Moveable Objects, Mobile Code

Kwindla Hultman Kramer

Submitted to the Program in Media Arts and Sciences, School of Architecture and Planning, in partial fulfillment of the requirements for the degree of Master of Science in Media Arts and Sciences at the Massachusetts Institute of Technology

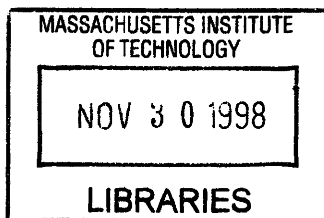
September 1998

© Massachusetts Institute of Technology, 1998. All Rights Reserved.

Author____
Program in Media Arts and Sciences
August 7, 1998

Certified by____
Mitchel Resnick
Associate Professor of Media Arts and Sciences
LEGO Papert Career Development Professor of Learning Research
Massachusetts Institute of Technology

Accepted by____
Stephen A. Benton
Chair, Departmental Committee on Graduate Students
Program in Media Arts and Sciences



Moveable Objects, Mobile Code

By

Kwindla Hultman Kramer

Submitted to the Program in Media Arts and Sciences,
School of Architecture and Planning,
on August 7, 1998
in partial fulfillment of the requirements for the degree of
Master of Science in Media Arts and Sciences
at the Massachusetts Institute of Technology

ABSTRACT

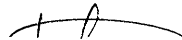
This thesis describes the design of a "toy system" of small, communicating computers. These computers, called Tiles, are a child-scale platform for exploring issues related to networks, communication and computational process. The system is designed to be engaging and easy to use, while at the same time providing opportunity to reflect upon a rich set of ideas and offering users a new set of cognitive tools. For example, Tiles are programmed using a *mobile code* approach, meaning that programs are written to move across the network from one Tile to the next. The conceptual clarity of the mobile-code metaphor allows even novice users to write interesting programs, while the depth of that metaphor as a subtle way to represent large-scale behavior across a network creates an opportunity for extended educational play.

Thesis Advisor: Mitchel Resnick
Associate Professor of Media Arts and Sciences
LEGO Papert Career Development Professor of Learning Research

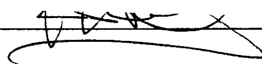
This research was sponsored by the LEGO Group, by Motorola, Inc., and by the Things That Think and Toys of Tomorrow Consortia of the MIT Media Lab.

Moveable Objects, Mobile Code

Thesis Committee



Thesis Advisor
Mitchel Resnick
Associate Professor of Media Arts and Sciences
LEGO Papert Career Development Professor of Learning Research
Massachusetts Institute of Technology



Thesis Reader
Pattie Maes
Associate Professor of Media Arts and Sciences
Massachusetts Institute of Technology

Thesis Reader
Erik Hansen
Director
LEGO Futura Boston Branch

TABLE OF CONTENTS

INTRODUCTION 9

1 BACKGROUND AND MOTIVATION 15

1.1 TOOLS TO THINK WITH 15
1.1.1 THREE KINDS OF TOOLS 18
1.1.1.1 Artifact tools 18
1.1.1.2 Cognitive tools 19
1.1.1.3 Substrate tools 19
1.1.2 THE BENEFITS OF META 20
1.1.2.1 Learning to Solve Problems: Procedural Decomposition 22
1.1.2.2 Another Analytic Tool: Emergence 24
1.1.3 THE TILES PROJECT 25
1.1.3.1 Substrate Choices 25
1.1.3.2 Cognitive Abstractions 28
1.1.3.3 Artifact Tools 30
1.2 MOBILE PROGRAMS: TINY COMPUTERS AND BIG NETWORKS 30

2 TILES – DESIGN AND IMPLEMENTATION 37

2.1 HARDWARE 37
2.1.1 MICROCONTROLLER AND MEMORY 37
2.1.2 NETWORK LINKS 39

2.1.3 USER INPUT-OUTPUT AND EXPANDABILITY 41
2.1.4 BATTERY POWER 41
2.2 SOFTWARE 42
2.2.1 A MINIMAL KERNEL 43
2.2.1.1 Infrared Communications 43
2.2.1.2 Memory Management and Task Switching 45
2.2.2 MOBILE PROGRAMS 47

3 FUTURE WORK 49

3.1 A SECOND MOBILE CODE IMPLEMENTATION 49
3.1.1 ABSTRACTION AND COMPACTNESS 49
3.1.2 SAFETY AND AVOIDING BUGS 50
3.1.3 LEVERAGING COMMON TOOLS AND PRIOR KNOWLEDGE 51
3.2 THE DESKTOP SIDE OF THINGS 51
3.2.1 PROGRAMMING REPRESENTATIONS 52
3.3 NEW HARDWARE 55

4 CONCLUSION 56

APPENDIX A: SCHEMATIC OF TILE CIRCUIT BOARD 58

REFERENCES 59

Introduction

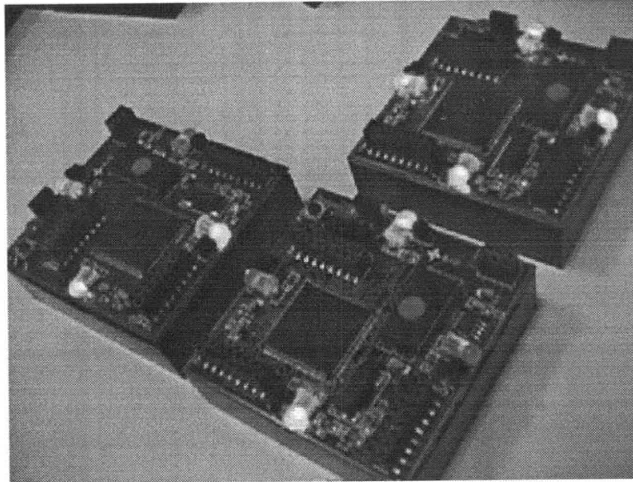
Computer networks have become, in recent years, indispensable fixtures of modern life. Digital communications technologies exert an ever-increasing impact on the way we work, play, learn, consume, converse and coexist. It has been argued that the rapidity of social change accompanying the growth and spread of new computer networks is unparalleled in recorded history[1]; whether or not this is true, we are certainly deploying technology that we have very little experience with and relatively little understanding of. We are learning as we go.

We have few direct precedents with which to compare our new information infrastructure. Massiveness of scope is juxtaposed with intimacy of effect, intricate technical complexity accompanies extreme decentralization. We are in the process of developing a new set of understandings about ubiquitous, large-scale networks and their effects on our world. To do this, we will need new metaphors that describe the mechanisms of these new technologies, and new intuitions about the ways in which they impact our lives.

This thesis is a description of a toy that is intended to provide a modest framework for exploring some of the characteristics of computational networks. The toy that I have designed is a construction kit of sorts, made up of a collection of small, square, block-like computers called Tiles. A set of Tiles allows one to build a little network of computers, and to arrange, rearrange and experiment with that network. The Tiles are a toolkit for exploring computational processes and the behaviors of collections of computers, just as a box of watercolors provides materials for

[1] See, for example, Cairncross. F. *The Death of Distance* (1997).

investigating color and pattern through painting, or a bag of Scrabble letters the opportunity to learn about language and words by via creative spelling.



three Tiles

The Tiles system is designed to be intuitive, engaging and accessible to novice users, while still providing a rich and flexible intellectual terrain for more experienced children (graduate students, for example) to explore. Each Tile is an independent, physical, touchable object that can be picked up and moved around, and the interaction between Tiles is consistent and straightforward: all Tiles can communicate with their four adjacent neighbors.

The goal of producing a comprehensible system – a "toy network" in the scientist's as much as the child's sense of the term – served as major organizing principle throughout the development of the Tiles. This emphasis on simplicity resulted in a network architecture characterized by extreme homogeneity and enforced local interactions. Each Tile is identical, and can run any piece of code that is passed to it by a neighbor. And, because the Tiles can communicate only with their "next-door neighbors," there is an explicit and consistent topological framework underpinning all network communications.

These efforts to build a comprehensible framework have not decreased the capacity of the Tiles system to produce subtle, counter-intuitive, or surprising behaviors. Far from it! One learns

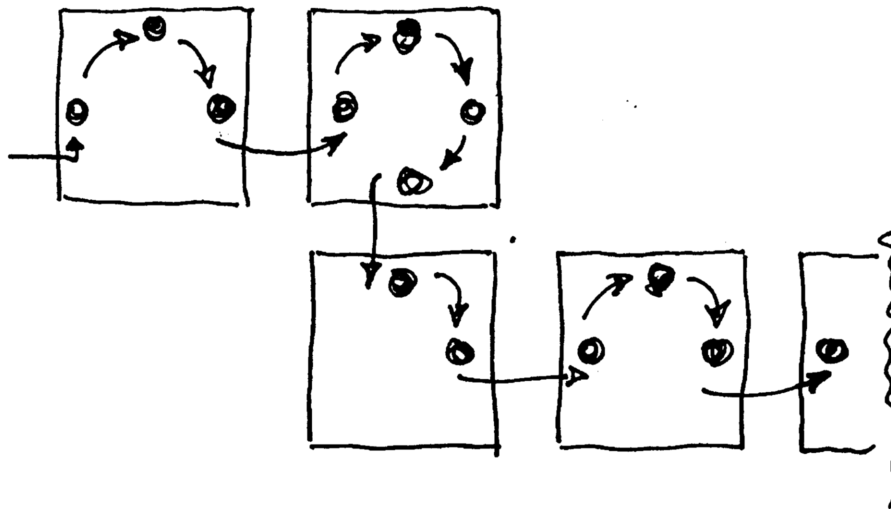
very quickly, playing with only a few programs on a few connected Tiles, that the landscape of possible interactions is dauntingly rich. In order to capture some of this richness, and to allow children to experiment with creating and dissecting complex interactions, the world of the Tiles comes complete with its own programming metaphor. Every program for the Tiles platform is a piece of *mobile code* with the potential to move across the network simply, easily and non-destructively. I will argue in this thesis that the mobile code approach is a useful way of thinking about the functionality that networks can provide, and serves as a cognitive tool that helps us understand and specify collective activity across multiple machines. Using this approach, complex interactions involving multiple Tiles and multiple programs can be designed and built from many small, comprehensible pieces of mobile code. A programming language that instantiates this metaphor can function as a powerful, illuminating tool for representing computation.

Theoretical assumptions, implementation details, and practical lessons will all be discussed in detail, below. Before moving on the body of this thesis, however, I would like to describe a scenario that communicates something of the flavor of the Tiles as they are intended to be used.

The Tiles made their public debut at a Media Lab conference on "Toys of Tomorrow." Each conference attendee was to be given a Tile, and we needed to develop an activity that encouraged collaborative play. These early incarnations of the Tiles were of relatively limited input and output capacities – each had four bicolor LEDs (one in the center of each side) and a single pushbutton – so the activity needed to be carefully-tailored to work well with only moderate visual feedback.

