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 PCAST report which said as much. Even in the down-turned economy, there are still huge numbers of IT jobs that are 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 that enormous load, 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* workforce of Computer Science professionals. Start asking around—we’re hearing about drop-out/failure rates in CS1 courses in the 15-30% range. A soon-to-be-published report from a working group at ITICSE 2001 this last summer found shockingly low performance on simple programming problems, even among second year students, at four schools in three different countries. (These results echo work by Elliot some 15 years ago, so it’s not clear that we ever have figured out how to teach programming.) The American Association for University Women’s report last year, *Tech-Savvy:Educating Girls in the Computer Age* points out that large numbers of women drop out or simply fail to enroll in Computer Science courses because they perceive CS 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? Let us suggest that it’s because we’re using out-dated view of computing and students—or maybe it’s never really been best practice, just current practice.

ENGAGING THE 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 & Deitel’s *Java: How to Program*. We’re not picking on them but using them as an example. Most CS1 textbooks are fairly similar in terms of their exercises. The first program discussed in

Deitel & 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? Instead, these sound like the kinds of exercises that the AAUW report was describing: Overly technical with little room for personal creativity.

IT'S ABOUT MEDIA

Today's desktop computers were invented to be multimedia composition and exploration devices. That's what the Xerox PARC Learning Research Group had in mind when they invented the desktop user interface while inventing Smalltalk. Alan Kay and Adele Goldberg spelled out their vision in a 1977 paper *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. That sounds more like what the Nintendo Generation is looking for.

In a lot of ways, that 1977 vision looks more futuristic today than it did then. The established practice of focusing on text and the occasional graphical user interface widget is well-entrenched. There are fewer 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 Best Buy and Radio Shack.

Part of the problem is 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 to.

But when you can get the programming platform issue aside, you can find that multimedia programming isn't all that complicated. In fact, it can fit in well into the scope of a CS1 course. The code for creating graphical transitions, for doing cel animations, even for synthesizing sounds is not all that complicated.

Alan Kay, Dan Ingalls, and their team have continued the work that they started at Xerox PARC. The language that they are now developing is called Squeak (http://www.squeak.org). They are using Squeak to help students learn to program by producing multimedia. Squeak comes with support for 3-D computer graphics, CD-quality sound recording and playback, digital video and audio (MPEG), web browsing and serving, and Flash animations. It runs on over 20 operating system platforms bit-identically: You can not only transport the source code between platforms, you can transport the raw binary code for the VM.

EXAMPLE: SOUND SYNTHESIS

We could use as an example any kind of media in Squeak, but let's take computer music as one that we have some experience with (at a class at Georgia Tech). Squeak has lots of support for generating specific notes (`(FMSound pitch: 100 dur: 0.5 loudness: 0.8) play.` Or `(FMSound pitch: 'c3' dur: 0.5 loudness: 0.8) play.`) or with specific timbres or whole scales (`(FMSound lowMajorScaleOn: FMSound brass2) play`) so it's possible to do the musical equivalent of "Hello, World!" More important, though, is the support to build sounds from the digital ground up, and thus use the concepts and programming structures of CS1.

Let's take 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 very 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. In Squeak, you can synthesize sounds the way that sound synthesis was invented: 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 and duration looks like this:

```
forFreq: freq amplitude: amp duration: seconds
      "Generate a mono SoundBuffer filled with a sine wave of the
given frequency, maximum amplitude, and duration in seconds"

| sr anArray pi interval samplesPerCycle maxCycle rawSample |
      sr := SoundPlayer samplingRate.
      anArray := SoundBuffer newMonoSampleCount: (sr * seconds) .
      pi := Float 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.

      1 to: (sr * seconds ) do: [:sampleIndex |
            rawSample := ((sampleIndex / samplesPerCycle) *
maxCycle) sin.
            anArray at: sampleIndex put: (rawSample * amp )
rounded.].
      ^ anArray
```

