Program and Algorithm Visualization in Engineering and Physics

Michael P. Bruce-Lockhart, Theodore S. Norvell
Memorial University of Newfoundland
Yianis Cotronis
University of Athens

mpbl@engr.mun.ca

Abstract

We report here on our experiences using a program animation tool, the Teaching Machine, for program and algorithm visualization for engineering and physics students at two universities. and the University of Athens, where it was adopted in 2005 to teach Physics students.

1 Introduction

We report here on our experiences using the Teaching Machine (TM) for program and algorithm visualization for Engineering and Physics students at two different sites: Memorial University, where it has been used since 1999 for teaching engineering students, and the University of Athens, where it was adopted in 2005 to teach Physics students.

The outline of the paper is as follows: In Section 2 we discuss the problem of what should be modelled in a program animation system. Section 3 is a short introduction to the TM. Sections 4 and 5 are respectively experience reports from Memorial University and the University of Athens. Section 6 presents a summary.

2 The Modeling Problem

Given a system T, Norman (1983) defined $M(T)$ as the mental model held by particular user of that system and $C(T)$ as a conceptual model, a "tool for the understanding or teaching of" T. Norman makes it clear T represents a physical system (in his paper a calculator). Yehezkel et al. (2004), in introducing EasyCPU, pointed out that if one replaces the actual system with a learning model $L(T)$, the $M(T)$ that a student arrives at may be different from the one arrived at if they had interacted with the original system. In as much as EasyCPU represents an Intel 80x86, T is still a physical system. In teaching early programming courses, what is the T of which we desire students to develop an effective mental model?

In developing the TM (Norvell and Bruce-Lockhart, 2000, 2004), we thought about that problem a lot. Working with weaker students in the lab, we found a lot of superstitious behaviour: they would throw lines of code at a problem as if they were spells, with little understanding of what they meant. We found ourselves preaching that every line of code had a specific meaning and purpose, that it was an instruction to a machine. Developing in the students an effective mental model of that machine goes a long way toward teaching them to reason about the code they write. That machine, T, we were programming (and which we wanted the students to understand) was not really a computer, at least in the conventional sense. Consider the following simple C++ code:

```
int x = 5; int y = 12; int z; z = y/5 + 3.1;
```

In the language of programming, we say, there are four *instructions to be executed*. Instructions to what and to be executed by what? T of course, but T is certainly not the hardware. The first three "instructions" are actually to the compiler. We view them as request for allocation of memory, in the stack if they are internal declarations, in the static store if external. The fourth is a minefield. There is a truncation and two automatic type conversions. If you really

want students to understand it you, need to be able to interpret the expression and see the conversions, *but these are normally inserted by the compiler.* CPU operations include fetching the value for `y` (whether in a register or memory), carrying out the various calculations, and writing the final value back to `z`.

We define T to be the abstract machine to which we are giving source-level instructions. That is, T is largely defined by the language; it is an abstraction combining aspects of the computer, the compiler, and the memory management scheme.

The TM was then designed as a means to show the students how this machine worked, so they could write programs to control it.

3 The Teaching Machine

The TM has been described elsewhere (Norvell and Bruce-Lockhart, 2000, 2004) so we will review it only very briefly. The TM interprets source programs in C++ or Java while displaying visualizations of the program and the abstract machine state: the current expression under evaluation, the memory (stack and heap), the symbol table, and a console for I/O.

Before the TM we were using debuggers to good effect. We believe, as do others (Cross et al., 2002), that debuggers are a powerful tool for visualization on they're own; so we built the TM on the metaphor of a debugger. Since we expected students to continue to use debuggers in the lab, it eases the burden on them of learning two tools.

As of 2006 the TM supports C++ and Java. For each language, the model consists of a compiler, an interpreter, and a model of the abstract machine's state. The compiler translates source code to a high-level machine code that is then interpreted. Using a high-level machine code for an abstract machine allows us to preserve such information as the tree-structure of expressions, the tree structure of the data, and the data-types of each data object.

The Teaching Machine reads in standard C++ or Java files, generally prepared using conventional tools. There are some restrictions on what it can handle, as neither compiler is a complete implementation. For example, we have implemented neither threads in Java nor templates in C++. Nevertheless, the subsets that are supported are large, standards conforming, and work well in the courses in which the TM has been used.

The source files are displayed in a source window where they may be stepped using standard debugger controls. There are separate windows for each type of memory store—in C++, static store, stack, and heap. There is a window that displays expressions, with buttons to allow them to be stepped-through separately. This display, called the Expression Engine, is similar to the Expression Evaluation Area of Jeliot 3 (Moreno et al., 2004). There is a simple display for console input and output. Finally there is a Linked View, which automatically generates graphical depictions of data structures, in a manner similar to that of the LJV tool (Hamer, 2004). All views evolve dynamically as the program is executed and all execution steps can be undone and replayed.