After prototyping several possible games and puzzles, we settled on a relatively simple activity in which there are two varieties of mobile program that inhabit the Tiles, red-lights and green-lights. Pressing the pushbutton on a Tile creates a program and starts it running, with the color pre-assigned and indicated by a sticker, so some Tiles could make red-lights and some

could make green-lights. The red-light programs hop around the four edges of the Tile in a clockwise direction, moving to an adjacent Tile whenever one is present. The green-light programs are exactly the same (but green, of course), except that they hop around counter-clockwise. Finally, programs can not hop on top of or over one another, so the programs will become "stuck" when there is no place to go, like cars in a traffic jam.



the path that a red-light program takes as it traverses several Tiles

This activity is simple enough that one can figure out the rules, and some basic patterns of behavior, with a little focused experimentation. There is enough complexity of interaction, however, that with several Tiles and several programs the behavior of the system rapidly becomes difficult to follow, and even in very simple configurations the paths the programs take can be quite surprising.

We noticed, while experimenting with the activity in preparation for the conference, that two very different styles of exploration manifested themselves as people tried to make sense of the Tiles. Some people carefully experimented with minimalist combinations of Tiles and programs, attempting to specify concrete rules of behavior. Others constructed sprawling

networks of Tiles and injected as many programs into the system as they could, then watched the resulting large-scale patterns of light and movement.

I was extremely pleased that the red-light/green-light activity – and, more generally, the Tiles framework itself – seemed rich enough to support these contrasting play styles equally well. I made one major adjustment, after watching my colleagues play with these two programs. In order to add dynamism to the system and ensure that the macro-pattern approach was as potentially interesting as the micro-experimentation style, I added some code to the programs that caused them to change their color and direction after being "stuck" in one place for eight seconds. This made it possible to sit back and watch system-level patterns iterate without intervening to free lights that begin to pile up in "corners" and "dead ends."

A number of games that one could play with the red-light/green-light programs emerged. Because a single Tile could create only one program at a time (the pushbutton was configured to start a program if pushed when the Tile was empty, but clear the Tile of any running programs if it was not), one obvious challenge was to create as many programs as possible using only a few Tiles. It happens that with two Tiles one can create at most five programs, by filling all four lights on one Tile. This takes a little bit of dexterity, as the timing required to transfer the last of the four programs is a bit tricky. A nice surprise awaits the person who manages to fill a Tile with four programs of the same color: after eight seconds have elapsed she will suddenly have a Tile that has four programs of the opposite color! The programs have decided that they are stuck and need to flip-flop.

It is also possible to look for "oscillating patterns" that either never get stuck, or get stuck but rebound in interesting ways after the eight-second swap. With three Tiles arranged in an L-shape, it is possible to inject a single red-light and a single green-light in such a way that they bounce off each other over and over without getting lodged together in a corner. With four Tiles and two programs of each color, it is possible to form a bounce-and-scatter pattern where the two

pairs become trapped at the same time in opposite corners, then swap colors and meet in the middle to bounce off each other and form the other possible pairing in the other two corners!

But my favorite game is to put a great many Tiles together in some kind of pattern – a large rectangular arrangement with empty space in the middle, for example – inject as many programs as possible, and watch the way the programs move with and against one another. By changing the geometric arrangement of the Tiles, the pattern of movement can be made to shift radically, from spatial concentration to diffusion, from rapid traffic through a given area to the presence of only rare and often mono-colored stragglers, from identifiable structure to apparent chaos. The direct malleability, experimental flexibility, and compelling visual effect of this kind of large scale collection is exactly what I hoped, while designing the Tiles, that they would be able to achieve.

1 Background and Motivation

1.1 Tools to Think With

This thesis explores a set of ideas about networked computation, and suggests a framework for thinking about programs that exist in a network context. The goal of my thesis work has been to develop a set of tools that support children in learning about programming collections of computers, so thinking about computers as educational platforms has been both a first step and a continuing component of this research.

Computers are powerful as tools because they are very flexible machines: a computer can be programmed to play music or to simulate predator/prey biology, to render three-dimensional worlds or to graph complex data along a pair of axes. A desktop computer is really a meta-tool, a platform for a great many different, subject-specific applications.

This generality is exciting to those of us engaged in educational research. The computer seems like a revolutionary enabler of open-ended learning. Computers are sometimes seen by educators as replacements for textbooks, as a new kind of packaging for curricular materials. And computers certainly do offer some advantages in this regard. A textbook must be printed with particular pages in a particular order, for example, while a computer program has no such limitation. But the computer, as meta-tool, is really more like paper and pencil than like a pre-printed book. We can use the computer as a framework that aids the exploration – not just the acquisition – of vast varieties of knowledge. The iterativity and flexibility of computational environments suggest new interactive, immersive possibilities for classrooms and curricula.

There is a general consensus among parents, educators and policy-makers that computers are important to contemporary education. But for the most part, computers are seen as important for essentially vocational reasons; most jobs in our society now require some knowledge of

computers. While vocational concerns are real and important, the cognitive and pedagogical benefits of integrating computers into the learning environment are extremely important, as well. Computers in our classrooms can do much more than simply prepare students for specific workplace roles.

Real computer-literacy isn't just a set of particular skills, but a new way of acquiring skills in general. The computer screen is a gateway to worlds of information, worlds that students can interact with and explore using new sets of computational tools. Successful students have always learned how to break complex knowledge down into manageable pieces, to build their own understandings of difficult ideas, and to motivate themselves while doing so.[2] Rich, varied computational environments can help students who have not been so successful in traditional classrooms to develop these abilities.

Learning, in the broad sense, means not just acquiring domain-specific knowledge, but acquiring approaches to and frameworks for that knowledge. An example: high school history textbooks are filled with "time lines," graphical diagrams of series of important dates. Time lines are an attempt to provide a simplified, chronological view of the textual information presented in the books, a literal rendering of the idea that a "this happened, then that happened" view of things is important to the study of history. The time line is a particular way of representing information, and as such functions as a *tool to think with*, a way of making sense of some piece of the world, a way of crafting explanation and understanding.

Computers excel at presenting us with *tools to think with*. An on-screen, annotated, interactive time line can contain far more information than a printed version, and render that information clearly and in detail. The print author's choice between narrative text and graphical time line is not so stark in the digital world. The distinction between the two frameworks blurs as the fluidity of the screen replaces the immutability of the page.

---

[2] Simon, H.A. "The computer as a laboratory for epistemology" (1992).

But even more importantly, computers are very good at processing abstract symbols, and so provide fertile ground for the development and utilization of new representations of knowledge. The digital computer is a kind of factory for *tools to think with.* An interactive, multimedia timeline is an amplification, to use John Seely Brown's phrase,[3] of an already-existing cognitive lever, whereas today's widely-available, powerful microprocessors allow us to create tools with no direct non-computational precedents. We can build, and subsequently use to great effect, new and dynamic representations of systems, of processes, and of domain-specific knowledge.

Computer-based simulation tools spanning a wide array of disciplines are currently available, and serve as testament to the new possibilities that computational media offer. From complex analysis environments for mechanical engineers, to educational tools for biology students, to spreadsheet-based financial applications, new iterative approaches to problem-solving stand alongside (and often supplant) traditional methodologies that relied heavily on laborious calculation. For example, differential equations were, until recently, the only representation available that enabled a researcher to dissect, understand and model certain kinds of dynamic systems in a rigorous, broadly-applicable, flexible manner. Now, however, programmatic, computational representations have proven themselves valid and useful alternatives in this domain.[4]

Programmatic representations can take many forms, but useful computer "languages" all provide programmers with the raw material for building their own extensions to language, for developing their own representations of knowledge. Computers, when used to their fullest

---

[3] Brown, J. S. *Idea Amplifiers: New Kinds of Electronic Learning Environments.*
[4] For some thoughts on the lessening importance of the mechanics of calculation, and on the effects this should have on high school mathematics coursework, see: Fey, J. T. ed. *Computing and Mathematics: The Impact on Secondary School Curricula.*

potential, are tools that allow us to build new kinds of tools.[5] The ultimate goal for the educational researcher is to provide an environment substantive enough that children can devise both new instances and new kinds of tools for themselves, environments that serve to help children both master and create tools to think with.

1.1.1 Three kinds of tools

What kind of computational environment pushes children to tackle big ideas? How can a computer program make important ideas palpable, intelligible and exciting to a student? The answers depend on how we use ideas as tools, and on how our ideas relate to the tools we use. A full theoretical investigation of these subjects is beyond the scope of this thesis, but I would like to describe the categorical framework that I used in developing the Tiles system. Beginning with a very simple definition of *tool* – a tool is something that helps someone to achieve a desired result – the tools we use can be divided into three groups:

1.1.1.1 Artifact tools

This category includes things that come to mind for most of us when we think "tool." Hammers, for example, and screwdrivers are so emphatically tools that they live in tool-boxes. Kitchen knives, or bicycle pumps, or sewing machines are slightly less likely to leap to mind, but still obviously fit our conception of basic tools.

Artifact tools are human-created objects that we use to directly enable some activity. Two more examples: I am using a word processing program to create this document; I fixed a leaky pipe last night with a wrench and a pair of pliers.

---

[5] Roy Pea discusses the transformative aspects of computational technologies, arguing that computers go beyond "amplifying" the capabilities of users to effecting a "reorganization" of the intellect, in Pea, R. "Beyond Amplification: Using the Computer to Reorganize Mental Functioning" (1985).

1.1.1.2 Cognitive tools

Cognitive tools are the ideas, patterns, mental templates, techniques or processes that we use to organize our thinking about some task. We might say that they are metaphoric rather than literal tools, although I think that that implies an overly restrictive definition of tool in the first place.

Cognitive tools can be as simple as the mnemonic "righty-tighty, lefty-loosy," or as complex as historical theories about aggregation of capital and alienation.

We use artifact tools to achieve specific effects – attach there, turn this, press that. We use cognitive tools, on the other hand, to reason about cause and effect. I am typing this thesis on a word processor, but, at a higher level, I am structuring the writing process according to a hierarchical, ever-evolving outline. The outline serves to organize my thinking, allowing me to chart my argument and judge the flow of the ideas I am trying to present. I write this outline down (or type it on my computer) but the important thing is the idea of it, the mental model I have of how a paper should be structured.

1.1.1.3 Substrate tools

Substrate tools are the constraints that define a problem. Here we stretch a naïve definition of tool even further: substrate tools are tools in the sense that they influence and restrict the possible, not in the sense that we explicitly use them in order to achieve it. Substrate tools are part of the context of a problem.

I am using a word processor to write this document, and the substrate in this case is language. I can't make up new words (at least, not too many), or violate collective ideas of grammar (at least, not too badly). What I can do is use the material of language to formulate, and then communicate, my ideas.

