programming for children has been the relative sparseness of programming projects geared toward the children's interests. Ambient programming potentially offers the ability to create customized clothing (like the flashing t-shirts), as well as a myriad other programmable objects (trains, marionettes, mobiles, dioramas, just to name a few possibilities). We believe that there is a strong possibility of enhancing children's involvement in programming through the use of computationally-enhanced physical objects.

For the purposes of this discussion, we will focus our attention on Duckie because it demonstrates one style of design in which full-fledged programmability is made compatible with the aesthetics of traditional children's artifacts. It also illustrates what we believe to be a potentially fertile, powerful technique for making children's toys truly programmable. On the one hand, this style of design permits the use of a full-fledged programming environment, with the entire range of expression (conditionals, iteration, procedures, etc.) that such a medium might find profitable to include. On the other hand, the means by which programs can be sent to it are playful, informal, and aesthetically consistent with many traditions of (non-programmable) toys and furnishings.

Its design is along the same tradition of "furry computational toys" that could be said to have been pioneered by Druin's creation of the NOOBIE playstation[25]. More recent influences

have been the toy-design efforts of Fernaeus and Jacobsson [56] and of Johansson [84]: both these projects however make use of RFID tags to convey program information to a microprocessor placed within a toy. (In the former case, the tags are cleverly embedded within the toy's various costumes; in the latter case as noted earlier, the tags are placed within objects that can be detected by the "nose" of the delightful stuffed dog.) Duckie does not make use of RFID tags to convey programs, but instead permits the toy to read programs directly from a computer screen (or from a hand-written barcode passed in front of him); in our view, this permits a degree of immediacy and informality in programming the toy that would be difficult to achieve by other means.

One primary lesson learned from prior work is that it is more profitable to eschew low level primitives in favor of high level task-specific primitives that allow one to express ideas with the minimum amount of effort. Of course, there has to be a fine line between how high is too high, and how low is too low, and where this line is drawn varies from one prototype system to another. To illustrate, recall this example program from Chapter 2, where we told Duckie to occasionally (at random) flash the lights on his head and flap his wings:

```
(define (coin-flip)
    (flash-and-flap (+ 1 (random 2)))
    (coin-flip))
```

This program, though remarkably short, is typical of the sort of things one might want to do with Duckie, and (perhaps more importantly) conveniently brief enough to be represented with barcodes or physical blocks. Despite its brevity, it undoubtedly includes some fairly sophisticated ideas in the context of children's programming: `coin-flip` is a recursive procedure; it includes a call to an already-written `flash-and-flap` procedure; and that already-written procedure takes a numeric argument. All of these inclusions are deliberate: the point is that Duckie is capable of reading (and acting upon) programs with an effectively unbounded range of complexity and sophistication. The `coin-flip` procedure is notable in another respect, in that it includes a probabilistic element: during the running of his program, Duckie flashes-and-flaps once, sometimes twice. Notions of this sort have historically proven especially difficult to capture within scaled-down, tangible

programming systems, but is readily expressed within our system.

More generally, the notion of ambient programmability extends far beyond this particular example of Duckie. The larger point is that programming can be approached through the creation of short "snippets" of code in a wide variety of formats like (as we have seen) color, sound, and motion; and that those bits of code can then be used to enhance or control the behaviors of all sorts of tangible artifacts. In the process, the activity of programming may itself become steadily demystified for children: rather than an unapproachable discipline requiring massive time and effort (like composing a symphony), programming may be introduced through instances of informal, environmentally situated creation (like writing a folk song).

Finally, as applied to children's artifacts, the idea of ambient programmability suggests a variety of objects that could be endowed with interesting behaviors. One might imagine programmable garments and wearables, game pieces, trading cards, children's furnishings (such as mobiles or kinetic artwork), and many other examples. By permitting these physical objects to be easily augmented by naturally-transmitted programs, children's artifacts can potentially embody the best features of both the tangible and computational worlds.

Ambient computing artifacts represent a profound advance in democratizing pervasive computing, much as an earlier generation of home computers democratized the ideas and practice of computation itself. As as a generation of young and amateur users in the 1980's familiarized themselves with programming through (among others) the Apple II, the Commodore 64, and the Atari 800, so is a new generation experimenting with ideas of embedded and pervasive computing by creating tangible artifacts that employ existing platforms like the LEGO Cricket [96], Arduino[3], and LilyPad[9], and new artifacts like Schemer, Learning Sensor, and Mimeolight. The importance of this new landscape of devices – for empowering amateur users, for sparking novel ideas in computational design, for providing intellectual growth and pleasure for youngsters – can hardly be overstated. The vision of ambient programming – exemplified by our prototype systems – is one where small pieces of computation, and the recipes for influencing their behavior, are scattered throughout the environment. The pieces can be used individually within a project; they can also

be combined with each other and with general-purpose programming systems. Programs can be constructed in informal, moment-to-moment ways and can be embedded inside or on the surfaces of day-to-day, physical objects. This ecological approach of ambient programming, by its nature, suggests that artifacts should be designed not as "stand-alone", closed systems, but rather with an eye toward their interaction with a wide range of other devices. Ambient programming, then, is not merely a matter of making small, programmable, versions of the "classical" components of computational systems: small computers, small sensors, small actuators, small power sources. Rather, it encourages the creation of objects that enable their users to better control their environment, allowing for expansion and recombination in unexpected directions.

Chapter 6

Ongoing Work & Future Directions

6.1 Toward Democratized Pervasive Computing

The traditional notions of ambient computing are changing. The vision of a world in which we interact with ubiquitous computers – but never actually write programs for those computers themselves – is being replaced by one where (re)programming takes center stage in the interaction. The increasing availability of tiny, embedded computation – and the growing ranks of hobbyists who design with them – are challenging the rhetoric that computational artifacts should be invisible or transparent, forming the background of one's environment, relieving tedium and discomfort. However, as previously noted, the act of programming these artifacts remains largely an abstract, sedentary, and time-intensive activity, more in keeping with writing elaborate programs on a conventional desktop computer. By contrast, the notion of ambient programming reconceives this practice as informal, opportunistic, physically active, and playful. Though our prototype systems represent only first steps in exploring what might well be a much larger territory of democratized design for ambient computing, they should cause us to take a step back in order to rethink more appropriate ways for programming ambient computing artifacts.

