Designing Tangible Programming Languages for Classroom Use

Michael S. Horn
Tufts University
Department of Computer Science
161 College Ave, Medford, MA 02155
michael.horn@tufts.edu

Robert J.K. Jacob
Tufts University
Computer Science
161 College Ave, Medford, MA 02155
jacob@cs.tufts.edu

ABSTRACT
This paper describes a new technique for implementing educational programming languages using tangible interface technology. It emphasizes the use of inexpensive and durable parts with no embedded electronics or power supplies. Students create programs in offline settings—on their desks or on the floor—and use a portable scanning station to compile their *code*. We argue that languages created with this approach offer an appealing and practical alternative to text-based and visual languages for classroom use. In this paper we discuss the motivations for our project and describe the design and implementation of two tangible programming languages. We also describe an initial case study with children and outline future research goals.

Author Keywords
Tangible UIs, education, children, programming languages

ACM Classification Keywords
H5.2. Information interfaces and presentation (e.g., HCI): User Interfaces.

INTRODUCTION
Recent research involving tangible user interfaces (TUIs) has created exciting new opportunities for the productive use of technology in K–12 classrooms. One area that might benefit from the application of this technology is that of tangible programming languages for education. A tangible programming language is similar to a text-based or visual programming language. However, instead of using pictures and words on a computer screen, tangible languages use physical objects to represent various programming elements, commands, and flow-of-control structures. Students arrange and connect these objects to form physical constructions that describe computer programs.

Figure 1. A collection of tangible programming parts from the Quetzal language

By giving programming a physical form, we believe that tangible languages have the potential to ease the learning of complicated syntax, to improve the style and tone of student collaboration, and to make it easier for teachers to maintain a positive learning environment in the classroom. However, tangible interfaces are not without drawbacks. The technology involved is often delicate, expensive, and non-standard, causing substantial problems in classroom settings where cost is always a factor and technology that is not dependable tends to gather dust in the corner. Thus, in order to better explore potential benefits of tangible programming, we began with the development of tangible languages that are inexpensive, reliable, and practical for classroom use.

In this paper, we describe the design and implementation of two tangible languages for middle school and late elementary school children: *Quetzal* (pronounced *ket-zal'*), a language for controlling LEGO Mindstorms™ robots, and *Tern*, a language for controlling virtual robots on a computer screen. In our design, we emphasize the use of inexpensive and durable parts with no embedded electronics or power supplies. Students create programs in offline settings—on their desks or on the floor—and use a

portable scanning station to compile their *code*. Because it is no longer necessary for teams of children to crowd around a desktop computer, collaboration between children is less constrained and less formal. Code snippets and subroutines become physical objects that can be passed around the room and shared between groups. Furthermore, because one compiler can be shared by several teams of children, teachers are able introduce programming concepts to entire classrooms of children even when there are a limited number of computers available.

It is important to note that tangible programming languages are not yet commercially available, and their use has been restricted almost entirely to laboratory and research settings. Thus, the advantages outlined above are hypothetical. Indeed, one of the primary goals of this project is to better understand how tangible languages might affect student learning in classroom environments compared to more conventional languages.

BACKGROUND

Related work

Several tangible programming projects influenced our work in this area. An early example of a tangible language is Suzuki and Kato's AlgoBlocks [8], in which interlocking aluminum blocks represent the commands of a language similar to Logo. More recently, McNerney developed Tangible Computation Bricks [6], LEGO blocks with embedded microprocessors. He also described several tangible programming languages that could be expressed with the bricks. In a similar project, Wyeth and Purchase of the University of Queensland created a language for younger children (ages four to eight) also using stackable LEGO-like blocks to describe simple programs [10]. Zuckerman and Resnick's System Blocks project [11] provides an interface for simulating dynamic systems. Wood blocks with embedded electronics express six simple behaviors in a system. By wiring combinations of the blocks together, children can experiment with concepts such as feedback loops through real-time interaction provided by the blocks. Blackwell, Hague, and Greaves at the University of Cambridge developed Media Cubes [1], tangible programming elements for controlling consumer devices. Media Cubes are blocks with bidirectional, infra-red communication capabilities. Induction coils embedded in the cubes also allow for the detection of adjacency with other cubes. Finally, Scratch is an educational language being developed by the Lifelong Kindergarten Group at the MIT Media Lab [5]. While not a tangible language, Scratch uses a building-block metaphor, in which students build programs by connecting graphical blocks that look like pieces of a jigsaw puzzle.