This last example, though, shows how loose these categories are. I am constrained by language, as I write this, but I am also actively using language to refine my ideas – revising as I

write, playing out connections that I had not fully seen until I began to write. So language functions here as both a substrate and a rich cognitive tool.

Category overlap is the rule rather than the exception, and this is why talking about substrate tools is useful. No activity takes place independent of context, and our tools evolve dialectically: we create artifact and cognitive tools to match a specific context; when we use those tools they effect change of that context, which leads to the need for new tools, ad infinitum.[6] The context of a problem is a crucial part of the development of our ideas and machines, because we build our ideas and machines out of, and in direct relation to, the material of context. Substrate tools are contextual material that is pressed into service as backdrop and basis for thinking and tool building.

Of course, artifact tools tend to have strong cognitive analogs, too. The cliché, "When all you have is a hammer, everything looks like a nail," refers to this duality. Musicians talk about "composing at the piano," and designers similarly about "thinking over the drafting table." In fact, the triptych typology of artifact, cognitive and substrate tools is itself a cognitive tool – in this case, a tool for thinking concretely about tools, and (like all tools) more useful in some circumstances than others. To paraphrase another old saw, you can't think about tools without thinking about thinking about tools.

1.1.2 The Benefits of Meta

We have now come full circle, back to the discussion of computers as meta-tools. The categories discussed above serve as framework for describing the uses to which computational generality can be put. I would like to argue that computers are not just flexible machines, but that they are flexible in a particularly interesting way: we can build environments inside the computer

---

[6] This assertion about what might be called the dialectic nature of artifact production derives from Hegelian and Marxist historical theory. More recently, the biologist Stephen Jay Gould has written about these issues in describing the differences between the historical sciences and the historical study of culture and cultural evolution. Gould, S. J. *The Panda's Thumb; More Reflections in Natural History* (1980).

in which artifact, cognitive and substrate tools are coextensive, carefully matched and tightly integrated. Seymour Papert's evocative term for such an environment is "microworld."

The original microworlds, described in Papert's 1980 book *Mindstorms*, were programming environments built around the Logo language. The most familiar features of Logo are its simple, easily-understood syntax, and its use of "turtles" as graphical actors. Logo is often used in classrooms, and provides a completely new way for children to explore geometry and problem-solving.

Logo's substrate is a graphical, cartesian plane.[7] Line segments and angles are the sole building blocks, encouraging children to think about how to build complex patterns using very basic components. There are no built-in curves, for example, in classic versions of Logo; children learn that they can construct arcs from many tiny straight lines.

The Logo programming language is an artifact tool. Primitives such as **forward** and **right** control the placement of lines on the screen. The "turtle" on the screen is also an artifact tool, which helps a programmer to track position and orientation information.

But of course, the Logo language and the turtle are also cognitive tools. An experienced programmer learns to think in the idiom of whatever language he or she is using, and children who program in Logo are no exception. The vocabulary and the syntax of the language are designed to encourage certain styles of thinking (more on this in a moment). And children tend to anthropomophize the turtle. Papert argues that this tendency to see problems from a turtle-ian perspective is very powerful; children imagine themselves into the geometric world, walking through problems in their mind, pretending they are turtles inside their own flatland.

---

[7] More properly speaking, the cartesian plane with line segments and angles is the substrate of "turtle graphics," an important Logo-based microworld. Because turtle graphics and Logo are so often used together, and I am interested here in the combined features of this particular microworld and the Logo programming language, I will refer to them together as Logo, throughout. For more information on turtle graphics see, Abelson, H and diSessa, A. *Turtle Geometry* (1986).

Logo's tight integration of substrate, artifact and cognitive tools is a luxury not often enjoyed outside the computational domain. Carpenters and joiners, for example, who have a rich tradition of inventing tools and approaches to match the job at hand, have far less control over substrate than does the designer of a Logo environment. Even architects, famous for ignoring practical or prudent parameters, are much more constrained by the natures of wood, concrete, glass and steel than a microworld inventor is by the nature of bits and pixels. In the real world, we usually design tools around the constraints of a particular medium, and layer our cognitive approaches on top of this sometimes uneasy marriage. On the blank canvas of a computer monitor, we can create matched sets of substrate, metaphor and artifact together and from scratch.

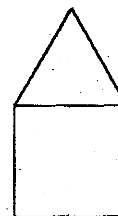
1.1.2.1 Learning to Solve Problems: Procedural Decomposition

One of the major cognitive tools that children learn as they become proficient in Logo is the principle that complicated problems can be broken down into less-complicated parts. The Logo syntax (along with Logo culture) encourages programmers to write their code in sections, called procedures, and "procedural decomposition" is the name the Logo community uses when talking about this style of problem-solving.

Procedures are used to encapsulate actions that are performed frequently, saving a programmer from retyping the same lines over and over. But procedural-decomposition is not just a labor-saving device; well-written programs use procedures as logical building blocks, so that the way the programmer thinks about a problem is obvious from the way the program is designed:

```
to house
right 90
square
triangle
end
```

This program, taken from Papert's Mindstorm (page 61), produces the following picture:



```
to square
  repeat 4 [
    forward 100
    right 90
  ]
]

to triangle
  repeat 4 [
    forward 100
    left 120
  ]
]
```

The house, it is clear from the source code, is not just a collection of lines, in the child-programmer's mind, but a triangle sitting atop a square.

The structural clarity that follows from procedural decomposition also sets the stage for learning to think about general approaches to problem-solving. Procedures can be written that perform differently under different circumstances, allowing programs to be written that deal with a class of problems, rather than a single instance:

```
to house :size
right 90
square :size
triangle :size
end

to square :side
  repeat 4 [
    forward :side
    right 90
  ]
]

to triangle :side
  repeat 4 [
    forward :side
    left 120
  ]
]
```

The house program can now draw houses of different sizes. Typing `house 200`, for example, draws a house that is twice as tall as one drawn by the command `house 100`.

Finally, a fluent Logo thinker is able to conceptualize problems in ways that would not occur to someone who doesn't "speak" the language. Recursion is one such conceptually advanced technique. Logo allows a procedure to invoke itself, so that a programmer can build a metaphorical stack of actions.

```
To spiral :distance
  forward :distance
  right 90
  spiral :distance + 5
end
```

This program, again borrowed from Mindstorms (p71) draws a square spiral, using recursion. Because the program calls itself as a subroutine, five lines of code do a job that would seem to require many more.

Recursion is not an obvious approach. In fact, many people find recursive procedures to be topsy-turvy or counterintuitive when they are first exposed to them. But a cognitive toolkit that includes Logo procedures allows a child to develop an experimental understanding of how recursion works and what it can be used for. With recursion itself added to the toolkit, a new logical lever becomes available for prying apart the subtleties of phenomena as diverse as linguistic meaning, cellular differentiation, and the calculation of compound interest.

1.1.2.2 Another Analytic Tool: Emergence

Research into kid-friendly programming dates back 30 years at MIT, and Logo has evolved a number of variants. Many of these are designed to help children develop specific sets of cognitive tools. StarLogo, for example, focuses on emergent phenomena – the interaction of simple processes to produce complex, often surprising, aggregate results.[8]

StarLogo allows users to write programs for many turtles at once, with all of these turtles coexisting, and perhaps interacting, on the screen. This approach, which computer scientists call massive parallelism, turns out to be a powerful way to think about certain kinds of problems. The Starlogo language and graphical displays provide handles for "getting a grip on" this thinking style in the same way that the original Logo syntax and turtle geometry provided a new way of learning algorithmic thinking and step-by-step problem solving.

Starlogo builds on a Logo foundation, and children who are conversant with Logo can learn to think in Starlogo just as a musician might take up a new instrument, or as a historian might tackle the study of an unfamiliar area or era. The new cognitive tools children learn from StarLogo complement old ones, and learning to think in a variety of ways is an important part of learning to think well. For example:

```
to circle
  repeat 360 [
    forward 1
```

These three procedures suggest three different ways to think about what a circle is. The

[8] StarLogo is described in Resnick, M. *Turtles, Termites and Traffic Jams* (1994). StarLogo is also freely available for download. http://www.media.mit.edu/starlogo/

```
    right 1
  ]
end

to starlogo-circle
  create-turtles 5000
  setxy 0 0
  forward 50
end

to patches-circle
  if
  ((patch-xpos)*(patch-xpos)) +
  ((patch-ypos)*(patch-ypos)) =
  (50 * 50) [
    set-patchcolor green
  ]
end
```

first procedure is the classic turtle-geometry circle expression; it draws a polygon with so many sides that it seems like a circle.

The second procedure, `starlogo-circle`, uses many, many turtles, all with random headings, to form the outline of a circle. This way of thinking about a circle is unconventional and surprising; from the chaos of 5000 turtles pointed every-which-way, a shape appears as if by magic. It's not magic, of course, but rather a nice lesson about randomness, large populations, and why a circle's radius is worth thinking about.[9]

The third procedure, `patches-circle`, is also a StarLogo program, this time using patches (the squares that make up the StarLogo plane), rather than turtles. The patches-circle algorithm is more like what you might find in a traditional high school geometry class than either of the other two.

1.1.3 The Tiles Project

The goal of this thesis project has been to develop a set of tools that encourage children to think deeply about computational networks. Taking Logo and StarLogo as models, this involves identifying core concepts and ideas, and building tools that allow children to explore and to "play with" – in more than one sense – these ideas.[10] Or, to recast the process in my substrate/cognitive/artifact tools vocabulary: the goal is to create raw materials that provide a meaningful context for exploration of networks, to identify or develop a set of useful, rich, rewarding cognitive abstractions for thinking about networks, and to figure out how to represent those abstractions as part of an accessible programming environment.

1.1.3.1 Substrate Choices

Three very different approaches to building a set of substrate tools were considered.

First: to create a simulation of networked computers on a single desktop machine. Simulation can be a powerful technique because the designer of a simulation can exert fine-

---

[9] Resnick, M. *Turtles, Termites and Traffic Jams* (1994) pp 95-6

grained control over the simulated world. It is possible in a simulation, for example, to carefully balance sophistication against comprehensibility, complexity against speed of execution, and variance against repeatability. The advantages of simulation are precisely the advantages, described above, of computational systems in general: the chance to create a microworld from scratch, tailored to encourage certain kinds of learning and exploration.

But the disadvantage is that a simulation is just that, a simulation. Working in simulation suggests a particular relationship between user and system, different from those that exist in systems that users perceive as "real." I wanted to create a system that extended beyond the boundaries of a single computer screen, that users interacted with as a multi-computer network.[11]

The second possible approach: to build a system spread across the Internet, in which many desktop computers could play a role. My interest in network architectures and networked systems stems from using the Internet, so building a net-centric environment was very appealing to me. I remember how exciting and revolutionary the World Wide Web first seemed, even on an old, slow computer through an even slower dialup phone line. To attempt to create a microworld that captures the sensation of extraordinary scale, dizzying connectedness, and (somehow, paradoxically) community intimacy that I experienced when first exploring the Web was very tempting.