The aesthetics and situational contexts of programming are also poised to change. Although it is unlikely that "classical" programming will disappear (and it should not, in our view), that style of programming will ultimately be seen as one spot in a much larger landscape of programming styles. Programming can be physical, tactile, sensually rich, aesthetically pleasing, or athletically demanding. One can combine the feel of wood, felt, paints, and papercrafts with programming. It

need not be a physically impoverished activity: one can write programs for Code Road by running about a room to slap down instructions ahead of the oncoming car; one can "wave" new programs to a computationally-enhanced quilt; one can create a painting that evokes different reactions in the ambient computing artifacts that pass over it. One can also participate in ambient programming through occasional, informal means – adding a bit of program to one's environment here or there, altering the symbols or meaning of a small chunk of program text – without making this activity the sole focus of one's day, in contrast to the traditional notions of programming as a solitary activity, requiring deep concentration and long periods of physical inactivity.

6.2 Ongoing Work

This section showcases the work of people who have incorporated versions of our prototypes (that we make commercially available) into their own ambient computing projects. Although we initially performed pilot tests of Mimeolight and Learning Sensor (mostly for feedback during the development process), we did not engage students in longer-term, closely-observed case studies. In lieu of these studies then, is a catalog of a number of projects that have been made during the course of the prototypes' development. Most of the projects were done individually by their designers, some in personal collaboration with the author, and others in larger workshops with multiple facilitators. The designers vary a great deal in educational background, age and programming experience: some are as young as 8 years old, some are experienced crafters who have never written a program before, and some are excellent programmers who are making a physical computing artifact for the first time. They also vary in purpose: some to create "purely aesthetic" projects; others use it to introduce programming (e.g. to children); and others to create systems for health-monitoring purposes. For the past two years, we have manufactured several of the prototype systems (and supporting pieces), and made them available for purchase through our company, Aniomagic. To date, we have sold about 1,500 Mimeolights, 300 Learning Sensors, and 1,200 Schemers, and have shipped them to 17 countries. Of the 1,200 Schemers sold, 520 of them were bundled in the "Bracelet Kit", a leather band which incorporates Schemer, lightboards, and a switch. Although we encourage customers to

publish their projects on various online forums (including ours), only a small, but vocal, number of customers have publicly documented their projects. Nonetheless, the projects we know about number in the hundreds, and we present a handful of representative examples here.

Although some of the projects were made in collaborative workshops, the vast majority of them were done by individuals, on their own, without involvement from the author, who had heard about the prototype systems either from others or by searching the Web. Table 6.1 gives a short summary of the level of involvement of the author in the various projects documented in this chapter.

Project	**Title (figure)**	**Involvement**
Moleque de Idéias	Interactive quilt, (6.1)	Direct involvement
Myriam	Cellphone cozy, (6.4)	Direct involvement
Anonymous	Bunny house, (6.5)	Direct involvement
Sabine	Salem's collar, (6.11)	Some assistance
CU Wellness Innovation Lab	Data logging bracelet, (6.2)	Some assistance
Swamy	Japanese inn, (6.6)	Some assistance
Mr. Shizuya	Beaded pendants, (6.7)	Arm's length
Kayobane	Light-up necklace, (6.9)	Arm's length
Hanakomet	Knitted heron, (6.10)	Arm's length
Osamu	Pacman cellphone strap, (6.13)	Arm's length
Osamu	Halloween mask, (6.16)	Arm's length
Angela	Fairy wings, (6.15)	Arm's length
Angela	Coffee cup holder, (6.12)	Arm's length

Table 6.1: Level of involvement of the author in the different projects: the first three projects were as a direct result of the author's participation in workshops; the middle three were done by the individuals with some level of assistance from the author; and the last set were done without involving the author.

We present some of this work to show how our ideas about this new style of programming – **ambient programming** – as put forth in this document are being disseminated, mostly through informal means.

6.2.1 Collaborative programming in the large

This 64-patch computationally-enhanced quilt was made during a workshop in Brazil. Each patch is decorated with high- and low-tech materials: paint, felt, buttons, sequins, Mimeolight,

Figure 6.1: 5-foot tall quilt of patches created in collaboration with Moleque de Idéias, an afterschool program in Niteroi, Brazil. Each patch is a computationally-enhanced piece furnished with Mimeolights (and optional buzzers and motors) that play prerecorded patterns in response to signals passing through the quilt.

Lightboards, motors and buzzers. By far our biggest workshop ever: 5 days, 10 core participants, including the author, and more than 300 collaborators aged 4 - 70. Most participants were passersby who spent just a few minutes sewing, ironing, decorating, or programming. Everyone was free to do something, even if they could only spend 5 minutes. An individual patch could have up to 10 authors, distributed in space and time.

Based on Leah Buechley's "Quilt Snaps"[7], each patch is a dynamic display, capable of producing dynamic patterns of light, sound, and vibration. Each patch is a piece of fabric containing a Mimeolight for computation, and output devices like lights, beepers, and buzzers. Each side has

connections to power and for communicating with other patches. The patches are sewn into an electronically active background canvas which provides power to all the patches. After decorating the patches with crafting and computational materials, the participants attached them into an electronically active background canvas with conductive thread (the background canvas provides power). Thus arranged in an 8x8 array, the patches can be connected to each other via removable arrow patches that direct the flow of signals through the quilt. As each patch is activated, it flashes a light, moves a motor, or sounds a buzzer and then sends the signal along to other patches. Some patches have touch strips that generate a signal in response to touch or light strips that sense the presence of light (from a flash light), so the patch can be activated by touch or light. The flashing (or motion, or buzzing) patterns that each patch displays were prerecorded into the Mimeolights (and can be recorded again when that particular Mimeolight is active). The overall result is a multi-piece quilt construction in which a path of light or sound travels from piece to piece, activating a dynamic pattern of light, sound, or motion as it moves along.

6.2.2 Monitoring and tracking personal health data

There is a growing interest in using information communication technology to enable people proactively manage their own health and wellness. Some of this involves capturing and displaying (at least privately) personal wellness data that can objectively show how active (or sedentary) a person is, or how many hours of continuous undisturbed sleep she gets each night. Here at CU's Wellness Innovation and Interaction Lab, researchers are investigating ways to capture health data with augmented low cost, wearable computational artifacts that unobtrusively sense and present wellness information to users in a playful, yet informative manner. One stated reason for this line of research is to address the childhood obesity epidemic, and they approach it in part by promoting a holistic view of exercise - every bit of physical movement counts towards one's physical activity goals and seeing this data presented in (e.g.) a personal artifact that one takes around everywhere reminds one of their goals, and reinforces the behavior to achieve those goals. The logging bracelet in Figure 6.2 is a modified version of the Schemer wristband previously introduced in Section 2.4.

Figure 6.2: Picture of the motion logging bracelet being worn. With the accelerometer and logging boards sewn in, the bracelet gives feedback on how much execise the wearer is getting. Photography - Morteira; production - ILhaDoUrsO; model - Nathalie Cohen.

