Mark Guzdial and
Elliot Soloway

Teaching the Nintendo Generation to Program

Preparing a new strategy for teaching introductory computer programming.

While the phrase "new economy" doesn't hold the same promise that it did for the dot-commers a couple years ago, we all still recognize that information technology has become the backbone of the world's economy. The National Science Foundation's Information Technology Research program was developed in response to the President's Information Technology Advisory Committee (PITAC) [6] report which said as much. Even in the down-turned economy, there are still huge numbers of IT jobs going unfilled.

Of course, none of this is news for those of us in the trenches teaching computer science. Our enrollments are bulging at the seams. We can't get enough faculty to cover the loads.

Despite those enormous enrollments, there are lots of indications that we're not doing enough, or not doing the right things, to meet the demand for a diverse, well-educated, large work force of computer science professionals. Start asking around—we're hearing about dropout/failure rates in CS1 courses in the 15%–30% range. A report in the ACM *SIGCSE Bulletin* from a working group at ITICSE 2001 found shockingly low performance on simple programming problems, even among second-year, college-level students at four schools in three different countries [5]. (These results echo our research from 15 years ago, so it's not clear we ever have figured out how to teach programming.) The American Association for University Women's report, *Tech-Savvy: Educating Girls in the Computer Age* [1] points out that large numbers of women drop out or simply fail to enroll in computer science courses because they perceive these courses to be overly technical, with little room for individual creativity

Why are we doing such a poor job at getting and keeping students in computer science? Here's our suggestion: Computer science educators are using an outdated view of computing and students. We teach computer science in much the same way as we learned it. We got excited about computer science while learning about laying out complex patterns of abstract decisions and computations, manipulating invisible data structures, and finally perhaps printing a number or a phrase. These activities matched the available computing hardware: The

TERRY MIURA

punched card, teletype, and generally text-focused I/O devices. Printing "Hello, World!" was interactive computing when most of us were learning to program. And we didn't get a lot of computer science majors in those days, either.

But, computing is much different today. The hardware enables us to do much more. In what follows, we propose a new strategy for teaching introductory computer programming we believe will attract a group of students not excited by the invisible, abstract, and text world we grew up with.

Engaging Students

Learning scientists have found over and over again that engaging the students is critical to deep learning. Sure, you can get students to memorize just about anything, but if you want them to understand it, you have to get them to think about it. Engaging students is critical for them to learn something well enough to use it again in a new situation.

College students today have been called the "Nintendo generation" or the "MTV generation." Their perception of technology and media has been profoundly influenced by these sources. The implication has often been that they need to consume mass quantities of fast-paced sound, graphics, and animation. Perhaps there's a more critical implication—that these are the kinds of media that Nintendo generation students want to produce when learning computer science.

Let's consider a popular textbook for CS1 today, Deitel and Deitel's *Java: How to Program* [2]. We're not picking on it but using it as an example. Most CS1 textbooks are fairly similar in terms of their exercises. The first program discussed in Deitel and Deitel is producing a line of text, akin to "Hello, World." The second places the text in a window. The next few produce numeric outputs in windows and then input numbers and generate calculator types of responses. Would one expect these kinds of exercises to be the ones to engage the MTV generation? Such exercises are exactly what the AAUW report describes as "tedious and dull."

It's About Media

Today's desktop computers were invented to be multimedia composition and exploration devices. The Xerox PARC Learning Research Group believed in a vision of the computer as a Dynabook: A tool for learning, through creation and exploration of a wide range of media. Pursuing that vision is what led them to invent the desktop user interface as part of their programming language, Smalltalk. Alan Kay and Adele Goldberg spelled out their vision of the Dynabook in a 1977 article, "Personal Dynamic Media," that talked about students building music, animation, and drawing systems—learning to program through the creation of media and learning to program in order to create media. Creating media sounds like what the Nintendo Generation is looking for.

In a lot of ways, that 1977 vision looks even more futuristic today than it did then. Kay and Goldberg describe animations, music synthesis, and drawing tools that resemble what we have today, but their media were created by students as they were learning to program. The established practice of having students focus on producing text and the occasional graphical user interface widget is well entrenched today. Who teaches CS1 by having students build animated horse races? Kay and Goldberg did 25 years ago. Ironically, there are fewer technical barriers to kids programming media today than there were in 1977. Computers today have magnitudes more zorch. High-resolution displays supporting millions of colors and sound cards with CD-quality recording and playback facilities are the default computing platforms offered at such stores as Best Buy and Radio Shack.