However, the great gift of flexibility that the Internet offers – I can talk to any computer on the net from any other – eventually came to seem *too* open for this project. I wanted to make interacting with the system feel very direct and tangible, which eventually came to mean two things in my mind. One: structured connectivity. I wanted a consistent, comprehensible pattern of connections between computers in the system. Users need to be able to visualize the web of connectivity that defines the network, and need to be able to rely on this substrate pattern as they

---

[10] On playing and playing with ideas, see, Resnick, M., Bruckman, A., and Martin, F. "Pianos Not Stereos: Creating Computational Construction Kits" (1996)
[11] For a discussion of simulation and software, see Starr, P. "Seductions of Sim" (1994).

build an understanding of interactions across the network. This suggests that developing a simpler, more restrictive topology than the Internet all-to-all model is important. Two: physical proximity. When I'm learning about the system and discovering the mechanics of networked communication, I want to be able to watch my programs at every step, and perhaps even to intervene in their execution or movement. I want the nodes in the network to be accessible and the relationships between them to be configurable. If one of my programs wanders off to another computer, I want to be able to follow it. If the computer where we wandered to is on the other side of the room, I can do that, but it's harder if that computer is on the other side of the building, and impossible if it's on the other side of the state. Most people don't have a room with ten (or twenty, or two hundred) computers in it, so a non-Internet – indeed, non-desktop-PC-based – implementation came to seem critical.

So I chose to follow a third path: to build a set of tiny, "kid-scale" computers that can be arranged together to create a miniature network. The precedents for this approach, along with the advantages of being able to physically manipulate these little computers, will be discussed futher in the section below, *Tiny Computers and Big Networks*.

Each of my purpose-built computers is a square tile, 2.5 inches on a side. Each Tile can communicate with neighboring Tiles lined up along its four sides. The Tiles can be arranged in two-dimensional patterns, like squares in a crossword puzzle, and adjacent Tiles "fit together" logically as well as physically (because only adjacent Tiles can pass bits back and forth).

Because the communication topology extends directly from the physical topology, understanding and visualizing interactions between the Tiles is quite straightforward. And the network is easy to reconfigure – as easy as picking up a Tile and moving it – with feedback from such a change occurring immediately.

1.1.3.2 Cognitive Abstractions

This project grows out of an exploration of metaphors and mechanisms for programming networked computers. The traditional tools for network communications are pipes and sockets, messages and procedure calls. I have, over the past two years, worked to develop a *mobile code* approach to writing programs that are inherently network-savvy.

It is useful, in many contexts, to think of programs as "living" on more than one computer. World Wide Web indexing robots, for example, often ascribe to the metaphor that they are moving from page to page, server to server – Web robots are known colloquially as "crawlers." In fact, these robots request information from disparate servers, but always execute on a single machine. What would it be like to be able to write programs that actually move from computer to computer?

It turns out that there is growing interest from a variety of computer science communities in the general area of state-encapsulated, movable programming.[12] Researchers with specific interest in programming language research, intelligent agents, and network design are all contributing to an increasingly-relevant dialogue. We are coming to understand, more and more, that current implementation practices will not scale adequately, and that the complexity and ubiquity of next-generation networks will require something different.

I wrote some simple tools in Java to work through my basic assumptions, spent some time thinking about what I really wanted, and read all of the research that I could find from people with similar ideas. The outlines of a project began to solidify in my mind, a project to create a set of tools that would make it possible to explore both the subtleties of writing movable programs and the prerequisites of an architecture that would make writing them possible.

---

[12] For some discussion of issues relating to mobile, object-oriented programming, see Tschudin, C. ed. *Mobile Object Systems : Towards the Programmable Internet* (1996). For a bibliography of academic papers related to mobile code research, see http://www.cnri.reston.va.us/home/koe/bib/mobile-abs.bib.html. For an extensive list of references on Mobile Agents and Distributed Objects research, see http://www.cetus-links.org/oo_mobile_agents.html

That project ultimately became the core work of my thesis. I have tried to construct a "toy system," which, taken as a pun, suggests two identities. The Tiles are a simplified, carefully-abstracted network of computers. Like "toy problems" in science, my toy system tries to excavate the complexity of the real world, highlighting crucial difficulties and exposing crux elements for examination. On the other hand, my creation is a toy system in the playful sense: I want the Tiles to be fun, interesting and compelling, to be toys that compute with each other, like Lego bricks are toys that attach to each other and slot cars are toys that race one another around a track.

The key idea embodied in the Tiles system is that a computer program can exist inside a network, rather than on a single computer. No one in the real world (no one writing production code that someone else depends on) writes programs like that, today. But in the near future, I think, we will be using mobile code, and using it for all kinds of things. A significant shift in the way professional programmers think about system architectures is on the horizon.

If Logo was designed to give children tools that allowed them to think in terms of procedural abstraction, the Tiles system is an attempt to offer tools that encourage thinking about processes interacting and changing over time. A Tiles program is a self-contained process that can exist through numerous shifts in computational context; writing a program for the Tiles that does much of anything interesting involves thinking pretty carefully about what it means to weather changes in context, to adjust successfully to unpredictable circumstances, and to communicate effectively with other programs that may share one's computational turf.

Tiles programs are each separate logical entities just as the Tiles themselves are separate physical objects. Programs move from Tile to Tile in a straightforward, comprehensible way, and can interact with other programs they meet as they travel. Users can create and modify Tiles worlds in two ways: they can write programs (possibly a great many programs) that inhabit a set of Tiles, and they can directly arrange (and rearrange in real time) configurations of Tiles for their programs to explore. The programming paradigm and the physical design of the system, cognitive

tools and substrate, were developed as a matched set: simple, mobile pieces of code and simple, kid-scale, movable computers.

1.1.3.3 Artifact Tools

The Tiles system depends on two categories of artifact tools. First, the feedback that the Tiles give to the user, as programs hop around the network. Second, programming and visualization tools running on a desktop computer that allow children to write the code that runs on the Tiles. Skeletal frameworks for both Tile output and desktop environments exist, but a great deal remains to be done in this area. A discussion of some possibilities, and of an array of short- and long-term goals, occupies the final third of this thesis, *Future Work.*

1.2 Mobile Programs: Tiny Computers and Big Networks

During the past two years, I have worked in the Epistemology and Learning Group building systems of very small computers. A desire to make hardware easier to use and less expensive than anything else available, so that we could embed real, programmable computers in rubber balls, palm-sized LEGO models, or children's plastic jewelry, provided initial motivation for this research.

As work progressed, and we began to experiment with our new components, it became clear that their matchbox scale suggested some very new ways of working within, and thinking about, systems consisting of multiple interacting computers. All of a sudden, we could easily have ten (or twenty, or thirty) computers in front of us on a desk, all crunching code and broadcasting messages. And we could dynamically reconfigure the system quickly and easily, by simply taking away some of the little computers and replacing them with others.

These tiny platforms began to change the way we talked about digital communications in general. Small, simple and physically manipulable, they pushed us to think hard about the core ideas behind networking infrastructure and implementation, and not to simply assume that all

networks should be some version of the star-configured, packet-passing substrates we are so familiar with.

In particular, we found that we were usually far more interested in how collections of our little computers acted together than in the behavior of any single one of them. We had anticipated that communication, cooperation and emergent behaviors would be important characteristics of our new systems, and our work with StarLogo was certainly a good grounding for thinking in "more-is-different" directions.[13] But still, we were surprised by how compelling it was to experiment with the interactions between our little computers – how we gravitated toward projects that involved several of them rather than projects that used just one.

The Dancing Crickets, which are a pair of wheeled robots that do the cha-cha, were programmed by Rick Borovoy and are an early example of this phenomenon. We still show the Dancing Crickets as an introduction to our work with tiny computers, and they never fail to elicit smiles from audiences.[14] There is something magical about the two crickets interacting, "dancing" together. Communication is such an important part of our lives; perhaps we find it easy to anthropomorphize computers that are so clearly communicating, and easy to identify with computers capable of behavior that mimics our own.

The scale of the crickets reinforces our intuitive identification with them, I think. We can see both crickets together in front of us, and can easily pick them up, move them around, or block the infrared signals they are sending one another. I wanted very much for the Tiles to have a similar immediacy, to be physically manipulable and part of the tangible world. Like the crickets, the Tiles are small enough that a large number of them can be arranged and rearranged by a single person in a compact space. Seventy-five of them fit on a small table with room to spare, or on the

---

[13] The phrase "more is different" is taken from the title of a paper that appeared in the journal *Science*, Anderson, P. W. "More is Different" (1972).
[14] For information about the Crickets see, http://fredm.www.media.mit.edu/people/fredm/projects/cricket/

floor within reach, which makes it easy to build and observe complex topologies with numerous potential interactions.

Rick and I developed a set of Programmable Beads as part of this same research agenda.[15] The beads are never used singly; they only make sense when several of them are used together. The classic demonstration of our Programmable Beads involves a string of them configured to "pass" a light from one to another, down the line and then back up. Because the beads slide along the string, and communicate only over a short distance and with their nearest neighbors, it is possible to "catch" the light at one end of the string, or to physically bridge the light across a gap by moving a bead at just the right time. This has proven to be a popular and evocative display of our new technology, for the metaphor by which one understands the dynamics of this simple string is both intuitively obvious and an important reference point by which to understand more complex interactions between beads.

Unfortunately, this metaphor – that the light "moves" – is not the metaphor used in the underlying program code. The underlying program, written in a traditional procedural language, must create the "illusion" that the light moves using control messages traded between neighbors. So while the person playing with the string sees a light hopping from bead to bead, as far as the program code on each individual bead is concerned, the LED is simply illuminated as an epiphenomenon of the message transactions. This disjunction between the perception of the string of beads and how the perceived behavior is implemented programmatically always seemed to me far from ideal.

As I worked with the beads and crickets, I became increasingly interested in developing new ways to think about them and about similar systems of interacting computers. Traditional programming idioms describe sequential computation on a single processor, but, as the beads example shows, are not very good at capturing the larger-scale behaviors that we found so

---

[15] For information about the Programmable Beads see, http://el.www.media.mit.edu/projects/beads/

interesting in our new networks. A major goal of this thesis project has been to build a system in which the disjunction between system-level behavior and individual computation at each logical node disappears.

Complex patterns can often be specified by describing simple, component behaviors.[16] This idea, that interactions between simple parts is a useful way to think about complex, far-flung effects, provides a starting point for developing a new approach to programming our communicating computers. The challenge, from this perspective, is to find a good set of "building blocks" that help us think about systemic processes. The beads provide the clue: if we are interested in the light, we should be able to write programs that encapsulate the light. If we want the light to move, our programming language should provide constructs that make that possible, rather than forcing us to fight against the environment to create a simulacrum of that behavior.