In addition to the controller and lights, it also contains an accelerometer for capturing the wearer's motion, and a data logging board for storing that information. Before beginning a capturing session (which could last as long as a week), the wearer moves sliders on the Web-browser programming interface or the physical programming card to select her "activity target" and how fast she wants the data to be logged. For instance, if she chooses to walk 5 miles throughout the week, Schemer will indicate how close she is to her stated goal by turning on a single light for each mile she actually walks.

She can then download that information by connecting the data logging board to her computer. It is anticipated (in the near future) that this information will be uploaded to a website where she track trends of her activity, and possibly participates in collaborative fitness goals with friends, family and co-workers.

Although the many benefits of a good night's sleep are well known, the increasing availability of tiny, wearable personal monitoring systems is also making it easier for people to track their sleep patterns. Data and trends gathered by these devices can help one identify reasons why one might wake not feeling rested, or why it takes a long time to fall asleep. Armed with the bracelet, one user was able to determine that she was waking up at least 10 times per night. She took this information to her doctor who recommended a treatment for sleep apnea.

Figure 6.3: A backpack with a force sensor and lights to visually indicate bag weight. By the author.

The weight-sensitive backpack, shown in Figure 6.3, is one way to visually notify the wearer of how heavy the bag is. Although one can easily check the weight just by carrying it, there are (as at least one person has noted) times when one should not habitually carry things over a certain weight, even when they are capable of doing so (e.g.) under a doctor's advice during therapy after back surgery. The backpack takes its cue from this elegant weight-sensitive tote bag[91] which also incorporated vibration to warn of too much weight. The bag incorporates Schemer, lightboards of different colors, a force-sensitive resistor, and a custom sensor board. With the force sensor tucked under the shoulder strap, Schemer turns on different lightboards depending on the weight of the bag when carried: the white light is lit when there is a little load, the green light comes on with moderate load, and the red light turns on when the bag is too heavy. These settings can be easily

customized by pointing a Web-capable mobile phone directly at Schemer, as previously described.

6.2.3 Accessories, "Practical stuff", and other Ambient devices

The cellphone cozy in Figure 6.4 is one example of the kinds of computational objects hobbyists are building with the prototype systems. It features a couple of Mimeolights that have been sewn underneath the flowers, a battery, and a pair of magnetic snaps that complete the circuit when the case is shut. The Mimeolights have been programmed to briefly replay a prerecorded pattern when they turn on; to reprogram them with new patterns, the wearer need only wave her hands.

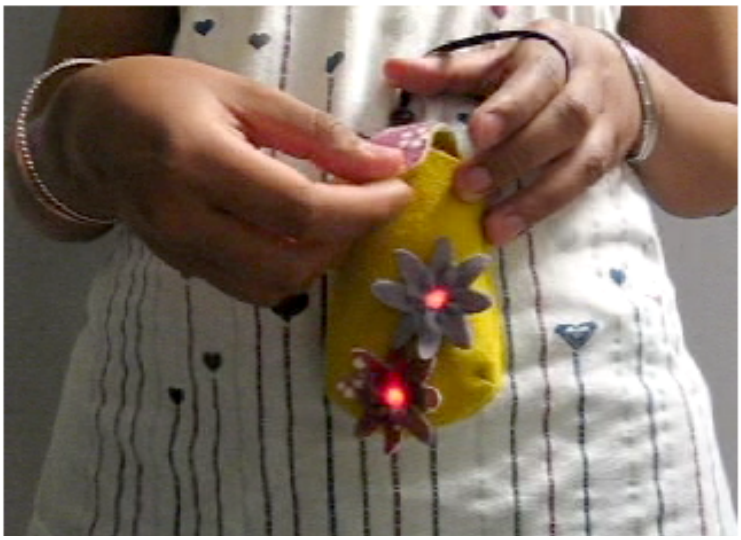

Figure 6.4: The Mimeolights embedded underneath the flowers flash briefly when the magnetic snaps are closed. The snaps are also conductive and act as a switch to power on the devices. By Myriam.

We have also found that people are increasingly making their own table-top ambient computing devices to provide feedback about environmental cues, like noise, temperature or humidity levels. For instance, using a sound sensor, in conjunction with Schemer and lightboards, one can make a playful doll house that responds to ambient levels of sound. The toy bunny house in Figure 6.5 is a modification to a commercially available foam assembly. The overall circuitry is much the same as the bracelet. In fact, most of the projects featuring Schemer are very similar in that they

Figure 6.5: At left, a picture of a sound-sensitive toy house. The flowers in front are lit up by lightboards in response to the (hidden) sound sensor, and the bunny at the right "shakes with excitement" when the environment gets really noisy. The picture on the right shows how the sound sensor is connected to the bus. By an 8-year old, with substantial help from the author.

either activate actuators under control of a programmed algorithm (e.g. randomly, sequentially); or they combine a sensor whose values determine which actuators are turned on, and when. Generally, when programmed to respond to environmental cues, Schemer will query any external sensor its connected to for messages containing (say) temperature or sound levels. If no such sensor is attached, it will use its own light sensor, making it responsive (at least) to changes in light levels. In a sense, the sensor program is opportunistic, using whichever sensor happens to be connected to it.

The model Japanese inn of Figure 6.6 is another take on this style of ambient computing. When connected to a temperature sensor, the internal lights, diffused by the paper window, shine different colors depending on ambient temperature levels: (e.g. red when hot, and blue when cold). Made from balsa wood and paper, it incorporates Schemer, lightboards, and an optional sensor. There are plans to connect it to the Web via an intermediary computer so it glows different colors in response to seismic activity.

Our prototype systems are also being incorporated into (in the author's opinion at least) highly fashionable pieces of jewelry, where fascinating patterns of light can be dynamically controlled

Figure 6.6: A model Japanese inn. Schemer and lightboards light up the insides, and additional sensors enable it to respond to temperature and sound. By Swamy.

by a user's touch or by the environment. Mr. Shizuya, a jewelry designer of renown in Japan has begun to incorporate Schemer into a line of jewelry, like the ones in Figure 6.7. A closeup picture of a pendant, in Figure 6.8, illustrates some of the detail.

Figure 6.7: A pair of beaded pendants on display on mannequin busts. By Mr. Shizuya.

Figure 6.9 shows another example of our prototypes being incorporated into "serious" jewelry. This time, the design includes a switch that allows the wearer to dynamically change parameters

Figure 6.8: A closeup showing the arrangement of components in one pendant. Schemer sits underneath the larger pendant in the middle – so it can still be reprogrammed – and is surrounded by 8 lightboards. Conductive thread connecting the assembly to a battery is cleverly hidden by passing it through the many tiny beads at the top.

Figure 6.9: At left, a beautiful necklace tastefully adorned with Schemer, lightboards, and a switch. At right, a closeup showing how the electronic components are integrated with traditional craft materials. By Kayobane.

to a running program.