4 Experience at Memorial University

At Memorial University, the TM is used in a three course stream in Electrical and Computer Engineering: Structured Programming (ENGI-2420), Advanced Programming (ENGI 3891), and Data Structures (ENGI-4892). ENGI-2420 teaches the basics of programming to all Engineering students, while 3891 emphasizes the use of classes and introduces pointers and heap allocation. ENGI-4892 emphasizes recursion and linked structures such as linked lists and trees.

Use of the TM in 3891 evolved through two transitions:

- From 1999 to 2001, we continued to use our old lecture transparencies but moved to the TM to illustrate specific points of interest. The TM was used more than the debugger, since it could show constructs the debugger could not. We spent less time at the

Cohort	ENGI-3891 2001					ENGI-3891 2002					ENGI-3891 2004				
Response	5	4	3	2	1	5	4	3	2	1	5	4	3	2	1
2. (effectiveness vs. manual means)	31	49	12	4	2	48	26	7	2	0	59	35	4	2	0
3. (effectiveness vs. computer means)	10	65	16	2	2	33	45	10	2	0	51	37	4	0	0
4. (understanding of examples)	16	65	16	2	0	38	45	10	2	0	55	43	2	0	0
12. (understanding of examples on-line)	11	56	22	11	0	30	48	13	9	0	35	41	12	0	6

Table 1: Survey results. Results in percentage of respondents.

blackboard drawing and spent more time discussing examples with the students. Using the TM, however, drew us to different examples. Its use was subtly changing what we taught. We found our slides and examples rapidly getting out of sync.

- In 2002, in order to provide a fully integrated experience in the lecture and to give students access to interactive notes afterwards, we took advantage of the fact that the TM was a Java applet to integrate it directly into the notes, which were rewritten in HTML. We realized that we had to make it as easy as possible for instructors to author such interactive notes, so we created WebWriter++ (Bruce-Lockhart, 2001), a set of javascripts with a couple of additional applets. The smooth integration considerably increased the amount of lecture time spent running examples in the TM. ENGI 4892 moved to the same format the next academic year, with the Linked View being used predominantly.

Student surveys have been carried out from time to time. Table 1 shows the results for three consecutive cohorts of ENGI-3891 students. The 2001 cohort (51 surveys out of 55 students) got the TM in standalone mode with fewer examples. The 2002 cohort (42 of 63) got integration, but notes were developed on the fly. The 2003 cohort was actually surveyed at the end of 4982 in 2004, when separate surveys were given to the same group of students (26 of 44 responded of whom 23 had done 3891); because the results were so similar the two were combined. For each question, 3 represents a neutral response, with 5 and 4 being very and somewhat positive, and 1 and 2 being very and somewhat negative. The table reports percentages with missing values representing either 'not applicable' or no response at all. All three cohorts had either not seen the TM at all or had seen very limited use of it in their first programming course, ENGI-2420, so were able to compare its effectiveness vs. other means (questions 2 and 3). Question 4 asked about the effect of the TM on the student's understanding of the examples it was used on, while question 12 asked about effect on understanding for those students who ran examples themselves on-line (only 9 of 51 in 2001, when few examples were available on-line; 23 of 42 in 2002 and 34 of 49 in 2003/2004). As can be seen, the results become more positive as the integration with the course notes becomes tighter.

Teacher ratings were independently surveyed. Figure 1 shows the results for 3891 through the last five years, using the same instructor. 2003 represents the mature version of the course as currently taught. Long regarded by students as one of the most difficult in Electrical and Computer Engineering, it was allotted four hours teaching hours a week, instead of three. Student satisfaction rose sufficiently high that in 2004 the Faculty of Engineering decided to cut the course back to three hours a week. Approvals dropped back to historic levels, but the course was delivered using 25% fewer lectures, while covering the same material, with no effect on outcomes. With this as evidence, the TM was (and is) regarded by the Faculty as successful in improving the delivery of 3891. A decision to move it into ENGI-2420 in 2004 was made on that basis.