It turns out that we very often want our "programmatic objects" to move, in the world of the beads. Thinking about programs – as in the light example – that pass from bead to bead feels like a natural way to express a lot of the behaviors that we care about. Some behaviors, like the simple moving light, are adequately described by a single mobile program. Others, like patterns that behave as standing waves, can be thought of as several programs sharing space on a single string. I wanted to create a programming system biased in favor of simple programs that are able to move around in a network, and explore the usefulness of these pieces of mobile code as conceptual building blocks.[17]

The Tiles were designed from the ground up to be a mobile-code system. Processes are explicitly represented as mobile – able to move seamlessly from Tile to Tile – for the simple

---

[16] For three different perspectives on complexity and emergent behavior, and on programming computers in ways that address the aggregation of simple parts, see: Resnick, M. *Turtles, Termites and Traffic Jams* (1994); Kauffman, S. *The Origins of Order* (1993); and Kiczales G., et. al. *The Art of the Metaobject Protocol* (1991).
[17] Several systems exist that make it possible to write certain kinds of mobile programs on desktop computers. These systems are important precedents in this regard. For descriptions of two of the most workable and well known, see: Lange, D. and Chang, D. "IBM Aglets Workbench: Programming Mobile

reason that most interesting play involves writing programs that effect more than one Tile. The system is purposely simplified toward homogeneity and explicit local communication, so that communications paths between Tiles are clear and it is easy to visualize how a program moves across the network. The Tiles architecture is as "transparent" as possible, offering an avenue into the exploration of mobile-code dynamics at every level of the system, and is a research platform from top to bottom. Real-world networks must account for performance, robustness, security, backward compatibility and the like; the Tiles offer an opportunity to concentrate on the implications of mobile-code design rather than deal with the tradeoffs required of production implementations.

Each program in Tiles world is supposed to be fairly simple. (Limitations on size, along with the structural properties of the programming environment, encourage this.) But the collective behaviors of several programs across a set of Tiles can be extremely complex. Writing programs for the Tiles, and balancing the often surprising effects that programs have on each other and their environments, teaches this lesson quite effectively. The Tiles, in large part, are intended as a tool for learning to think about the results of interactions between numerous independent processes.

Here work on the Tiles overlaps with traditional computer science research, particularly research on distributed systems and autonomous agents. How to build complex systems from simple, comprehensible parts is a central question for computer scientists, and the Tiles are an excellent test-bed for exploring a multi-layered, decentralized approach to these issues. The world of the Tiles is clean and purpose-built, free of many of the constraints of real-world systems. Computational efficiency is not of great concern, nor is system security, and each Tile is simple enough that we can expose and scrutinize its inner workings. We are free to build, with the Tiles, an interesting model in which every traditional property of an "operating system" is created by mobile programs. Following Huberman, we might call these cooperating programs, along with

---

Agents in Java, A White Paper" (1996) http://aglets.trl.ibm.co.jp/whitepaper.htm; and Gray, R. "Agent Tcl:

other programs that use the services they provide, a computational "ecology," highlighting the fluid, organic feel of such an approach.[18]

Mobile programs create a system structure for each Tile, and other mobile programs use that structure. All large computer systems are built from layers and layers of code, but the Tiles architecture is extreme in its homogeneity and flexibility – each of the layers of the system is just like all the others, built from extremely simple, mobile, changeable parts. The core principle of the Tiles system, that we can create complexity by controlling the interaction of simple processes, extends all the way down below the user level through the "operating system" and down to the tiny kernel that governs bit-by-bit communication.

Two theoretical advantages of the "mobile code all the way down" approach for networks of computers:

1) Flexibility – a system built of mobile objects is inherently changeable over time. Flushing the system and starting over is accomplished simply by diffusively repopulating the nodes. Interesting problems regarding graceful accommodation to change present themselves (how to handle object versioning or to plan dispersion or node tracking). More traditional approaches simply don't allow such incremental revision.
2) Scalability – there need not be fixed limits in the number of nodes in a given system. As mobile objects take the form of independent and distributed encapsulations of data, behavior and intensionality, there are no centralized resource bottlenecks. Such systems tend not to "break," but to degrade as complexities, "geographic" dispersions, or sheer numbers grow beyond the comfort-thresholds of particular algorithms.

A transportable agent system" (1995) http://www.cs.dartmouth.edu/~agent/papers/cikm95.ps.Z

It is worth noting that the second property is not always considered an advantage. Telephone networks, for example, are designed with a sharp delineation between functionality and non-functionality (you have a dial-tone or you don't): the advantage of marked component failure is that it's sometimes easier to understand definitive failure than "emergent" performance degredation. For the former, a single problem point can usually be identified, whereas the latter may result from the long-term interactions between numerous semi-separate sub-systems.

A major goal of my thesis work is the development of tools that suggest new ways of thinking about such scalability problems. The Tiles are a baby step towards a new approach to designing networked systems, as well as an incubator for new ideas about how to conceptualize, analyze and dissect their behavior.

---

[18] B.A. Huberman. *The Ecology of Computation* (1988). For a discussion of various approaches to building loosely-coordinated mobile agent systems, see Minar, N. "Computational Media for Mobile Agents" (1997) http://nelson.www.media.mit.edu/people/nelson/research/dc/

2 Tiles – Design and Implementation

The Tiles system is an open-ended microworld – children can write new programs, build their own configurations, and craft ideas using the raw materials of tiny computers and mobile code. The Tiles are also open-ended at a lower level – as a computational platform they are quite reconfigurable, so that both the hardware and the software will be able to evolve over time. Building a flexible system (from both of these perspectives) was a primary goal from the beginning of the design process.

2.1 Hardware

Each Tile is a tiny computer, with a single-chip microcontroller as its functional core. Our research group has been building microcontroller-based hardware for several years, so I had a wealth of second-hand experiential knowledge to rely on in making the early design decisions on this project. Working with two of our other research platforms, the Crickets and the Programmable Beads, had convinced me that over time we would find ample use for as much computational power as I could find a way to shoe-horn into a small form factor. In the Tiles context, computational power means three things: microcontroller speed and sophistication, memory size and access time, and communications bandwidth. In addition, in order to put their computational abilities to good use, the Tiles must be able to accept input from and provide feedback to their users. And finally, each Tile runs on batteries and requires circuitry to support this power supply.

2.1.1 Microcontroller and Memory

The first step in designing the Tiles was to choose a microprocessor, as that choice would influence every other subsequent hardware decision. After examining the product lines of

Arizona Microchip, Hitachi, and Motorola (among others) I settled on the Motorola 68HC12A4. The 6812 is a reasonably fast, sixteen-bit microcontroller, has an excellent array of on-board peripherals, and can be configured to automatically interface to external memory. In addition, Jim Sibigtroth of Motorola Semiconductor offered to help get us started working with the chip that he had designed, and promised to be available to answer some of the inevitable questions that arise when working with a new piece of hardware.

The 6812A4 has one kilobyte of internal memory (along with 768 bytes of slower, non-volatile eeprom). That amount is an improvement over the 84 bytes of the PIC16LF84 we used in the Programmable Beads, but still not nearly enough for building complex, multi-program, mobile-code projects of the kind that I envisioned. My goal – absent of practical considerations – was to include enough synchronous, random-access memory on each Tile that we could never think up an activity for which we lacked necessary RAM. Microcontroller manufacturers as a rule include relatively little RAM in their designs, however, so the next step after settling on the 6812 was to design the interface to an external SRAM chip or chips. Fortunately, this is relatively straightforward. The 6812 provides a full-width external bus, in addition to some nice software features that allow memory larger than 64 kilobytes to be paged almost automatically at the hardware level.

External memory chips are quite large (because of the number of pins that parallel access to data requires), so they take up quite a lot of space on a circuit board. This, along with the relatively high cost of SRAM, set an upper bound on the amount of memory practical for each Tile. I found a 128k, byte-wide fast SRAM device made by Motorola that satisfied my criteria and was available in prototype quantities and an industry-standard package. This part, MC6326B, can operate at the regular eight megahertz bus speed of the 6812, so the system never needs to wait for data to be available from external memory. Bus speed seemed important to me, as mobile programs would often be stored in, and executed directly from, external memory. Using a pair of byte-wide devices together could have increased the effective speed of the bus even more, but

two SRAM chips would have taken up too much space on the circuit board and increased the price of each Tile substantially.

My back-of-the-envelope calculations settling on 128 kilobytes of memory were as follows: to accommodate 50 mobile processes at any one time on any one tile would require 100K of external space if each process occupied a two kilobyte block of RAM. The standard 128Kx8 chips make this possible, with a little extra breathing room besides.

2.1.2 Network Links

In a network environment, effective computational power is often limited by communications speed; a fast processor does you little good if that processor is always waiting to send or receive data through a narrow "pipe." The Tiles are extremely network-centric, and the demands on the network channels can be relatively large because entire programs need to move from Tile to Tile. For this reason, it was important to find hardware that could handle high bit-rate communications over the short distances between Tiles.

I also felt that it was important to make the Tiles able to communicate without having to be touching. We worked hard on the Programmable Beads to achieve "contactless" communication, and I think that that work paid off. Much of the magic of the Beads is due to their easy, intuitive "interface": slide two of them close together and they "talk," slide them apart and they do not.

The neighbor-to-neighbor design of the Tiles (which is similar to the design of the Beads) made it easy to rule out radio-frequency communication as a possible strategy. Tiles should only talk to other Tiles that are right next to them, and each side of each Tile should be able to carry on a separate conversation with its respective neighbor. RF would have been a poor choice on both counts, as distance-limiting would have been difficult, and radio is not particularly directional at frequencies that would have been practical for us to use.

Inductive coupling, which we had used in the Programmable Beads project, was another possibility. However, inductive coupling approaches suffered from two major drawbacks, low bit-rates and a lack of off-the-shelf parts. I built the transceivers for the beads from scratch, winding coils by hand and using discrete capacitors and diodes to feed a low-power comparator. Even with considerable tweaking I was only able to achieve a data rate of about 200 bits per second using the necessary small, low-power chips. In the end, the system I built for the beads was extremely reliable and had a nice form factor, but was impossible to manufacture in quantity and was quite slow.

The third obvious candidate for the Tiles network physical layer, along with radio and inductive coupling, was low-power infrared. Infrared devices are relatively power-efficient, inherently directional, and available in off-the-shelf integrated circuits. The major outstanding question was whether I could effectively limit infrared transmission to achieve a consistent "connection" range of one to two inches. The infrared parts we used in the crickets were actually quite inconsistent, exhibiting extreme sensitivity to ambient light, with transmission range and reliability falling off markedly under both halogen and sunlight.