As a reprise of Duckie, "Hanakomet" made this absolutely delightful knitted heron that glows from the inside. Like Duckie, it can be programmed from the desktop, or from barcodes passed in

Figure 6.10: Knitted heron. By Hanakomet.

front of it. Unlike Duckie however, this version does not flap its wings, though it does play a piece of music under the right conditions.

Figure 6.11: A stuffed toy with a touch-activated collar that flashes light patterns in the dark. A Learning Sensor and 6 Mimeolights are hidden underneath the felt; the heart-shaped pendant is a felt battery holder. By Sabine, with help from the author.

"Salem" (Figure 6.11), is a child's stuffed toy whose neck is adorned with a touch-activated flashing collar. When his collar is touched, it flashes a preprogrammed pattern for about 5 seconds, "expressing himself", according to his owner. The collar incorporates a Touch Learning sensor, which can be calibrated to turn on different outputs depending on how close a hand comes to it (i.e. it can distinguish among no touch, 1/2" approach, and full-on contact). The sensor is

subsequently attached to 6 Mimeolights that flash patterns that can later be changed when the child waves her hands over them.

Figure 6.12: Picture of the coffee cup holder being used at left; the temperature learning sensor is hidden behind the penguin's belly and the LEDs are his eyes. At right, a closeup showing how the components are sewn into the holder. By Angela.

The coffee cup holder of Figure 6.12 is one that warns the user when her drink is too hot. It comprises a temperature learning sensor, and a two-color LED that shines red when her coffee is too hot, and green when it is just right. She can recalibrate the temperature settings with a paper clip in the manner described earlier. She chose to make this project because "it is self-contained and you don't need to hook up any external microcontrollers to play around with it."

This decorative cellphone strap was made to celebrate 30 years since Pacman's introduction. It uses Schemer and 5 lightboards to simulate Pacman eating the ghost characters. The pattern is changed by holding the strap up to the cellphone screen; the speed at which the lights disappear into Pacman's mouth is changed by pressing the switch.

Figure 6.14 shows a vibrating robot – a "bristlebot" – that uses a light-sensitive learning sensor and a motor to move only under certain lighting conditions. When the motor (with an unbalanced weight, like in a cellphone) runs, it causes the assembly to skitter across a surface. Although the concept has been demonstrated by hobby roboticists for quite some time, the toothbrush version was recently popularized by Evil Mad Scientist Laboratories[127].

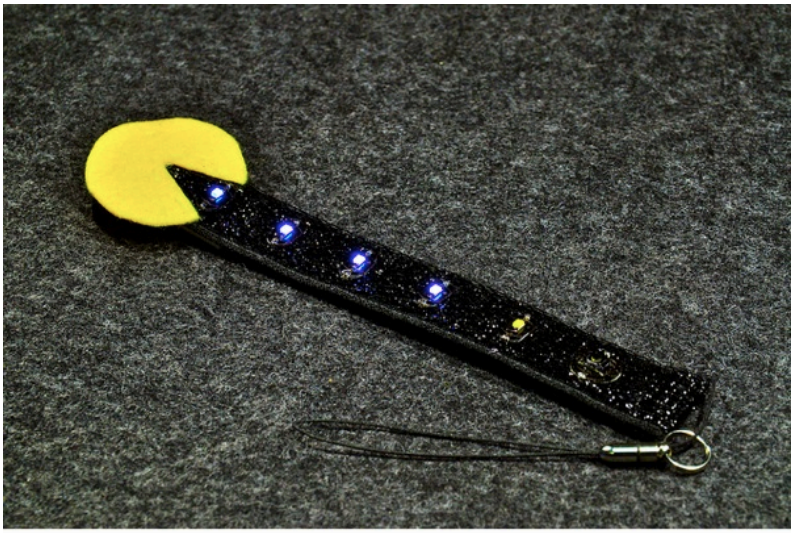

Figure 6.13: Picture of the cellphone strap featuring Pacman "eating up" lights. It incorporates Schemer under the yellow character, 5 lightboards, and a switch at right. By Osamu.

Figure 6.14: A "bristlebot" robot that skitters across a surface when light is shone on it. It consists of a learning sensor, pager motor, and battery, all placed atop a toothbrush head. By Osamu.

The project in Figure 6.15 features a delicate set of wings adorned with Schemer (in the middle) and 10 lightboards to provide dynamic decorative patterns. The patterns can also be set to change depending on the amount of light that falls on them.

The mask of Figure 6.16 lights up in response to the wearer's voice. It employs a near-range microphone to pick up nearby sounds, and like a voice-level meter, turns on the lights depending

Figure 6.15: A picture of "Fairy Wings" - a set of wings decorated with Schemer (in the middle) and lightboards (at the edges). By Angela.

Figure 6.16: A mask bedecked with lights that turn on in response to the wearer's voice. By Osamu.

on how loud she declares "trick or treat!" In a pinch, the mask can be reprogrammed by whistling (or saying) the right phrases, though this facility was not explored in this project.

6.2.4 Expanding the ecosystem

We are also seeing the beginnings of a wider movement of democratized design, where others use our prototypes to create building blocks for new types of ambient programming artifacts. For instance, armed with the Schemer bus communication protocol, one can program a more general purpose computing platform (such as the Arduino), to interface with (say) lightboards. In other cases, one can integrate Schemer with new output devices like electroluminescent wire (EL-wire) or electronic ink (e-ink); with new input devices like pressure, humidity, and proximity sensors that are not provided with the prototype systems; and with other communication channels like WiFi or SMS by providing an appropriate "interface layer."

Figure 6.17: Picture of a bra that flashes patterns in time to the wearer's heartbeat. The incorporated lightboards and Schemer communicate with a commercial heart rate monitor, worn as a chest strap. By Chung-Hay.

One example of building upon our prototypes in ways we did not anticipate is a bra, shown in Figure 6.17, that flashes patterns of light in time to the wearer's heartbeat: the faster it beats, the faster the patterns. The components are connected to a commercially-available heart rate monitor, and the designers used an Arduino board to transduce information from the monitor to sensor input for Schemer. A similar project, worn as a choker around the neck, uses a pulse oximeter to measure the amount of oxygen in the wearer's blood. This information is then displayed as a

series of lights: the higher the percentage of oxygen (which normaly ranges from 85% to 100%), the more lights appear on the choker. In this case too, the signals from the oximeter are converted to Schemer messages using an Arduino board.