In these examples, the blocks that make up the various tangible programming languages all contain some form of electronic components. When connected, the blocks form structures that are more than just abstract representations of algorithms. They form working, specialized computers that can execute algorithms through the sequential interaction of the blocks. Our model differs from these languages in that programs are purely symbolic representations of algorithms—much in the way that Java or C++ programs are only collections of text files. An additional piece of technology, a compiler, must be used to translate the abstract representations of a program into a machine language that will be executed on some computer system. This approach cuts cost, increases reliability, and allows greater freedom in the design of the physical components of the language.

Reality-Based Interaction

Tangible programming languages exhibit two fundamental principles of the *reality-based interaction* framework described by Jacob [4]. First, interaction takes place *in* the real world. That is, students no longer program behind large computer monitors where they have easy access to distractions such as games, IM, and the Web. Instead they program in more natural classroom settings such as on their desks or on the floor. Ideally, this gives teachers more flexibility to determine the structure and timing of in-class programming activities. It may also allow students to more easily transition between computer and non-computer work.

Second, interaction behaves more *like* the real world. That is, tangible languages take advantage of students' knowledge of the everyday, non-computer world to express and enforce language syntax. For example, Tern parts are shaped like jigsaw puzzle pieces. This provides a physical constraint system that prevents many invalid language constructions from being assembled as physical constructions. Furthermore, the metaphor of the jigsaw puzzle provides culturally-specific hints which imply syntax. In other words, the form of the parts suggests that they are to be connected in a particular way.

LANGUAGE OVERVIEW

Quetzal

Quetzal is a programming language for controlling the LEGO Mindstorms™ RCX brick. It consists of interlocking plastic tiles that represent flow-of-control structures, actions, and parameters. Statements in the language are connected together to form flow-of-control chains. Simple programs start with a Begin statement and end with a single End statement. For example, figure 2 shows a program that starts a motor, waits for three seconds, and then stops the motor. Programmers can add or change parameter values to adjust the wait time and the motor's power level. The order in which the statements are connected is important, but the overall shape of a program does not change its meaning. By inserting a Merge statement into the program, we can create an infinite loop. Here we don't need an End statement—the robot will execute this program until turned off. With Quetzal, loops in a program's flow-of-control form physical loops program structure. Using other statements, programmers can add conditional branches and concurrent tasks. Certain statements also accept parameter values

which can include constants and sensor readings. Parameter tokens are plastic tiles with specific shapes to represent their data type. These tiles can be inserted into slots in the top face of statements.

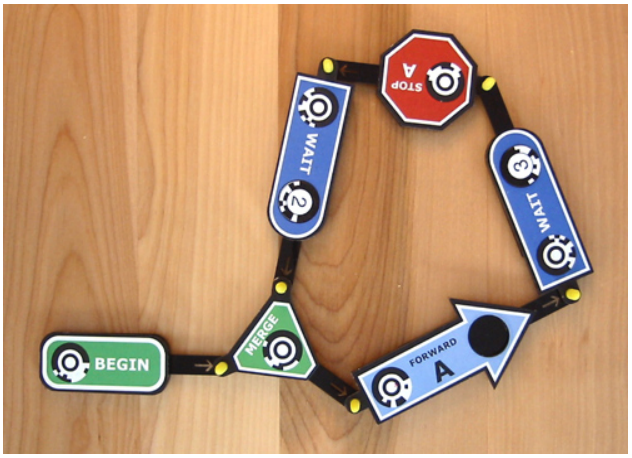

Figure 2. A merge statement creates a loop in the program.