One reason for not introducing programming via multimedia construction is the lack of a good multimedia programming platform. Java is the most popular CS1 programming language today, but the Java 2 Media Framework is complex and isn't completely ported to all operating system platforms. Other popular CS1 languages like C++, Scheme, and Python offer little support for multimedia, and certainly not for all hardware and operating system configurations. This lack of support for multimedia might be due to the perception that multimedia programming is an advanced topic, something that CS1 students might one day aspire. No

one does multimedia first.

But, if the platform does support multimedia—as does Alan Kay et al.'s Squeak—multimedia programming can fit in well within the scope of a CS1 course. Indeed, the code for creating graphical transitions, for doing cell animations, even for synthesizing sounds is not all that complicated. We have been using the programming language Squeak for three years at Georgia Tech with over 100 students per semester [3]. Squeak (www.squeak.org) is a cross-platform multimedia environment that is the evolution of Smalltalk toward the Dynabook, championed by Alan Kay, Dan Ingalls, and a large open source community. Students at Georgia Tech use Squeak to build MPEG movie editors, personalized newspapers built on harvested Web content, math equation layout editors, and 3D adventure games. Admittedly, we're not yet in CS1—ours is a sophomore requirement. But it's through our use of Squeak and watching students rise to the challenge of multimedia programming that we came to the realization that multimedia-first is a viable way to introduce computing.

Kids can produce their kind of media using today's technology. In fact, they want to. And they'll learn programming doing it.

Example: Sound Synthesis

Last year, we ran a pilot class on computer music implementation for a dozen undergraduates so we could see what a multimedia-first approach might look like. The easiest way to start with computer music is to record oneself (Squeak provides a built-in, cross-platform digital recorder), save (name) the sound, and play it back—even at a different pitch, so the sound becomes an instrument. This is exactly the approach of modern sampling keyboards. The musical equivalent to "Hello, World!" in the domain of computer music is (in Squeak) (SampledSound soundNamed: 'mySound') play. To use our recorded sound as an instrument is a simple extension: ((SampledSound soundNamed: 'mySound') pitch: 'c') play.

We can go further down this path by using our sound to play recorded MIDI files. But let's take a different path to explore some (slightly) more complex algorithms. We'll use as an example the creation of sounds via additive synthesis. Additive synthesis is an old technique for sound synthesis (pre-dates Yamaha synthesizers) which doesn't generate musical sounds. It has the advantage, though, of being understandable and allowing the users to generate different kinds of sounds with not very much code.

Additive synthesis works by summing sine waves at different frequencies to create new kinds of sounds. The simplest way to do additive synthesis is the way that it was invented on early hardware: By stuffing numbers into a buffer and sending the buffer to a digital-to-audio converter. A Squeak routine (method) for generating a sound buffer with a given frequency, amplitude (roughly, volume), and duration requires no more than a simple loop and calculation. The problem requires some trigonometry, but nothing too complex.

forFreq: freq amplitude: amp duration: seconds
"Generate a monophonic SoundBuffer (array) filled with a sine wave of the given frequency (freq), maximum amplitude (amp), and duration in seconds (seconds)"

```
| sr anArray pi interval samplesPerCycle maxCycle rawSample |
sr := SoundPlayer samplingRate. "The Sampling Rate"
anArray:= SoundBuffer
    newMonoSampleCount:
    (sr * seconds). "The
    array for the sound"
pi := Float pi.   "The constant Pi"
interval := 1 / freq.      "Time
        between cycles,
        inverse of frequency: seconds per cycle"
samplesPerCycle := interval * sr.
        "secs/cycle * samples/second = samples per cycle"
maxCycle := 2 * pi.      "Maximum
        radians per cycle"

1 to: (sr * seconds ) do: [:sampleIndex |
    rawSample := ((sampleIndex/
        samples PerCycle) * maxCycle) sin. "Compute
        a sound sample value"
        anArray at: sampleIndex
```

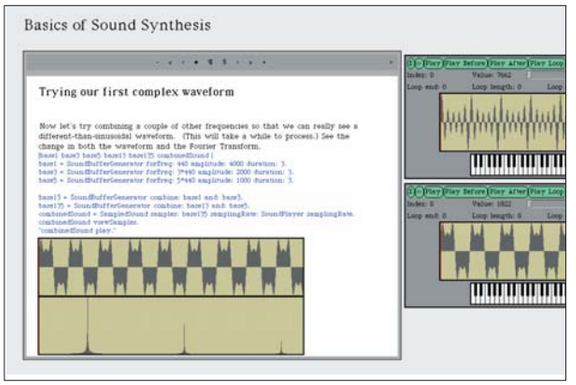
```
        put: (rawSample *
        amp ) rounded.
            "Insert the sam-
            ple into the sound at
            the right amplitude"
        ].
    ^ anArray
```