In another example, the designers of the "NorthPaw"[129] are investigating how to simplify the wiring of their electronics parts with Schemer-compatible components. Briefly, the NorthPaw is an anklet or belt (and in a more recent incarnation, a dress[123]), that tells the wearer which way is North. The anklet contains an electronic compass and eight cellphone vibrator motors, distributed spatially in 8 compass directions. Only one motor is active at any given time, and the motor closest to North vibrates continuously. Interestingly, some wearers claim that after a few weeks of constantly wearing the anklet, one stops noticing the vibrations and instead gains an intuitive sense of which way is North. As currently implemented, each motor needs its own wire, resulting in 8 separate motor wires (plus an additional wire for common ground). To that end, the designers are collaborating with the author to create a new class of output devices (in this case motors; Figure 6.18 shows an early prototype), as well as input devices (Schemer-compatible compass), that would greatly simplify the wiring of their product. Equally interesting is the possibility of the end-user being able to customize the settings for their anklet by pointing it at a screen or passing barcodes in front of it, although this has not yet been seriously considered.

In a third example (an upcoming project), another designer is creating a wireless version of the Schemer bus, so he can collect sensor data remotely and have it affect running programs on a computationally-enhanced dress covered in hundreds of lights. The interface is a wireless sensor-network radio, and the connection to it is mediated by a more general-purpose embedded computer. This same idea will be extended to giving members of a fashion show audience wireless versions of the physical slider card, so they can change parameters and programs running on the dress remotely.

Figure 6.18: Repurposing a lightboard to run a motor. The original LED has been removed, with this tiny vibrating motor soldered onto the board in its place.

6.3 Future directions

The prototype systems are still very recent, and rough in many respects. There is thus ample work to be done in the very near term in improving their implementation. For instance, with Code Road, predicting the extent of the car's turn at any given point is unreliable, which in practice makes it difficult to create a "floor program" that can be reliably read. There are also a number of issues with with Schemer that need to be resolved: it has a relatively limited vocabulary which prevents users from writing more expressive programs; it does not communicate with programs running on other Schemers; and during programming, if the Web-browser happens to be performing some other activity (say playing a video), the timings for downloading programs change significantly, preventing programs from being reliably transmitted.

Another pressing need for improvement concerns objects' self-disclosure; as implemented, there is no way for these prototype systems to reveal their programs to users. In other words, given an artifact, one cannot read in the text of its running program, either for informative or debugging purposes. One can only make assumptions about the current program in the device based on the text on the screen, barcodes laid out floor, of from its behavior. However, since programs could have

been gathered from many places – from different "nuggets" – the aggregate program in the device remains largely unverifiable. One solution may come by way of the video camera that is becoming ubiquitous on desktop and mobile computers these days. One can imagine going up to one such computer and holding up the ambient artifact to the camera. The program is then "flashed" to the camera, and those patterns can then be interpreted back into program text. Another option might be to create a new display artifact which, with the appropriate sensing devices, could receive programs in various forms (flashing lights, barcodes, whistles) and display the text they represent. One can then imagine going up to (say) a wall and holding up their computational artifact in order to see the system's program.

Other avenues that would be useful to explore include encoding programs in animated videos, or having the systems "draw out" a picture representation of an internal program. In other words, given a program, one can have a system create on screen or paper, a symbolic representation of that program that can then be read in by it or other devices. The device could also "paint" a picture containing a program (or embed programs into existing graphics), or produce a series of notes that one can then whistle or use to tune a series of railings, per the example in Section 5.1.2.

It might also be useful to embed a camera into something like Schemer, endowing it with the ability to react to the shape and form of objects in the real world so it might be able to recognize (say) numeric glyphs. The availability of increasingly powerful computation and imaging algorithms in tiny packages suggest that embedded vision may not be too far off in the future.

Such imagined improvements are not particularly technologically ambitious or profound in and of themselves: but together they can vastly increase the power and expressive range of things people can do with ambient programs.

Looking to the longer term, we view these systems as the first steps of a larger exploration into potential venues for ambient programming more generally. We are interested in the possibility of creating accessible readable symbolic programs on a variety of surfaces (walls, floors, notebook pages), and combining those programs with an equally broad variety of devices capable of reading and responding to those programs. The prototype systems demonstrate the feasibility of ambient

programming as a style of program construction, but at the same time represent a decidedly early step in what promises to be a much vaster landscape of ambient programming systems and design.

The prototype systems give us a tremendous opportunity to rethink the desktop-centric notions of programming. They lend themselves well to creatively embedding programming instructions in everyday items and making them available within the physical context of use. Going back to the program embedded in a painting in Section 5.1.2, one can imagine that by sliding a pendant in front of one such painting, one could read in a program to dynamically change the pendant's colors. A museum curator could make visitors' light-up shirts behave differently depending on the exhibit they are in front of. It is our feeling that ambiently-programmable items (like Schemer) are a good - though tiny - step in enabling a more pedestrian approach to programming. The word pedestrian is used, not to mean boring or unimaginative, but to mean commonplace, casual, straightforward and in the same physical context as the passer-by.

Before we close, we'll make one short aside about the need for a better aesthetic for electronics, as pioneered by Leah Buechley[9]. We too find resonance with the position that current electronics (at their core) remain utilitarian and have been designed with little regard to aesthetics: (i.e black box chips, angular green copper clad boards, plastic battery holders, etc). As electronics continues to migrate from the desktop into clothing and other everyday, wearable situations, now is as good a time as any to seriously think about how electronics would better fit couture. This means (for instance) manufacturers should consider how to make LEDs in attractive sequin shapes, or microcontrollers that look like colorful buttons, or switches that look like zippers, or battery holders that look like appliqué, and so on. We hope that the next generation of prototype systems we create will share some of these characteristics, so they look less "electronic" and more like "craft".

We are continuing active development of the existing systems as well as creating new ones. In the longer term, the objects themselves are likely to evolve and will suggest still newer opportunities for ambient programmability. For instance, a wider range of Mimeolights that respond with (e.g.) musical tones or mechanical behavior rather than only with LEDs, should be possible; likewise,

we imagine versions of the Pendantif geared toward "communities of jewels", in which (e.g.) a program can be communicated from necklace to necklace; our immediate term plan is to expand the rudimentary programming interface, and then to implement means for "sending" color values over the Internet to distant partners.

Clearly, there is much still to do; but we believe that these initial steps reflect a possibility for a much broader change in end-user programming. By devising scenarios for programming objects in one's environments, garments, jewelry, toys, etc., we can move programming away from its traditional association with the desktop. Rather, we can re-imagine the nature of programming as an activity woven, in small, playful, and helpful ways, into our day-to-day lives.

Bibliography

[1] H. Abelson and G.J. Sussman. Structure and Interpretation of Computer Programs. MIT Press, Cambridge, MA, 1996.

[2] G.D. Abowd and E.D. Mynatt. Charting past, present, and future research in ubiquitous computing. In ACM Transactions on Computer-Human Interaction, 7, pages 29–58, 2000.