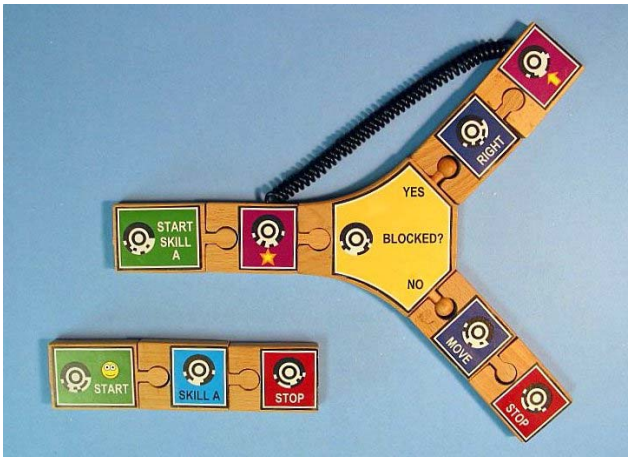

Figure 3. Tern programs may include conditional branches, loops, and subroutines

Tern Overview

The Tern language is based on the text-based programming language described in *Karel the Robot: A Gentle Introduction to the Art of Programming* [7]. With Tern, programmers connect wooden blocks shaped like jigsaw puzzle pieces to form flow-of-control chains. These programs control simple virtual robots in a grid world on a computer screen. Multiple robots can interact in the same world, and teams of students can collaborate to solve challenges such as collecting objects and navigating through a maze. A teacher might project the grid world on the wall of a classroom, so that all students can participate in one shared activity. Like Quetzal, Tern programs can include loops, branches, and parameter values. The Tern language also includes the ability to create subroutines called *skills*. Skills are defined using a special *Start Skill* block and can be invoked from anywhere in the flow-of-control chain. Figure 3 shows a sample program with a skill definition. Coiled wire connects special *Jump* and *Land* statements. These statements work in a way similar to a GOTO statement in a text-based language.

Tern was developed after Quetzal. In our initial evaluations with Quetzal, we found that children tended to spend more time building and playing with their LEGO creations than did programming them. While this is certainly not a bad thing, from a research perspective we are more interested in the programming aspect of the children's activities than the building aspects. Thus, one of our primary goals with Tern is to provide activities more focused around programming. Accordingly, the only way for children to control their on-screen robots is to write programs that tell them what to do. To enable robots to accomplish more sophisticated tasks, children must learn to write more sophisticated programs.

IMPLEMENTATION

The implementation of these languages uses a collection of image processing techniques to convert physical programs into machine code. Each statement in a language is imprinted with a circular symbol called a SpotCode [2, 3]. These codes allow the position, orientation, relative size, and type of each statement to be quickly determined from a digital image. Parameter tokens are also imprinted with similar visual codes. The image processing routines use an adaptive thresholding algorithm [9] and work under a variety of lighting conditions without the need for human calibration.

Our prototype uses a digital camera attached to a tablet or laptop PC. The camera has an image resolution set to 1600 x 1200 pixels. A programming surface approximately 3 feet wide by 2 feet high can be reliably compiled as long as the programming surface is white or light-colored. A Java application controls the flash, optical zoom, and image resolution. Captured images are transferred to the host computer through a USB connection and saved as JPEG images on the file system. With this image, the compiler converts a program directly into virtual machine code (in the case of Tern) or into an intermediate text-based language such as NQC (http://bricxcc.sourceforge.net/nqc) in the case of Quetzal. Students initiate a compilation by pressing an arcade button on the scanning station. The entire process takes only a few seconds, and, with Quetzal, programs are automatically downloaded to a LEGO computer. Any error messages are reported to the user. Error messages include a picture of the original program with an arrow pointing to the statements that caused the problem. With Tern there are *no* language syntax errors. The only possible errors are due system problems such as the camera being disconnected.

