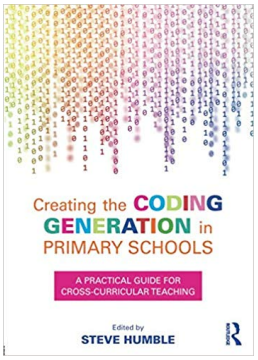
A chapter from the book, Creating the Coding Generation in Primary Schools: A Practical Guide to Cross-Curricular Teaching

Humble, S. (Ed.). (2017). *Creating the coding generation in primary schools: A practical guide for cross-curricular teaching*. Routledge.



Ten Considerations for Teaching Coding to Children
By Gary S. Stager, PhD and Sylvia Martinez

There is a "back to the future" quality to the newfound interest in children coding. When microcomputers began entering classrooms thirty-five years ago, programming is what kids and teachers did with them. As the software industry developed, software designed for education emerged. That software was primarily games designed to teach through constant testing of existing knowledge through flashcard-style drill and practice, while a few titles were more constructivist in nature. Things shifted dramatically in the late 1980s when seemingly overnight the rhetoric around computers in education shifted to using Office applications so that a generation of fifth graders would develop terrific secretarial skills. The "one computer classroom" became popular in the early 1990s. Now the teacher was the star of the classroom theatre and the computer was her prop for engaging children in specific activities carefully designed to teach a specific concept.

Beginning in the late 1990s, the five-paragraph essay migrated to the computer screen in the form of "digital storytelling." Kid-Pix-like graphics, simple animation, PowerPoint presentations, and digital video editing was perhaps more expressive than memorizing the menus in Microsoft Word, but no more technically sophisticated. Technology standards authored by organizations such as the International Society of Technology in Education did not mention programming at all and emphasized digital citizenship, communication, and other generic skills better suited for a Computer Appreciation course, rather than computer science.

The 21st Century has seen microcomputers mostly used for test preparation or standardized testing with the occasional app thrown in to justify tablet purchases or present an illusion of modernity.

This oversimplification of recent history suggests not only different views of educational technology, but also a shift in agency from the learner, to the teacher, and ultimately to the system. In too many schools, the very same computer was once used as an intellectual laboratory and vehicle for self-expression, (Kohl, 2012; Gary Stager, 2003) has been reduced to a tool of delivery, compliance, and surveillance.

The good news is that programming and computer science are back in vogue in the guise of coding. This presents reason for optimism and opportunities for those of us who have been advocating for all children to develop programming fluency across the curriculum for several decades. The current interest in coding is likely rooted in three phenomena.

1. Economic insecurity addressed by politicians and business leaders calling for better S.T.E.M. education in order to create "college- and career-ready" students (USDOE 2010).
2. The embarrassing state of computer science education in the United States, most notably in the low female and minority participation rates in the AP Computer Science exam (Erickson 2015).

3. The emergence of a maker movement combining digital fabrication, physical computing, programming, and timeless craft traditions.

Reintroducing computer science into the intellectual diet of children via the maker movement emphasizes individual creativity, authentic problem solving, and a multitude of contexts for programming through a variety of projects appealing to a diverse population of learners. Such a learner-centered approach to coding empowers students through modern meaningful experiences. The result may even address the first two concerns.

Teachers eager to embrace the promise of coding should seek inspiration from educators with a demonstrated track record of teaching kids to program based on solid progressive traditions, rather than the commercial hucksters, app makers, testing conglomerates, and professional associations who presided over the near extinction of computer science in schools.

Effective computer science education for primary students requires implementation strategies and curricula that move beyond empty rhetoric, mindless cheerleading, or kneejerk criticism. We cannot afford to put the assessment cart before the learning horse and allow those with little or no understanding of computing to set the standards. Therefore, realizing the promise of computer science for all children is predicated on the following ten observations.

#1 Computer science is the new liberal art

We are horrified by the energy, time, and opportunities wasted by two decades worth of adults arguing against providing each child with the opportunity to learn to program computers. The vehemence with which this case has been argued has been remarkable and destructive. One would be hard-pressed to find a similar example of such anti-intellectualism as, "Not everyone needs to learn to program…" when talking about teaching students to read, write, or learn science. Yet curriculum, policy, and classroom practice have been denatured by such ignorance and arrogance. There are plenty of questionable things taught to children, but no one ever proclaims that not every child should haiku or take Algebra II. Why would you even imagine not teaching every kid to program?