[3] Arduino. Arduino: electronics prototyping platform, http://www.arduino.cc.

[4] S. Bannasch. Educational Innovations in Portable Technologies. Kluwer, New York, 2001.

[5] Botanicalls. Botanicalls Kits, http://www.botanicalls.com, 2009.

[6] V. Braitenberg. Experiments in Synthetic Psychology. MIT Press, Cambridge, MA, 1984.

[7] L. Buechley. Quilt snaps: A fabric based computational construction kit. In WMTE '05: Proceedings of the IEEE International Workshop on Wireless and Mobile Technologies in Education, pages 219–221, Washington, DC, USA, 2005. IEEE Computer Society.

[8] L. Buechley and M. Eisenberg. Boda blocks: a collaborative tool for exploring tangible three-dimensional cellular automata. In CSCL'07: Proceedings of the 8th iternational conference on Computer supported collaborative learning, pages 102–104. International Society of the Learning Sciences, 2007.

[9] L. Buechley, M. Eisenberg, J. Catchen, and A. Crockett. The lilypad arduino: Using computational textiles to investigate engagement, aesthetics, and diversity in computer science education. In SIGCHI conference on Human factors in computing systems (CHI), 2008.

[10] L. Buechley, N. Elumeze, and M. Eisenberg. Electronic/computational textiles and children's crafts. In Interactive Design and Children (IDC), 2006.

[11] Eva Burneleit, Fabian Hemmert, and Reto Wettach. Living interfaces: the impatient toaster. In Proceedings of the 3rd International Conference on Tangible and Embedded Interaction, TEI '09, pages 21–22, New York, NY, USA, 2009. ACM.

[12] L. Cardelli. A language with distributed scope. In 22nd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Language, pages 286–297, 1995.

[13] A. Cockburn and A. Bryant. Do it this way: equal opportunity programming for kids. CHI, pages 246–251, 24-27 Nov 1996.

[14] Scheme community. Revised^6 report on the algorithmic language scheme, 2007.

[15] Playful Invention Company. Picocrickets, http://www.picocricket.com.

[16] G. Cross. Kid's Stuff. Cambridge MA: Harvard University Press, 1997.

[17] A. Cypher, editor. Watch What I Do: Programming by Demonstration. Cambridge MA: MIT Press, 1993.

[18] Timex DataLink. http://www.timex.com.

[19] M. Dertouzos. The user interface is the language. Jones and Bartlett Publishers Inc., Boston, MA, 1992.

[20] Ambient Devices. Ambient orb, http://www.ambientdevices.com/cat/orb/orborder.html, 2008.

[21] Ambient Devices. Ambient umbrella, http://www.ambientdevices.com/products/umbrella.html, 2009.

[22] P.H. Dietz. Very low cost sensing and communication using bidirectional leds. In International Conference on Ubiquitous Computing (UbiComp), pages 29–58, 2003.

[23] C. DiGiano and M. Eisenberg. Self-disclosing design tools: A gentle introduction to end-user programming. In Proceedings of Designing Interactive Systems (DIS '95), pages 189–197, 1995.

[24] A. diSessa. Changing Minds: Computers, Learning, and Literacy. Cambridge MA: MIT Press, 2000.

[25] A. Druin. Noobie: the animal design playstation. SIGCHI Bull., 20(1):45–53, 1988.

[26] D. Dube. Bit: A very compact scheme system for embedded applications, http://citeseer.ist.psu.edu/dube00bit.html.

[27] A. Dunne and F. Raby. Fields and thresholds, presentation at the doors of perception, 1994.

[28] M. Eisenberg. Programming in Scheme. The Scientific Press, Redwood City, CA, 1988.

[29] M. Eisenberg. Programmable applications: Interpreter meets interface. Technical report, MIT, 1989.

[30] M. Eisenberg, N. Elumeze, and T. Wrensch. Computationally-enhanced craft items: Prototypes and principles, 2004.

[31] N. Elumeze. Smarttiles: Towards room-sized, child-programmable computational artifacts. Master's thesis, University of Colorado at Boulder, 2007.

[32] N. Elumeze and M. Eisenberg. Smarttiles: Designing interactive "room-sized" artifacts for educational computing. Children, Youth and Environments, 15(1):54–66, 2005.

[33] N. Elumeze and M. Eisenberg. Towards ambient programming for children. In Cognition and Exploratory Learning in Digital Age (IADIS CELDA 2006), 2006.

[34] N. Elumeze and M. Eisenberg. Buttonschemer: Ambient program reader. In Proceedings of Mobile HCI 2008), 2008.

[35] B. Fry et. al. Processing, http://www.processing.org.

[36] C. Rader et al. Degrees of comprehension: children's understanding of a visual programming environment. In CHI, pages 351–358, New York, NY, USA, 1997. ACM.

[37] C. Wisneski et. al. Ambient displays: Turning architectual space into an interface between people and digital information. In Proc. International Workshop on Cooperative Buildings, A Language with Distributed Scope., pages 22–32, 1998.

[38] D. W. Hogg et al. Braitenberg creatures. MIT Media Lab, 13, 1991.

[39] F. Martin et al. Metacricket: A designer's kit for making computational devices. IBM System Journal,, 2000.

[40] F. Scharf et al. Tangicons: Algorithmic reasoning in a collaborative game for children in kindergarten and first class. In IDC '08: Proceedings of the 8th International Conference on Interaction Design and Children, pages 186–189, New York, NY, USA, 2008. ACM.

[41] G. Fischer et al. Beyond object-oriented technology: Where current approaches fall short. Human-Computer Interaction, pages 79–119, 1995.

[42] H. Raffle et al. Topobo: a constructive assembly system with kinetic memory. In CHI 2004, pages 647–654, 2004.

[43] H. Raffle et al. Beyond record and play. In In Proceedings of CHI 2006, pages 681–690, 2006.

[44] J. Malonet et. al. Scratch: A sneak preview. In Int'l Conference on Creating, Connecting, and Collaborating through Computing, Kyoto, Japan, pages 104–109, 2004.

[45] J. Patten et. al. Audiopad: A tag-based interface for musical performance. In Proc. Conference on New Interface for Musical Expression, 2002.

[46] K. Ryokai et al. Io brush: Drawing with everyday objects as ink. In Proc. CHI, 2002.

[47] K. Van Laerhoven et. al. Pin&play: Networking objects through pins.

[48] M. Conway et al. Alice: Lessons learned from building a 3d system for novices. In SIGCHI conference on Human factors in computing systems (CHI), 2000.

[49] M. Eisenberg et. al. Invisibility considered harmful: Revisiting traditional principles of ubiquitous computing in the context of education. In Proceedings of IEEE International Workshop on Wireless, Mobile and Ubiquitous Technologies in Education (WMUTE), 2006.