INITIAL EVALUATION

We conducted an initial evaluation with nine first and second grade children in a week-long day camp called "Dinosaurs and Robots" conducted at the Eliot-Pearson

School at Tufts University. The purpose of this investigation was to iron out any usability problems and get a basic sense for how students would react to physical programming. As part of the camp, the children used a Quetzal prototype to program robots that they had constructed. This investigation provided encouraging evidence that Quetzal can be viable and appropriate language for use with children in educational environments. For example, all of the children were easily able to construct and flow-of-control chains and read the sequence of actions out loud when asked. While not all of the children were able to understand the effects their programs would have on their robots, some were able to make predictions and correctly identify *bugs* in their code. After initial instruction, the children were able to build programs without direct adult help. There were also several examples of ad hoc collaboration between the children.

Figure 4. A student constructs a program with Quetzal during a week-long day camp on dinosaurs and robots.

NEXT STEPS

Our work with tangible programming languages is ongoing. We would like to expand the capabilities of the languages, improve the existing prototypes, and conduct more formal evaluations of their effectiveness in classroom settings. Future evaluations will be conducted with late elementary and middle school students. After our experience with first and second graders, we feel that programming activities will be more developmentally appropriate for older children. We also believe that it is important to conduct these evaluations in real-life educational settings such as after school programs or classrooms.

CONCLUSION

In this paper we described the design and implementation of two tangible programming languages for use in educational settings. Unlike many other tangible programming languages, our languages consist of parts with no embedded electronics or power supplies. Instead of real-time interaction, our languages are compiled using a portable scanning station and reliable computer vision technology. This allows us to create durable and inexpensive parts for practical classroom use. We described an initial usability session and also outlined future directions in our research.

ACKNOWLEDGEMENTS

We thank the Tufts University Center for Children (TUCC) and the University College of Citizenship and Public Service (UCCPS) for their generous financial support. We acknowledge the Center for Engineering Education Outreach (CEEO) at Tufts University for materials used in this project. Kevin Joseph Staszowski was the principal Investigator for the Dinosaurs and Robots project. Finally, we thank the National Science Foundation for support of this research (NSF Grant No. IIS-0414389). Any opinions, findings, and conclusions or recommendations expressed in this article are those of the authors and do not necessarily reflect the views of the National Science Foundation.

REFERENCES

1. Blackwell, A.F. and Hague, R. Autohan: An architecture for programming in the home. In Proc. IEEE Symposia on Human-Centric Computing Languages and Environments 2001, pp 150-157.
2. de Ipina, D.L., Mendonca, P.R.S. and Hopper, A. TRIP: A low-cost vision-based location system for ubiquitous computing. Personal and Ubiquitous Computing, 6 (2002), pp 206–219.
3. High Energy Magic. http://www.highenergymagic.com
4. R.J.K. Jacob. "CHI 2006 Workshop Proceedings: What is the Next Generation of Human-Computer Interaction?," Technical Report 2006-3, Dept. of Computer Science, Tufts University, Medford, Mass. (2006)
5. Maloney, J., Burd, L., Kafai, Y., Rusk, N., Silverman, B., and Resnick, M. Scratch: a sneak preview. In Proc. *Second International Conference on Creating, Connecting, and Collaborating through Computing C5 '04*. IEEE (2004), pp 104-109.
6. McNerney, T.S. From turtles to Tangible Programming Bricks: explorations in physical language design. Personal Ubiquitous Computing, 8(5), Springer-Verlag (2004), pp 326–337.
7. Pattis, R.E., Roberts J., Stehlik, M. *Karel the Robot: a Gentle Introduction to the Art of Programming, 2nd edition*. John Wiley and Sons, Inc. 1995.
8. Suzuki, H. and Kato, H. Interaction-level support for collaborative learning: Algoblock–an open programming language. In Proc. *CSCL '95*, Lawrence Erlbaum (1995).
9. Wellner, P.D. Adaptive thresholding for the DigitalDesk. Technical Report EPC-93-110, EuroPARC (1993).
10. Wyeth, P. and Purchase, H.C. Tangible programming elements for young children. In Proc. *CHI'02 extended abstracts*, ACM Press (2002), pp 774–775.
11. Zuckerman, O. and Resnick, M. A physical interface for system dynamics simulation. In Proc. *CHI '03 extended abstracts*, ACM Press (2003), pp 810-811.