One doesn't need to understand anything about Squeak to see that this is 10 lines of code with a single "for" loop in it. To add these sine waves together one simply adds the values from the sound buffers with the same index values, like this:

```
combine: soundbuffer1 and:
soundbuffer2"Add two Sound-
Buffers  (arrays) together"

| newsound |
(soundbuffer1 size) = (sound
buffer2 size)
    ifFalse: [^self error: 'Sound
    buffers must be of the
    same length'].

newsound := SoundBuffer
newMonoSampleCount: (sound
    buffer1 size). "The resultant
    sound"

    1 to: (soundbuffer1 size) do:
[:index |
        "Add up each of the sam-
ples"
    newsound at: index put:
            (soundbuffer1 at: index)
+           (soundbuffer2 at: index)].
    ^newsound.
```

Six lines of code, and we now have an additive sound synthesizer by calling these methods with appropriate parameters. Based on our experience, we believe that Nintendo generation students will prefer learning about array manipulation where the example results in producing sound as opposed to sorting payroll IDs or doing linear searches for student names.

In Squeak, we cannot only generate sounds using these 16 lines of code, but we can also look at the waveforms, play the newly created sounds, and even do Fourier analyses on them. In our computer music course, we even used Squeak to create the lecture notes, where we build sounds, listen to them, and analyze them from within a Web browser (see the figure).



Reviewing computer music lecture notes from within a browser, using wave editors to explore the sounds.

Squeak has even more sophisticated multimedia aspects in its base distribution, like FM synthesis and wavelets support, so better (more musical, more sophisticated, more interesting) computer music is available, too. Thus, along the multimedia spectrum of computer music, it's possible to span the easy concepts of "Hello, World!" through introduction to programming up to serious programming—while engaging a set of students who are turned off by the text-first view of programming that we emphasize today.

Back to the Future

We have used "Hello, World!" for the past 25 years because text was the medium that was easiest to manipulate with the given technology. Today's technology can manipulate sound, graphics, and video with the same responsiveness and ease. Today's technology produces the media that "kids these days" are consuming. These same kids can produce their kind of media using today's technology. In fact, they want to. And they'll learn programming to do it.

We are sympathetic to the complexities of really making this "multimedia-first" approach work. We readily admit that neither the University of Michigan nor Georgia Institute of Technology are currently following a "multimedia-first" approach in our instantiations of CS1. But, if we want to attract and keep the MTV/Nintendo generation students sitting in our classes, we must reach out and use their media, use their modes of expression. Interestingly enough, in so doing we cannot

only teach all the "old concepts" but we can also have our students use modern ideas such as Fourier transforms. And, in using their media we are tacitly saying: we value you and your ideas. Students won't miss that gesture. Indeed, they will reciprocate and value more what we are trying to teach them. In so doing, we will provide alternative paths into computer science for students who might have turned away from the "Hello, World!" view of computing. C

References

1. AAUW. Tech-savvy: Educating girls in the new computer age. American Association of University Women Education Foundation, New York, 2000.
2. Deitel, H.M. and Deitel, P.J. *Java: How to Program*. Prentice-Hall, Upper Saddle River, NJ, 1999.
3. Guzdial, M. *Squeak: Object-Oriented Design with Multimedia Applications*. Prentice-Hall, Englewood, NJ, 2001.
4. Kay, A. and Goldberg, A. Personal dynamic media. *IEEE Computer* (Mar. 1977), 31–41.
5. McCracken, M., Almstrum, V., Diaz, D., Guzdial, M., Hagan, D., Kolikant, Y.B.-D., Laxer, C., Thomas, L., Utting, I., and Wilusz, T. A multinational, multiinstitutional study of assessment of programming skills of first-year CS students. *ACM SIGCSE Bulletin 33,* 4 (2001), 125–140.
6. PITAC. Information technology research: Investing in our future. President's Information Technology Advisory Committee, 1999; www.ccic.gov/ac/report.

Mark Guzdial (guzdial@cc.gatech.edu) is an associate professor in the College of Computing at the Georgia Institute of Technology.

Elliot Soloway (Soloway@umich.edu) is a professor in the College of Engineering, School of Information, and School of Education at the University of Michigan.