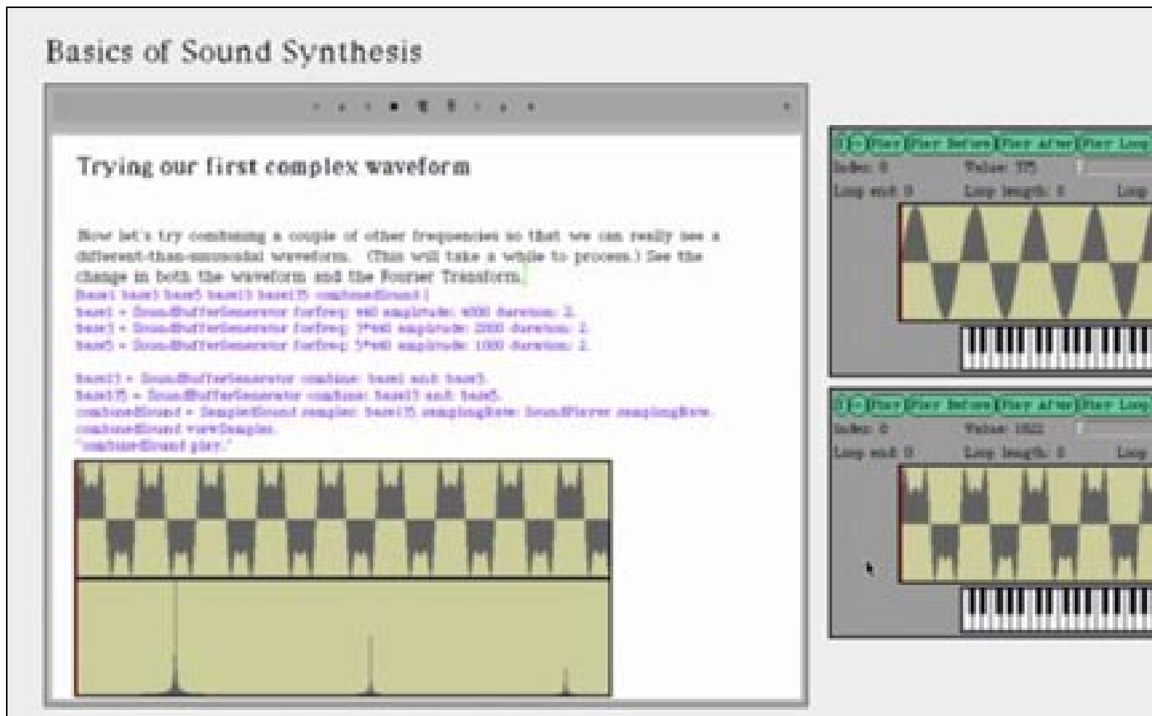
You don't to understand anything about Squeak to see that this is 10 lines of code with a single FOR loop in it. To add these sine waves together, you simply add the values from the sound buffers with the same index values, like this:

```
combine: soundbuffer1 and: soundbuffer2
      | newsound |
```

```
	(soundbuffer1 size) = (soundbuffer2 size)
		ifFalse: [^self error: 'Sound buffers must be of the
same length'].

	newsound := SoundBuffer newMonoSampleCount: (soundbuffer1
size).
	1 to: (soundbuffer1 size) do: [:index |
		"Add up each of the samples"
		newsound at: index put: (soundbuffer1 at: index) +
(soundbuffer2 at: index)].
	^newsound.
```

Six lines of code, and we now have an additive sound synthesizer. In Squeak, we can not 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. At Georgia Tech, we've started teaching a *Computer Music Implementation* course (for undergraduates) where we use Squeak to create even the course notes, where we can do all of these things from within a browser, running Squeak as a plugin (Figure 1).



Squeak has more advanced pieces, like FM Synthesis and Wavelets support, already implemented 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 might be turned off by the text-only view of programming that we introduce today.

BACK TO THE FUTURE

If we knew 25 years ago that we could teach programming with multimedia, and that the multimedia activities would engage students and would allow them to apply their personal creativity in their programs, why aren't we doing it today? Did we really believe that "Hello, World!" was capturing students' imaginations and drawing them in to Computer Science? Or was it simply the easiest path with the numbers of students we had and the kinds of machines that we had available?

We are sympathetic to the complexities of really making this work. We admit that at neither University of Michigan nor Georgia Institute of Technology are we currently following a "multimedia-first" approach in our instantiations of CS1. The path to get from here to there has lots of stones blocking the way. Nonetheless, we think it's worthwhile and that it's time to start.

The US economy drives the world's economy today, and the US economy runs to a large extent on information technology. There is an enormous need for large number of computer science professionals who are excited about the technology, who bring a diverse range of backgrounds and problem-solving skills to the table, and who sign up for and pass CS1. We have little choice: We must figure out how to engage this Nintendo Generation, to draw more of them to Computer Science, and to keep them there. We believe that shedding the text-only view of programs and embracing the rich media view of 25 years ago is a step in the right direction.