After experimenting with a variety of high bit-rate receivers and transceivers, I settled on the Sharp IS1U20, a simple, three-pin receiver that is able to trigger about every eight microseconds, meaning that it can receive a maximum of about 110 kilobits of data per second. The IS1U20 proved admirably robust under a variety of light conditions, and required the addition of only a single (though rather large) bypass capacitor to be fully functional.

We had discovered, while developing an earlier infrared design, that our standard red/green bicolor LEDs interfered with infrared reception. I wondered if I could use a visible red LED to transmit data, making it possible to actually see the communication between Tiles. It turned out that of all the LEDs I tested, only the original bicolor LEDs are "messy" enough in the infrared to stimulate the receiver (and, perhaps unsurprisingly, only when turned on red, not green), but that those particular bicolors are extremely consistent and useful as short-range

transmitters. Sending 15 milliamps at five volts through the red side results in a transmission range of just under two inches.

2.1.3 User Input-Output and Expandability

As the core hardware choices – microcontroller and support components, memory, infrared receivers and LEDs – began to fall into place, it became clear that all of the desired circuitry, including a rich variety of user input and output devices, would not fit onto a single circuit board. The Tiles needed to be fairly small; making them square and 2.5 inches on a side was my goal. A double-decker design, with core functionality on one circuit board and additional components on a second board stacked atop the first, was one way to fit more parts into the 2.5 inch form factor. Taking this approach meant that I would be able to revise the top board over time, or even have multiple top boards for different Tiles, without changing the core microcontroller/memory/communications part of the design.

I think the flexibility of the two-board design will prove valuable over the long term. However, no top boards have been designed yet! We have manufactured 200 Tiles, but are using them with only the minimal input and output included on the core boards. These minimal features are: 1) a single pushbutton and 2) the bicolor LEDs used for infrared communication, which can also be used as visible red or green lights.[19] Multiple top boards are on the drafting table, which include such features as dot-matrix arrangements of LEDs, light-sensors, and piezoelectric speakers, and will be discussed in section three, *Future Work.*

2.1.4 Battery Power

The final necessary piece of the Tiles hardware is the power supply circuitry. I wanted each Tile to be a self-contained, completely portable computer, so some form of battery power

---

[19] The two modes – communication and visible display – don't interfere, for the most part. The operating system software flashes the light very quickly, while sending data, so that only a faint red glow is perceptible to the human eye.

was needed. After experimenting with a variety of voltage regulators, inductors, switches and relays, I finally settled on a simple, expedient solution – powering the Tiles directly from four rechargeable, nickel-cadmium, AA batteries. Four NiCads deliver current at between 5.2 and 4.8 volts, more or less obviating the need for a separate regulator, and the resulting power supply is clean and stable.

The batteries can be recharged (without removing them) via a three-terminal header, and Fred Martin designed a board that can charge ten Tiles at a time. Thus far the power-supply circuitry has worked extremely well, and the simple solution of NiCads without a regulator seems to have been a good one. The AA batteries we are using advertise a current capacity of 600 milli-amp hours, and a fully-functioning Tile consumes about 60 milliamps, so we get about 10 hours of operation under continuous use. The Tiles kernel that we wrote, however, is configured to switch into a sleep mode after a period of inactivity, reducing current consumption to around 10 milliamps.

2.2 Software

The Tiles require three different sets of software. First, each Tile has a tiny, but crucial, operating system kernel that handles core tasks. Second, mobile programs live in networks of Tiles, hopping from one to the next. And third, software on a larger computer – such as a desktop pc – is important, so that programs can be written and downloaded easily. Brian Silverman and I worked together on a first-generation kernel for the Tiles, and I've written a number of mobile programs. Much exciting work remains to be done at all three levels, however, and future plans will be discussed in the following section. Here, I would like to detail what has been accomplished thus far.

2.2.1 A Minimal Kernel

The goal in writing the core system code for the Tiles was to do as little as possible as well as possible. As discussed in the first section of this thesis, interesting functionality in Tiles world is supposed to be built from layers of mobile code, rather than included as part of a static, resident operating system. There are tradeoffs and practical considerations involved, here, even with a purpose-built platform such as the Tiles, and those will be discussed shortly. First, however, I'd like to give an overview of how the kernel that Brian and I wrote works.

There are three pieces of functionality that cannot be implemented as mobile processes: bit-level communications, first-order memory management, and task switching. The Tiles kernel must handle these.

2.2.1.1 Infrared Communications

Of the three components of the kernel, the communications code was the hardest and most tedious to write. As is often the case, conflicting criteria governed the development of the communications protocol and underlying implementation. I wanted to use a framework that would offer high-bandwidth, low-latency data transfer with minimal processor overhead, robust bit-transfer behavior and a simple, easily comprehensible encapsulation scheme. Obviously, the definition of all of these criteria is relative, and no single system can accomplish all things in all situations. What was obvious from the outset was that existing standard protocols were either too complex and messy (IrDA) or too slow and simplistic (classic Sony remote control protocol).

After a year of experience with the Crickets and Programmable Beads, I tended to gravitate towards simple binary encodings based on the timing between successive pulses. This approach, derived from that used in the Sony remote control protocol, is robust, easy to understand, flexible with regard to alterations (like the addition or subtraction of start bits, stop bits, and the like) and relatively straightforward to debug.

In order to satisfy the criteria of low processor overhead, and to make it possible for a Tile to hold a conversation with all four of its neighbors simultaneously, I wrote an initial version of the communications code that was completely interleaved and driven by hardware interrupts. The timer facilities of the 6812 allow a single pulse to trigger a piece of code asynchronously, so my software stored separate data streams for each transmitter/receiver pair and forwarded complete bytes to be interpreted by the other pieces of the kernel with a tag saying where they had originated. This system worked very well, but was unfortunately quite slow. The overhead of interrupt handling and context-switching for each byte pushed the maximum throughput on a single channel down to about twelve kilobits per second. Twelve kilobits translates to about 1500 bytes, so that a two kilobyte mobile program would take more than a second to move between two Tiles, which is unacceptably long.

The alternative to interleaved communication is to dedicate the processor to sending or receiving as fast as possible on one side at a time, and hope that other neighboring Tiles are willing to wait if they also would like to communicate. This solution is not nearly as elegant, but neither is it unprecedented; all real-world communications protocols assume that valid, available receivers will often be temporarily busy, and provide ways to deal with this. Brian Silverman and I re-wrote the communications code, dedicating the processor to dealing with bits on a single channel. The new code transmitted about 80 kilobits per second – a substantial improvement, and a safe compromise relative to a theoretical limit with our code of about 95 kilobits per second. This translates to 10,000 bytes per second, meaning that a two kilobyte mobile program requires only one-fifth of a second to transfer, a rate that falls well within acceptable limits.

There is only one kind of "message" allowed between two Tiles: an entire mobile program. This is a fairly radical abstraction, but is consistent with the Tiles philosophy that everything interesting should be built from mobile code. The protocol layer that sits atop the byte-by-byte code simply attempts to verify that the sender is asking to transfer a program, and if the incoming data appears valid passes it to the memory management and task switching modules.

There are hooks in the code for check-summing transmitted data, but I haven't yet written the necessary routines. Early testing suggested (to my surprise) that receipt of bad bytes was so rare as to be non-existent, but longer-term use has indeed shown that when a Tile's batteries start to get very low, or when users place Tiles just at the edge of transmit/receive range, problems arise. Implementing checksums is high on the list of incremental improvements that need to be made to the kernel code.

2.2.1.2 Memory Management and Task Switching

In keeping with the attempt to build simple, efficient services into the kernel that restrict as little as possible what can be built at higher levels, the memory management and task switching schemes are quite straightforward. Memory is allocated in two-kilobyte blocks, which contain both program code and stack. When a program moves from a Tile, the block is marked as free. Conversely, when a program arrives on a new Tile the memory manager scans the address space until it finds a free block, into which it writes the program code. Currently, the memory manager also constructs a new stack for a newly arrived program, meaning that program stacks are not passed from Tile to Tile. Like the checksum routines mentioned above, this will change with the next revision of the kernel.

The choice of two-kilobyte-long blocks is a compromise between providing enough space for programs, ensuring that programs must transfer themselves rapidly, and making the kernel fast and simple. I could have chosen a larger fixed size for memory blocks, or implemented a variable-sized memory scheme. The former would have allowed programs that are large enough to be noticeably slow to transmit and would also have limited the number of programs that can be stored in the 128 kilobytes of SRAM on each Tile. The latter choice would have made the kernel much slower and more complex, significantly increasing execution overhead (as well as the potential for bugs in kernel code).

The task switching system, like the memory manager, places as little responsibility for complex decision-making on the kernel as possible. The Tiles use cooperative, rather than pre-emptive, multi-tasking, so that programs are never forcibly interrupted by one another. This means that programs need to be polite and yield to one another at regular intervals, and that programs that fail to yield prevent other programs from running. It also means that users writing programs must think about the results of the interactions between those programs, and that users have a great deal of latitude in deciding how their collections of programs will behave. While cooperative multi-tasking is rarely the right choice for real-world systems, I think that forcing Tiles programmers to take responsibility for the lowest levels of cooperation between their processes is an important component of the educational and experimental character of the system.

Readers inclined toward skepticism may have asked why the memory manager and task switching logic needs to be built into the kernel, given the stated philosophy of removing everything possible from the lowest layer of operating system code. Couldn't objects dynamically and cooperatively manage memory space, and couldn't a task-switching module be part of a mobile suite of functionality rather than hardwired into the kernel? The answer to both these questions is yes. However, two goals for the Tiles architecture conflict: one, to build the system as much as possible from mobile programs and at run-time, and two, to provide a basic, functional core environment on each Tile so that even a very simple program is able to execute, wherever it may find itself on the network.

The way around this dilemma is to think of the kernel itself as built from a variety of pieces, all of which are potentially mobile and alterable. A program can move through the network updating the memory model, or the task switcher, or even the communications code, on every Tile it comes across. These changes would be semi-permanent, lasting until the next program that needs to rewrite the core libraries comes along. In this way, every piece of the system can be built by and from mobile processes, but critical processes leave behind permanent pieces of themselves. The operating system already takes this approach in places; code that is

used frequently by many agents (like routines that turn on and off the bicolor lights) are burned into the slow-to-change EEPROM rather than stored in RAM. The next step in this direction is to make the kernel core more carefully modular and abstracted, so that the communications code, memory manager and task switcher are not quite so tightly interconnected, making possible changes to each individual section, as needed.

2.2.2 Mobile Programs

Programs for the current generation of the Tiles are written, like the kernel, in Motorola assembly language. Assembly language is not very accessible, and far from the ideal representation for specifying mobile programs. The *Future Work* section below will discuss plans for a revised kernel that includes a byte-code interpreter, and new tools that allow programs to be written in higher-level languages. The current kernel and assembly-language mobile programs have been valuable mainly as proofs of concept, and will soon be replaced with a second-generation implementation.