The impact of the TM in 2420 is hard to measure yet due to confounding factors such as a doubling of the section size to over 200 and a change to the curriculum. In trying to make his

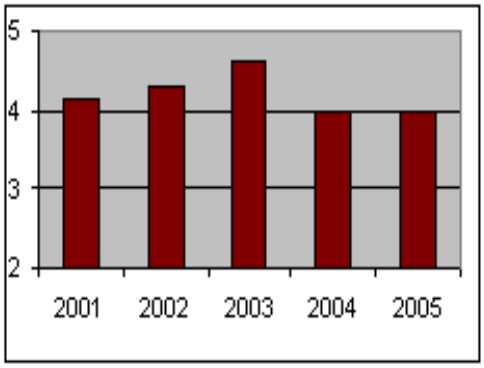

Figure 1: Quality of instructor ratings for 3891. 2=fair, 3=good, 4=very good, 5=excellent.

notes and lectures as understandable as possible in such a large venue, the instructor (Bruce-Lockhart) fell into a classic trap. The students got into the habit of letting the instructor do all the work. In the taxonomy developed at the 2002 Programming Visualization Workshop (Rößling et al., 2004) we collectively got stuck at stage 2: viewing. Although the TM had been designed to allow students to study examples on their own, and the students of 3891 and 4892 appear to be doing just that, the message hadn't been properly delivered to 2420 students. As an interim measure, the entire class was taken to the lab in sections and walked through fresh examples of pass-by-value vs. pass-by-reference. The objective was to get them to use the TM themselves and take them to the point where they could predict what the next step would produce (stage 3, responding). It is too early to tell what the real impact is yet, but numerous students approached the instructor afterwards to express approval of the technique and to suggest it be a regular feature of the course, but much earlier in the term.

5 Experience at Athens

At the University of Athens one of the authors has been teaching the subject of Principles of Programming Languages and Programming Techniques for the past six years, as part of the curriculum of a joint M.Sc. degree between the Departments of Physics and Informatics & Telecommunications. It is a well-known and highly respected conversion programme in which mostly Physics graduates from a number of Greek Universities enrol. The course is divided into three parts: the first and second parts discuss syntax (regular expressions, BNF) and computation aspects of programming languages, respectively; the third part discusses programming techniques (structured programming, top down program development).

The TM was introduced the fall 2005 semester augmenting teaching for the second and longest part, the computational aspects. Our teaching approach is analytical rather than empirical, aiming for exposing students to concepts recurring in programming languages, rather than teaching a specific language; students separately follow a two hour per week course on C. The teaching material for the past three years has been based on the six computation chapters of Part II of Fisher and Grodzinsky (1993), which fits well with our teaching approach. Students were given a number of small course-work exercises, which aim to practice and establish concepts.

Although students were satisfied with the course and appreciated the importance of it, they could not complete all exercises and they just "passed" the final written paper. A number of reasons contributed to this situation, the most significant being the difference between the cultures of Physics and Computer Science: most physics students consider programming "simple" when compared to laws, theories and models of the world. Consequently, as they are not easily convinced that a more complex programming world exists, they keep their misconceptions on how programs work.

We were convinced that a visualizing tool would help to clarify concepts and bring out their misconceptions, a tool which would visualize program execution above the level of a compiler. Having taught compilers for a number of years, we did not want a tool for visualizing compilers, but acting as the operational semantics for languages abstracting away compiler details. Searching the Web, we came across the TM, which proved to be the tool that we needed, at least regarding the abstraction level. The TM displays memory (static, stack, and heap), in which values of objects are depicted in decimal and binary, showing the address and space they actually use; and a stack implementation of a symbol table. TM has three levels of visualizing program execution (procedures, statements, and expressions) all with backtracking. It also has a Linked View (objects and pointers to them are actually depicted as graphs), which was really an unexpected and valuable feature.

We found that the TM did not impose at all on our teaching approach, but rather complemented it. We implemented tens of small programs demonstrating programming concepts, in the spirit of Fisher and Grodzinsky (1993). The TM could depict each point in focus. We were able to demonstrate principles, consequences, and some "not expected" behaviour. Let us mention a few notable examples. The (binary) representation of float and double was used to show that $(1.0/n)*n$ may not be 1 (a really surprising fact for physicists), or that we could "compute" $\sin(x)$ and get a value greater than 10^8. It was demonstrated that the concept most students had for one-to-one association of variable names to objects in memory is a misconception. We relied on the expression execution of the TM to demonstrate that expressions in C++ may be undefined and return strange results. We demonstrated the differences between parameter passing by (pointer) value and by reference; a misconception found not only among students but also in a number of programming books.

Most of the programs were demonstrated in class, but, as students had a copy of the TM, they could run their own programming experiments at home and very frequently came in class with questions.

We believe that most, if not all, learning styles as proposed by the Felder and Silverman (1988) may benefit from the use of TM. Active learners (AL) tend to retain and understand information best by doing something active with it; reflective learners (RL) prefer to think about it quietly first. AL may try their own programming "experiments" while RL have the opportunity to review the material shown in class using the TM.