[50] M. Felleisen et. al. The structure and interpretation of the computer science curriculum. J. Funct. Program., 14(4):365–378, 2004.

[51] M. Resnick et al. Programmable bricks: Toys to think with. In IBM Systems Journal, volume 35(3), pages 443–452, 1996.

[52] M. Resnick et. al. Digital manipulatives: New toys to think with. In Proc. CHI, 1998.

[53] M. Resnick et al. Scratch: Programming for Everyone, http://scratch.mit.edu/pages/research. MIT Media Lab., 2007.

[54] M. Yim et al. Polybot: a modular reconfigurable robot. In IEEE Intl. Conference on Robotics and Automation, 2000.

[55] P. Frei et al. curlybot: Designing a new class of computational toys. In CHI Letters, 2, pages 129–136, 2000.

[56] R. Tieben et al. actdresses: interacting with robotic devices - fashion and comics. In HRI '10: Proceeding of the 5th ACM/IEEE international conference on Human-robot interaction, pages 353–354, New York, NY, USA, 2010. ACM.

[57] R. Watanabe et. al. The soul of activecube: implementing a flexible, multimodal, three-dimensional spatial tangible interface. Comput. Entertain., 2(4):15–15, 2004.

[58] S. Card et. al. A morphological analysis of the design space of input devices. In ACM Transactions on Information Systems,, number 2 in 9, pages 99–122, 1991.

[59] S. Hosomi et al. A system for controlling led blink in wearable fashion. In International Wireless Communications and Mobile Computing Conference (IWCMC), pages 665–670, 2007.

[60] S. Seitinger et al. Urban pixels: painting the city with light. In CHI '09: Proceedings of the 27th international conference on Human factors in computing systems, pages 839–848, New York, NY, USA, 2009. ACM.

[61] T. Wrensch et al. The rototack: Designing a computationally enhanced craft item. In Proceedings of DARE 2000, Designing Augmented Reality Environments, pages 93–102, 2000.

[62] A. Faaborg and H. Lieberman. A goal-oriented browser. In ACM Conference on Computer-Human Interaction (CHI-06), Montreal, pages 647–654, 2006.

[63] M. Feely. Picbit: A scheme system for the pic microcontroller.

[64] T. Flowers, C.A. Carver, and J. Jackson. Empowering students and building confidence in novice programmers through gauntlet. In Frontiers in Education, 2004.

[65] M. Gardner. Wheels, Life, and other Mathematical Amusements. New York: W. H. Freeman., 1985.

[66] Greyworld. "railings" - http://greyworld.org/archives/35.

[67] BRIO Group. Brio trains, http://brio.knex.com.

[68] High-Low Tech Group. Lilypond website: http://lilypond.media.mit.edu.

[69] D. Gudeman. Representing type information in dynamically-typed languages. Technical Report TR93-27, Tucson, Arizona, 1993.

[70] G. Hatano and K. Inagaki. Two Courses of Expertise. Child Development and Education, pages 262–272. Freeman, San Francisco, 1986.

[71] J. Healy. Failure to Connect. New York: Simon and Schuster, 1998.

[72] C. A. R. Hoare. Hints on programming language design. Technical report, Stanford University, 1973.

[73] M. Horn and R.J.K. Jacob. Designing tangible programming languages for classroom use. In TEI '07: Proceedings of the 1st international conference on Tangible and embedded interaction, pages 147–150, New York, NY, USA, 2007. ACM.

[74] J. Heiner S. Hudson and K. Tanaka. The information percolator: Ambient information display in a decorative object. In Proc. UIST '99, pages 141–148, 1999.

[75] AgentSheets Inc. Agentsheets, http://www.agentsheets.com.

[76] MicroEngineering Labs Inc. Picbasicpro, http://www.melabs.com.

[77] Stagecast Inc. Stagecast creator, http://www.stagecast.com/creator.html.

[78] Adafruit Industries. Adafruit wave shield for arduino, http://adafruit.com.

[79] Chumby Industries. Chumby, http://www.chumby.com, 2009.

[80] Educational Innovations. Frightened grasshopper, http://www.teachersource.com/ultraviolet/solarenergy/robotikits.aspx.

[81] Instructables. Instructables website: http://instructables.com.

[82] iRobot. Roomba home cleaning robot, http://www.irobot.com, 2010.

[83] H. Ishii and B. Ullmer. Tangible bits: Towards seamless interfaces between people, bits and atoms. In CHI, pages 234–241, New York, NY, 1997. ACM Press.

[84] S. Johansson. Sniff: designing characterful interaction in a tangible toy. In IDC '09: Proceedings of the 8th International Conference on Interaction Design and Children, pages 186–189, New York, NY, USA, 2009. ACM.

[85] K. Kahn. Toontalk, http://www.toontalk.com.

[86] K. Kahn. Drawings on napkins, video-game animation, and other ways to program computers. In Communications of the ACM, number 8 in 39, pages 49–59, 1996.

[87] C. Kelleher and R. Pausch. Using storytelling to motivate programming. Commun. ACM, 50(7):58–64, 2007.

[88] H. Kuzuoka and S. Greenberg. Mediating awareness and communication through digital but physical surrogates. In Proc. CHI' 99 Companion, pages 11–12, 1999.

[89] S. Lee. Putting it in context: A syntactic theory of incremental program construction http://citeseer.ist.psu.edu/lee96putting.html.

[90] Lego. Lego mindstorms, http://mindstorms.lego.com.

[91] M. Leung. Weight-sensing tote, http://mtifall09.wordpress.com/author/markaleung.

[92] C. Lewis. Nopumpg: Creating interactive graphics with spreadsheet machiner. technical report cu-cs-372-87, department of computer science, university of colorado at boulder, 1987.

[93] H. Lieberman, editor. Your Wish is My Command: Programming by Example. Morgan Kaufmann, San Francisco, 2001.

[94] F. Martin. Ideal and real systems: A study of notions of control in undergraduates who design robots. In Y. Kafai and M. Resnick (eds.), Constructionism in Practice. Lawrence Erlbaum, Hillsdale, NJ, 1992.

[95] F.G. Martin. Handyboard, http://handyboard.com.

[96] Fred Martin. Crickets, Tiny Computers for Big Ideas, http://fredm.www.media.mit.edu/people/fredm/projects/cricket. MIT Media Lab., 1998.

[97] D. L. Maulsby and I. H. Witten. Inducing programs in a direct-manipulation environment. SIGCHI Bull., 20(SI):57–62, 1989.

[98] T. McNerney. Tangible programming bricks: An approach to making programming accessible to everyone. Master's thesis, MAS, MIT, 2000.