A mobile program for the Tiles is an assembly-language program that knows something about its own format, and knows how to call kernel subroutines that will move it to another Tile, copy part of itself to another Tile, or end its execution and free the block of memory it occupies. A mobile program includes some header information that defines its length, the "direction" that it is facing (what channel to use for a move or copy operation), a starting address to begin its execution, and a clean-up address to use after moving it but before erasing the memory it inhabited. A very simple example, in Motorola assembler format, follows. Comments and explanation are delimited by semicolons:

```
$BASE 10T                           ; decimal default base
#include      "kernela.asm"         ; another file listing shared
                                    ; definitions

program
; header information
            dw       stend-program     ; length
            db       0                 ; direction
            db       0                 ; checksum (not used yet)
            dw       start-program     ; relative start address
            lbra     clean_up-program ; relative clean-up address
```

```
; the program logic
start
            jsr       redon_v        ; turn a red light on
            ldd       #1000          ; wait for one second
            jsr       delay_v        ; ---
loop        jsr       move_v         ; and move to the next Tile
            ldd       #500           ; wait for 1/2 second
            jsr       delay_v        ; ---
            bra       loop           ; and try again to move

clean_up    jsr       led_off_v      ; we turned a light on before
                                     ; we moved, so we need to turn
                                     ; it off, here in the cleanup
                                     ; routine.
stend       ; a placeholder to tell the assemble where the program
            ; ends.
```

3 Future Work

A great deal of work remains to be done for the Tiles to become the rich, flexible tools that I would like them to be. In particular, they are not yet accessible or simple to use – there is no easy path to beginning an exploration of the Tiles microworld. In this final section I would like to discuss plans for future development of the Tiles hardware, software and desktop environments. Some of these date back to my earliest conception of a "stacking blocks" project, and some have emerged recently as we have played with the first-generation, proof-of-concept implementations detailed in the previous section.

3.1 A Second Mobile Code Implementation

The foundation of the Tiles architecture is the operating system kernel on each Tile, the code that handles the core functionality of the system. As discussed in the previous section, several changes to the kernel are in order (checksums on infrared communications and increased modularity between subsections, for example), but a much larger revision is also planned: Tiles programs will be compiled and passed around as platform-independent byte-code, rather than as 6812 machine code. This implies that the operating system kernel will include an embedded virtual machine, and will take an active role in program interpretation rather than simply delegating code execution to the microprocessor hardware. The new kernel will be bigger and more complex (neither of which are virtues), but the benefits of a system built around an interpreter are three-fold:

3.1.1 Abstraction and Compactness

The machine code of the 6812 processor is not particularly well suited to describing inherently mobile processes. Direct addressing is the microprocessor's natural mode of operation, for example, whereas mobile programs use relative addressing for almost everything. Translating

a high-level description of a mobile process into machine code requires some inelegant compromises, such as repeated use of idioms that compile relative addressing into the indirect – from a mobile-code perspective the "almost-but-not-quite direct" – native addressing mode of the 6812. This observation is not a criticism of the 6812 cpu, which, like every other microprocessor on the market today, has an instruction set that is intended primarily to provide straightforward and efficient access to the underlying hardware, but it is evidence that there are potential benefits to taking a byte-code approach.

An appropriate additional layer of abstraction on top of the machine code will improve the "fit" between mobile processes and their representations inside the Tiles. This closer fit results in smaller program code; frequently-used routines and inelegant compromises are accounted for by the kernel's byte-code interpreter once each, rather than embedded over and over in the code of every mobile program, so a byte-code representation will generally be much more compact than its machine-code analog. This means both that more programs can fit in memory on each Tile and that the time required for a program to move from one Tile to another is reduced.

3.1.2 Safety and Avoiding Bugs

Another benefit of an extra layer of abstraction is the ability to isolate the processor from some kinds of program problems. It is very easy for a mobile program to crash the current kernel; a single errant machine-language instruction will usually send the processor spinning out of control. A byte-code system that is written with particular patterns of use in mind can provide safety precautions and sanity checks to guard against this. A misbehaving process can often be identified and politely dismissed, before it brings the system down around it.

3.1.3 Leveraging Common Tools and Prior Knowledge

There are two possible ways to go about incorporating a virtual machine into the Tiles kernel. The first is to design a new instruction set from scratch. The second is to adapt and make use of an existing byte-code language. The advantage of starting with a blank slate would be complete freedom to design a new, tailored representation. On the other hand, using a system that already exists would likely mean that tools (such as compilers, development environments, and the like) are available and useful, and that people who are already familiar with the existing system could quickly and easily begin to write code for the Tiles.

It happens that there is a byte-code system, called Java, in common use and under active (frantic, some might say) development by Sun Microsystems, and many other companies as well. Java was originally conceived as a way to send pieces of code across a network safely and efficiently and has a great deal of momentum behind it. Many people are comfortable programming in the language, and there are tools available for all common desktop platforms.

The advantages of building a Java virtual machine for the Tiles are clear, and Brian Silverman and I have been working on an initial implementation. There are some technical disadvantages, however, to using Sun's language. The Java instruction set has a number of features that are not useful on a platform as small as the Tiles, and the extensive object-oriented features of the Java virtual machine are not appropriate in our context, again for reasons of scale. We have chosen to use a subset of the Java byte-codes, and to simplify the format of compiled Java object files. Still, any experienced Java programmer, working on any desktop computer, will be able to write mobile programs for the Tiles immediately and easily, using tools and a language with which she is already familiar.

3.2 The Desktop Side of things

An improved kernel and embedded byte-code interpreter are the foundation for much more exciting developments: new tools that make the Tiles approachable, usable and engaging for

inexperienced programmers. I am looking forward to building accessible, intuitive ways to program the Tiles, graphical visualization and debugging environments that help children construct understandings of complex behaviors, and hooks that allow desktop programs to send code to Tiles.

3.2.1 Programming Representations

As discussed in the first section of this Thesis, the Tiles were conceived as a microworld designed to encourage experimentation with networked computation. The system was built around the metaphor of mobile code, the idea that programs should be able to move easily from computer to computer. Developing new and appropriate representations for mobile programs is a crucial part of this project, along with the implementation of tools that allow these representations to become real, executing programs.

Representations – programming languages – function as cognitive tools that allow us to instantiate the mobility metaphor. Programming environments – editors, compilers, graphical shells, and the like – are instrumental artifacts that we can use to reduce our representations to practice. Thus far, new languages and tools for the Tiles exist only on the drawing board, but now that the hardware platform has been built I hope that progress on a rich software infrastructure will proceed apace.

The first step toward an accessible end-user environment is the development of a simple, sensible and consistent textual language for writing mobile programs. Our research group has accumulated a wealth of experience developing and teaching programming languages based on Logo, and a Logo-derived syntax and vocabulary are a natural starting point for this project. Logo encourages procedural decomposition and clarity, and has a very "low floor," meaning that children find it easy to get started programming in the language. The simple program listed in assembly at the end of the last section might become, in a Logo-derived idiom:

```
to start
  nose-light-on
```

```
  wait 1
  try-to-move
end

to try-to-move
  move
  if moved [
    start
  ]
  else [
    wait .5
    try-to-move
  ]
end

to clean-up
  nose-light-off
end
```

The program has changed a good deal relative to its assembly equivalent, although much of the basic structure remains. The most important differences are the syntax of the Logo language – this program is straightforward and easy to understand – and the fact that the "packaging" (start address, length, checksum and the like) has disappeared from the Logo code; such low-level details are taken care of by the compiler.

The concept of a "clean-up" routine remains, offering programmers a tool to deal with the subtleties of writing programs that can move from place to place. In this simple case, a light is turned on (the nose, or the light in "front" of the program). The programmer wants the light to stay on until the program has moved to another Tile. But there is a problem. After the program has moved it is no longer on the first Tile, so it will be unable to turn the light off.

There are several ways to deal with this: the programmer could turn the light off just before trying to move and on again just after, on the assumption that the program is so fast that the eye will be fooled; or the programmer could send the code back to the first Tile to turn the light off, and then back again to the second to resume operations; or the programmer could create a proxy process to return, turn the light off and die. All of these are valid ways to deal with certain situations. But the clean-up routine, as a simple piece of code, like any other except that it is guaranteed to be called as part of a successful move, is a useful, sensible feature that provides an intuitive avenue into thinking about these issues.

In addition to a long history developing text-based programming tools, our group has some, more recent, experience developing graphical programming languages. We are very interested in the possibilities for new kinds of representation, and for new tools that will help children build deep understandings, that graphical programming seems to promise. We have learned, however, that creating new visual representations that are as effective and rich as our more classic textual tools is extremely difficult.

I first became interested in graphical programming in the context of our Crickets and Programmable Bricks projects. After canvassing the current literature on visual languages – absorbing with much fascination the Self project's work in this area at Sun[20], Elizabeth Freeman's Map project at Yale[21], and other examples of the visual coding state-of-the-art – several of us worked together to design LogoBlocks, a drag-and-drop programming tool that eventually became part of the LEGO Mindstorms programmable brick product. During work on the LogoBlocks environment, I came to believe that graphical programming approaches are most powerful when they are designed for limited and specific functions, and that text-based languages remain better as general-purpose programming tools. Perhaps this is the result of the deep flexibility and descriptive qualities of language, and that textual programming idioms, though impoverished when compared with natural language, inherit some of this extraordinary malleability. Or perhaps I am wrong, and we just haven't figured out, yet, how to build efficient, coherent visual representations for general programming use.

In any case, I believe that there is much potential for graphical representations of certain kinds of Tiles programs, and would like to explore using the full spectrum of the desktop computer's media capabilities – sound, graphics, and animation, in addition to text – to paint a

---

[20] Chang. B, W. and Ungar, D. "Experiencing Self Objects: An Object-Based Artificial Reality" (1990) http://www.sunlabs.com/research/self/papers/experiencing-self-objects.html

[21] Freeman, E., et al. "Uniformity of Environment and Computation in MAP" (1996). For additional essays on related topics, see Burnett, M. et. al. *Visual Object-Oriented Programming: Concepts and Environments* (1995).

varied, compelling picture of the processes on the Tiles, and to allow users to "paint" processes of their own in many different ways. Different people absorb and appropriate ideas by different mechanisms, and one of the most exciting things about developing a new microworld is the chance to place a number of varied, complementary cognitive levers in children's hands, and to see which of these tools to think with are most useful, and in what contexts.