Sensing learners tend to like learning facts while intuitive learners (IL) often prefer discovering possibilities and relationships. The TM only helps with IL but everybody is sensing sometimes and intuitive sometimes.

Visual learners (ViL) remember best by demonstrations. Clearly the Teaching Machine is strongly oriented toward ViL. Verbal learners (VeL) get more out of words–written and spoken explanations. Although VeL may benefit less, it is accepted that everyone learns more when information is presented both visually and verbally.

Sequential learners (SL) tend to gain understanding in linear steps, with each step following logically from the previous one; global learners (GL) tend to learn in large jumps, absorbing material almost randomly without seeing connections and then suddenly "getting it." Textbooks and classes essentially benefit SL as each teaching topic focuses on a specific aspect of programming. As the TM is the unique tool for demonstrating all programming topics, GL tend to benefit as well since the "whole picture" is there at any time.

6 Summary

The TM has proved to be useful in the delivery of a variety of courses. Its ability to handle both physical and abstract models has made it successful for helping engineering and physics students understand the abstract machine defined by the programming language. In one case it has helped reduce the resources required to teach a course. Its deep modeling of how a compiler and a processor interact allowed it to be used at Athens in ways the original designers

had not anticipated.

Does it help students develop effective mental models: $M(T)$ approximating the instructors $C(T)$? Anecdotal evidence supports this conclusion: one anonymous response to the Memorial surveys is telling. *"The Teaching Machine provides a great means of visualizing the internal mechanisms and operations of software. When I write code, I visualize how the Teaching Machine would translate it."*

Finally, we have come to realize that the TM is a solid platform for new developments. It is easy to use, robust, flexible, and reasonably complete. Its data model is easily extendable to more powerful, algorithmic visualizations. By providing it with specialized input and output plugins we can provide better visualizations and better interactivity; we expect that adding such plugins will be far easier than developing specialized visualizers from scratch.

References

Michael P. Bruce-Lockhart. WebWriter++: A small authoring aid for programming. In *Proceedings of the Newfoundland Electrical and Computer Engineering Conference*, St. John's, Newfoundland, 2001.

James H. Cross, T. Dean Hendrix, and Larry A. Barowski. Using the debugger as an integral part of teaching CS1. In *32nd ASEE/IEEE Frontiers in Education Conference*, pages F1G–1–F1G–6, 2002.

Richard M. Felder and Linda K. Silverman. Learning and teaching styles in engineering education. *Engineering Education*, 78(7):674–681, 1988. With preface http://www.ncsu.edu/felder-public/Papers/LS-1988.pdf (2002).

A. E. Fisher and F. S. Grodzinsky. *The Anatomy of Programming Languages*. Prentice-Hall, 1993.

John Hamer. A lightweight visualizer for Java. In *Proceedings of the Third Program Visualization Workshop, Research Report CS-RR-407, Department of Computer Science,University of Warwick*, pages 54–61, 2004.

Andrs Moreno, Niko Myllerand Erkki Sutinen, and Mordechai Ben-Ari. Visualizing programs with Jeliot 3. In *Advanced Visual Interfaces*, 2004.

Danald A. Norman. Some observations on mental models. In Dedre Gentner and Albert L. Stevens, editors, *Mental Models*, chapter 1, pages 7–14. Lawrence Erlbaum Associates, 1983.

Theodore S. Norvell and Michael P. Bruce-Lockhart. Taking the hood off the computer: Program animation with the Teaching Machine. In *Canadian Electrical and Computer Engineering Conference*, Halifax, Nova Scotia, May 2000. Available at http://www.engr.mun.ca/∼theo/Publications.

Theodore S. Norvell and Michael P. Bruce-Lockhart. Teaching computer programming with program animation. In *Canadian Conference on Computer and Software Engineering Education*, Calgary, Alberta, 2004. Available at http://www.engr.mun.ca/∼theo/Publications.

Guido Rößling, Felix Gliesche, Thomas Jajeh, and Thomas Widjaja. Enhanced expressiveness in scripting using AnimalScript 2. In *Proceedings of the Third Program Visualization Workshop, Research Report CS-RR-407, Department of Computer Science,University of Warwick*, pages 10–17, 2004.

Cecile Yehezkel, Mordechai Ben-Ari, and Tommy Dreyfus. Inside the computer: Visualization and mental models. In *Proceedings of the Third Program Visualization Workshop, Research Report CS-RR-407, Department of Computer Science,University of Warwick*, pages 82–85, 2004.