[99] J. Montemayor. Physical programming: tools for kindergarten children to author physical interactive environments. PhD thesis, University of Maryland., 2003.

[100] B. et al. Myers. End-user programming (invited research overview). In In Proceedings of CHI 2006, pages 75–80, 2006.

[101] B. Nardi. A Small Matter of Programming. Cambridge MA: MIT Press, 1993.

[102] NeuroSmith. Music blocks, http://shop.smallworldtoys.com.

[103] F. Nielson and H.R. Nielson. Interprocedural control flow analysis. In European Symposium on Programming, pages 20–39, 1999.

[104] Nike. Nike+, http://www.nikeplus.com, 2010.

[105] Fischell Department of Bioengineering. ARMPIT Scheme, http://armpit.sourceforge.net. University of Maryland at College Park., 2010.

[106] C. O'Malley and D. S. Fraser. Literature review in learning with tangible technologies. nesta futurelab report 12, 2005.

[107] T. Oppenheimer. The Flickering Mind. New York: Random House, 2003.

[108] A. Gomez Ortigoza. Kokoro, http://anaiid.com/kokoro, 2009.

[109] S. Papert. Mindstorms. Basic Books, New York, 1980.

[110] S. Papert. Situating Constructionism. in Harel, I. and Papert, S. eds. Ablex Publishing Company, Norwood, NJ, Hillsdale, NJ, 1991.

[111] Parallax. Basic Stamphttp://www.parallax.com, 2008.

[112] K. Pepper and Y. Kafai. Creative coding: The role of art and programming in the k-12 educational context.

[113] R. Perlman. Using computer technology to provide a creative learning environment for preschool children. Master's thesis, AI, MIT, 1976.

[114] J. Piaget. The language and thought of the child. Academic Press, London, 1959.

[115] Polar. Polar chest strap, http://www.polar.fi/us-en, 2010.

[116] W. Poundstone. The Recursive Universe. New York: William Morrow, 1985.

[117] R. Poynor. The Hand That Rocks the Cradle. ID Magazine, 1995.

[118] J. Qi and L. Buechley. Electronic popables: exploring paper-based computing through an interactive pop-up book. In Proceedings of the fourth international conference on Tangible, embedded, and embodied interaction, TEI '10, pages 121–128, New York, NY, USA, 2010. ACM.

[119] M. Resnick. Turtles, Termites, and Traffic Jams. Cambridge MA: MIT Press, 1994.

[120] B. Rogoff. Apprenticeship in thinking. Oxford University Press, New York, 1990.

[121] B. Rogoff. Children's guided participation and participatory appropriation in sociocultural activity, pages 121–153. Erlbaum, Hillsdale, NJ, 1993.

[122] B. Rogoff. The Cultural Nature of Human Development, pages 236–281. Oxford University Press, New York, 2003.

[123] M. Scheff. The North Skirt, http://steampunkworkshop.com/north-skirt, 2010.

[124] E. Schweikardt and M. D. Gross. roblocks: A robotic construction kit for mathematics and science education. In International Conference on Multimodal Interaction, 2006.

[125] E. Schweikardt and M. D. Gross. A brief survey of distributed computational toys. In he First IEEE International Workshop on Digital Game and Intelligent Toy Enhanced Learning, DIGITEL 2007, pages 57–64, 2007.

[126] E. Schweikardt and M. D. Gross. The robot is the program: Interacting with roblocks. In TEI 2008: Second International Conference on Tangible and Embedded Interaction, 2008.

[127] Evil Mad Scientist. Evil mad scientist laboratories, http://www.evilmadscientist.com.

[128] M. Sekiguchi. "ubwall", ubiquitous wall changes an ordinary wall into the smart ambience. In sOc-EUSAI '05: Proceedings of the 2005 joint conference on Smart objects and ambient intelligence, pages 47–50, New York, NY, USA, 2005. ACM.

[129] Sensebridge. Northpaw: Haptic compass anklet, 2010.

[130] B. Shneiderman. Direct manipulation: A step beyond programming languages. IEEE Computer, 16(8):57–69, 1983.

[131] J. Silver. What would be one-derful? http://web.media.mit.edu/ silver.

[132] A. Smith. Using magnets in physical blocks that behave as programming objects. In TEI '07: Proceedings of the 1st international conference on Tangible and embedded interaction, pages 147–150, New York, NY, USA, 2007. ACM.

[133] A Smith. Handcrafted physical syntax elements for illiterate children: initial concepts. In IDC '08: Proceedings of the 8th International Conference on Interaction Design and Children, pages 186–189, New York, NY, USA, 2008. ACM.

[134] B. Sutton-Smith. Toys as Culture. NY: Gardner Press, 1986.

[135] H. Suzuki and H. Kato. Algoblock: A tangible programming language, a tool for collaborative learning. In Proc. Fourth European Logo Conference, pages 297–303, 1993.

[136] ThingM. Blinkm, http://thingm.com/products/blinkm.

[137] R. Tilley. "bamboo forest, kyoto, japan" http://www.art.com.

[138] J. Tudge. Processes and consequences of peer collaboration: A vygotskian analysis. Child Development, 63:1364–1379, 1992.

[139] Violet. Nabaztag, http://www.nabaztag.com.

[140] L. S. Vygotsky. Mind in society: The development of higher psychological processes. Harvard University Press, Cambridge, MA, 1978.

[141] M. Weiser. The computer for the 21st century, 1996.

[142] M. Weiser. Some computer science issues in ubiquitous computing. In Communications of the ACM, number 75–84 in 36, AUG 1998.

[143] M. Weiser and J. Brown. Designing calm technology, 1996.

[144] K. N. Whitley. Visual programming languages and the empirical evidence for and against. Journal of Visual Languages and Computing, pages 109–142, 1997.

[145] J. M. Wing. Viewpoint: Computational thinking. Communications of the ACM, 43(3):33–35, 2006.

[146] T. Wrensch. Programming Computationally Enhanced Craft Items. PhD thesis, University of Colorado at Boulder, 2002.

[147] T. Wrensch and M. Eisenberg. The programmable hinge: computationally enhanced crafts. In In Proceedings of UIST '98, pages 89–96, 1998.

[148] P. Wyeth. How young children learn to program with sensor, action, and logic blocks. Journal of the Learning Sciences, 17(4):517–550, 2008.

[149] P. Wyeth and G. Wyeth. Electronic blocks: Tangible programming elements for preschoolers. In In Proceedings of the Eighth IFIP TC13 Conference on Human-Computer Interaction (Interact 2001), 2001.

[150] T. Yuasa. Lisp on lego mindstorms. In International Lisp Conference 2003, 2003.

[151] R. J. Yun. Raymatic: http://mtifall09.wordpress.com/2009/12/15/raymatic-an-ambient-interactive-picture-frame.