3.3 New Hardware

In addition to taking advantage of the desktop computer's multi-modal capabilities, it is also important build more interactive richness into the Tiles themselves. The stacked-board design, described in the section two, will make iterative development of the Tiles hardware relatively easy. No top boards have yet been built, but a number of designs, in various stages of both completion and practicality, do exist. The first one, which we will likely prototype in September, packs as much colorful and tactile input and output into four square inches as possible:

- 20 tricolor LEDs
- magnetic switch
- light sensor
- pushbutton
- analog dial
- 2-axis accelerometer

The LEDs, arranged in a matrix, visually dominate the Tile and provide a reasonably rich, flexible display. The magnetic switch and light sensor allow each Tile to function in accord with their environment, responding to shadows, flashlight beams, or strategically-placed magnets. The pushbutton and analog dial let programmers make adjustments, set values or trigger actions at "run-time." And the accelerometer provides a very tactile, tangible mode of interacting with the Tiles: I could write a program that "speeds up" if I shake a Tile, or a program that detects when any Tile is moved, in order to keep score in a puzzle game.

Taken together, these features complete the Tiles as platforms that integrate computation and physical form, bits and atoms. In Media Lab parlance, the Tiles are Things That Think. The ability to respond to environmental changes (like light), or to be adjusted independently of interaction with a desktop computer (by the turning of a dial or the waving of a "magic," magnetic wand), or to react to being shaken, makes the Tiles manipulable, independent, and fully part of the world around them.

Music is another important modality that is missing from the top board just described. We have talked quite a bit about the role that sound might play in the world of the Tiles, and we will certainly be building sound into future top boards. Some of Tod Machover's students, part of the Opera of The Future group at the Media Lab, have expressed an interest in working with the Tiles to design hardware and activities involving music and sound, and I'm looking forward to working with them and learning more about the creative possibilities of auditory media.

Finally, I would like to incorporate radio-frequency communication into some top-board designs. Having built a network with a clear, comprehensible, rectilinear topology, I would like to be able, on occasion, to violate that topology. Groups of Tiles, each with an RF-equipped node among them, could communicate at a distance; programs could hop from a "colony" in my office to one in the office next door, like ants crawling through a long tube between farms, or bees making the journey from hive to flower patch, and back.

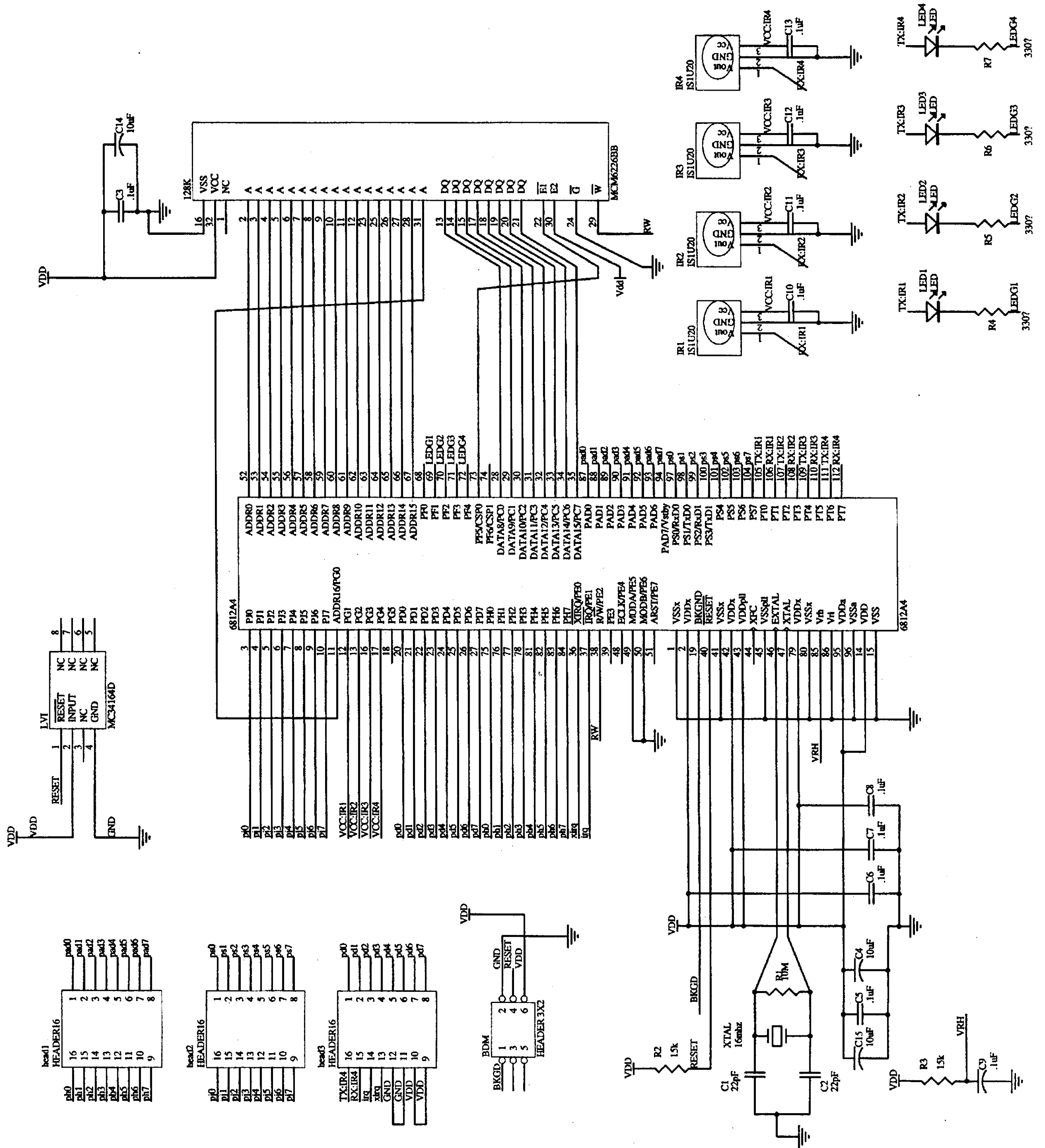
4 Conclusion

I think that the major initial goal of the Tiles project has been achieved: to build a platform that is powerful enough to be flexible, that admits of variation and experimentation and that is limited in its scope of possibility mainly by the number of hours we are able to devote to developing it further. There will not be enough time in the next year or so to address all of the ideas, additions, applications and explorations that have struck us, thus far, much less those that we have yet to stumble upon.

The core hardware is in place, and software at the proof-of-concept stages has been tested quite extensively. Much has been accomplished; I have spent the last year working mainly on the Tiles. But, in some important ways, nothing has been achieved yet. The Tiles are not ready for use by children in classrooms (they are not even particularly usable by other researchers at the Media Lab) and the rich input and output that has always been an integral part of plans for the Tiles remains to be built. We will continue to develop the Tiles, and significant progress in these areas will certainly be made over the next few months.

The most difficult, and the most rewarding, part of the Tiles project lies ahead – finding compelling, interesting, deep and useful vocabularies to use in thinking about networked programs. Drawing on the successes of projects such as StarLogo, and looking toward a future in which all computers talk to one another and many programs migrate across networks, we can work to develop representations of mobile code that find a place in our minds' eyes, in text and diagrams, on the computer screen, and finally in real-time effects and interactions across constellations of Tiles.

Appendix A: Schematic of Tile Circuit Board



References

Abelson, H and diSessa, A. *Turtle Geometry : The Computer As a Medium for Exploring Mathematics.* 1986. MIT Press.

Anderson, P. W. "More is Different" *Science*, 1972. Vol. 177, p 393-396.

Brown, J. S. *Idea Amplifiers: New Kinds of Electronic Learning Environments.* 1984. Palo Alto, CA: Xerox Palo Alto Research Center, Intelligent Systems Laboratory.

Burnett, M. et al. *Visual Object-Oriented Programming: Concepts and Environments* 1995. Manning Publications

Cairncross. F. *The Death of Distance: How the Communications Revolution Will Change Our Lives.* 1997. Harvard Business School Press.

Chang, B. W. and Ungar, D. "Experiencing Self Objects: An Object-Based Artificial Reality" 1990. The Self Papers. http://www.sunlabs.com/research/self/papers/experiencing-self-objects.html

Freeman, E. et al. "Uniformity of Environment and Computation in MAP" 1996 IEEE Symposium on Visual Languages, September 3-6, 1996.

Gould, S. J. *The Panda's Thumb; More Reflections in Natural History.* 1980. Norton.

Gray, R. "Agent Tcl: A transportable agent system" *Proceedings of the CIKM Workshop on Intelligent Information Agents, Fourth International Conference on Information and Knowledge Management (CIKM 95)*, Baltimore, Maryland, December 1995. http://www.cs.dartmouth.edu/~agent/papers/cikm95.ps.Z

Hylton, J. "Mobile Code Bibliography" http://www.cnri.reston.va.us/home/koe/bib/mobile-abs.bib.html

Kramer, K. and Borovoy, R. "Programmable Beads" http://el.www.media.mit.edu/projects/beads/ 1997. MIT Media Lab.

Kauffman, S. *The Origins of Order : Self-Organization and Selection in Evolution.* 1993. Oxford University Press.

Kiczales G., et. al. *The Art of the Metaobject Protocol.* MIT Press, 1991.

Lange, D. and Chang, D. "IBM Aglets Workbench: Programming Mobile Agents in Java, A White Paper" IBM, 1996. http://aglets.trl.ibm.co.jp/whitepaper.htm

Lugmayr, W. "Cetus Links -- Distributed Objects & Components: Mobile Agents" http://www.cetus-links.org/oo_mobile_agents.html

Martin, F. "Crickets, Tiny Computers for Big Ideas" http://fredm.www.media.mit.edu/people/fredm/projects/cricket/ MIT Media Lab.

Minar, N. "Computational Media for Mobile Agents" http://nelson.www.media.mit.edu/people/nelson/research/dc/ 1997. MIT Media Lab.

Papert, Seymour. *Mindstorms: Children, Computers, and Powerful Ideas.* 1980. HarperCollins.

Pea, Roy D. "Beyond Amplification: Using the Computer to Reorganize Mental Functioning," *Educational Psychologist.* 1985, Vol 20, No. 4, 167-182.

Resnick, M. *Turtles, Termites and Traffic Jams: Explorations in Massively Parallel Microworlds.* 1994. MIT Press.

Resnick, M., et al. “Pianos Not Stereos: Creating Computational Construction Kits” *Interactions*, 1996. vol. 3, no. 6, September/October. http://el.www/Papers/mres/pianos/pianos.html

Resnick, M., et. al. “Digital Manipulatives: New Toys to Think With” *Proceedings of the CHI ’98 Conference.* 1998. http://el.www.media.mit.edu/papers/mres/chi-98/digital-manip.html

Simon, H.A. "The computer as a laboratory for epistemology," in L. Burkholder (Ed.), *Philosophy and the Computer*. 1992. Westview Press.

Starr, P. “The Seductions of Sim” *American Prospect.* 1994, no. 17, pp 19-29

Tschudin, C. ed. *Mobile Object Systems : Towards the Programmable Internet.* 1996. Springer Verlag.