While there may indeed vocational benefits of learning to program, the real reason we should teach all children to program is because it gives them agency over an increasingly complex and technologically sophisticated world(Stager, 2014) . It answers the question Seymour Papert began asking a half century ago, "Does the computer program the child, or the child program the computer." This is a fundamental question of democracy that should concern every parent and educator.

Computer science is the new liberal art. It is also a legitimate science that students need to learn. However, unlike other branches of science, CS is beneficial in every other discipline. Any vocational pursuit in the arts, sciences, or humanities requires control over the computer. There may also be no more profound way than programming to develop habits of mind like persistence, perspective, or causality. Boys and girls need to experience the joy, creativity, and satisfaction associated with making something out of nothing or bending the computer to one's will. If we are to have computers in education, then learning to program grants the greatest return on investment.

#2 We have done this before

While the use of the term "coding" as a substitute for "programming" may be new, we have taught lots of children and their teachers to program all over the world in the recent past. Such awareness should inspire and support future efforts.

Hobbyist programming flourished prior to the dawn of the maker movement or Computer Science for All (CS4All) efforts. David Ahl, Editor of Creative Computing Magazine, told Gary Stager that his publication reached a subscriber base of 400,000 in 1984. In the mid-1980s, long before computers were ubiquitous in schools, Dan Watt sold more than 100,000 copies of his book, *Learning with Logo*, largely to school teachers interested in teaching programming to children. Beginning in 1989, Australian schools pioneered 1:1 computing when the first schools anywhere outfitted every child with a personal laptop computer. The purpose of those laptops was for children to program across the curriculum, make learning more personal, and realize the progressive visions of John Dewey, John Holt, and Seymour Papert. (Grasso & Fallshaw, 1993; Johnstone, 2003; GS Stager, 1998; Gary Stager, 2006; G. S. Stager, 1995)

We have also seen how elitist and theoretical approaches to computer science, as demonstrated by the Advanced Placement examination system can turn droves of students, especially women and minorities, away from computing while leaving most teachers with the impression that their academic subjects have nothing to do with it. (Erickson 2015, Herold 2014)

This has happened before. In the 1980's there was intense interest in computer science corresponding to the rise of personal computing. Because there were no additional spaces for university computer science majors, introductory courses got harder to weed out the rise in the number of students who were "harder to teach", which really meant those with less experience in high school. (Guzdial 2014) Today, as popularity again drives renewed interest in computer science, the limited number of teachers and slots in computer science classes raise the bar, making it even less possible for beginners to find a place to learn. We must find ways to reach all students with programming activities that are interesting and fun, opening doors for all kinds of students, even ones who are interested in programming not to become computer scientists or engineers, but want to use the limitless power of the computer to explore areas of their own interests.

Arthur Luehrmann coined the term computing literacy in the early 1970s. The term later morphed into computer literacy with a focus on the machine, rather than on process. Although many schools have eliminated programming from their computer literacy offerings, Luehrmann was quite specific in defining the term.

"Computer literacy must mean the ability to do something constructive with a computer, and not merely a general awareness of facts one is told about computers. A computer literate person can read and write a computer program, can select and operate software written by others, and knows from personal experience the possibilities and limitations of the computer." (A. Luehrmann, 1980)

Even if our primary goals include awareness of the computer, digital citizenship, or basic computer operation, these concepts are best developed in the meaningful context learning to code.

Educators in the past have demonstrated how to create compelling programming activities for children that offer teachers evidence of curricular relevance. In other words, we *know* how to make programming "hard fun" (Berry & Wintle, 2009; Seymour Papert, 1993; S Papert, 2002) for children while revealing the educational benefits teachers seek. The Logo literature is filled with such ideas.

#3 Learning to program takes more than an hour

Advocacy groups, such as Code.org, have done an impressive job of raising the visibility of coding as an important skill young people should acquire, even if their motives are vocational or commercial in nature. In addition to lobbying efforts, their annual Hour of Code campaign has captured the attention of politicians, the media, and school leaders.

Recognizing that the existing school curriculum is morbidly obese, Hour of Code says to educators, "C'mon you can find an hour once a year to expose kids to coding." To support this effort, web sites like code.org offer countless little coding games and puzzles that kids can complete in a matter of minutes without the participation of a teacher, who is assumed to know little or nothing about computer programming.

The success of Hour of Code in raising the visibility of coding must be tempered by the schools who congratulate themselves for literally doing the least they can to introduce children to computing. It is too easy to use Hour of Code participation as a substitute for action. Quality work takes time!

#4 Fluency is the goal

Exposure may be a worthy first step as long as substantive plans are clearly outlined for truly learning to code. Exposure itself is insufficient. Fluency must be the goal. We want children to be able to create, invent, dance, sing, write, and debate ideas via programming. Such fluency requires the same levels of dedication necessary to become good at any other time-honored pursuit. (Cavallo 2000)

"Technological fluency will be valued far less as something needed for the workplace than as a language in which powerful ideas can be expressed." (Seymour Papert, 1998)

Websites like Code.org introduce needless confusion with a smorgasbord of coding options using many different programming languages each with quite specific syntax. Educators report how we "do a little Scratch, then we do some Java, then some Codestudio…" This may be novel, but has little educational value. Sampling programming languages is as foolish as offering a few hours of instruction in multiple foreign languages or reading one page each of a dozen works of great literature. Well-meaning teachers new to coding need guidance in selecting a language appropriate to their student population and programming goals. Then they need to stick with it long enough for some kids to become sufficiently proficient to explore their own ideas and share expertise with their peers.

Jumping around different programming languages may appear sophisticated, but is merely an exercise in false complexity (Squires & McDougall, 1994; Squires & Preece, 1996) with little benefit to learners.

There are programming languages designed with learning in mind (Logo, MicroWorlds, Scratch, Snap!, Turtle Art) and others with a more vocational focus (Python, C++, Java, Processing). Learning to code in any language is better than not coding at all. That said, if we wish to democratize coding and have all students code, it is important to use a language designed for learning. (Greenberg, 1991; Harvey, 1982,

1993, 2003; Seymour Papert, 1999) Such languages have accessible interfaces, consistent syntax, helpful error messages, and contain objects to think with that encourage mathematical thinking and problem-solving skills, even when programming an interactive story or an animation for history class.

The extraordinary popularity of the Scratch programming language is based on its clever design, online library of more than fourteen million projects, and being free. It follows the old Logo edict of "low threshold" and with the addition of new dialects and increased functionality, also promises "no ceiling." (Seymour Papert & Watt, 1977) Scratch is neither a baby programming language or a stepping-stone to coding. It *is* a programming language. The block-based interface makes the process of coding more concrete for many learners. The ability to easily share projects, open the hood on those projects, remix them, or borrow the code for use in your own creations inspires lots of young programmers. Scratch is a descendent of Logo and embodies many of its Piagetian and Papertian traditions in its design.

The creation of Scratch dialect SNAP!, complete with recursion, first-class objects, and program as data creates a glide path for young coders to develop real computer science fluency. The fact that Snap! is used in the University of California at Berkeley's Beauty and Joy of Computing course ("Beauty and Joy of Computing," 2016) and as a pathway new Advanced Placement Computer Science Principles class by the same title, solidifies Scratch's status as a rich environment for learning to code and developing powerful ideas in computer science, mathematics, and other disciplines.

The Scratch dialect Beetle Blocks allows children to program in 3D and export those files to 3D printers. The "secret sauce" of Tickle, a version of Scratch for the iPad, is that in addition to Scratch functionality, the user can now program low-cost drones, robots, LEGO, and even home lighting systems. Being able to program your toys to dance, fly, and speak generates other exciting contexts for learning to code.

If fluency is the goal, students should perhaps spend a year or more learning with a single programming language.

#5 There is no computer science without computers

It would appear self-evident that one needs computer access to learn computer science. Computer coding requires a computer, right? Not necessarily if one is to believe the K-12 digital technologies policy statements prepared in the US, UK, and Australia. These frameworks, policies, and curricula place a greater emphasis on computational thinking than computing or coding. This focus on computational thinking assumes a lack of access to computers or teachers capable of teaching coding.

While we appreciate that computational thinking provides valuable opportunities for problem solving, critical thinking, and analysis, such skills would be enriched in the context of computer programming. Computational thinking without programming is just math. Since mathematical knowledge construction is more efficacious when situated in a programming context, we stand by the radical claim that computer programming with computers is better than without them. (Harel, 1991; Harel & Papert, 1990; Kafai & Resnick, 1996) We share Piaget's belief that knowledge is a consequence of experience and that the experience of programming computers is richer than learning *about* computer programming. (Ackermann, 2001; Duckworth, 1996; Kamii, 2000; Kamii & Joseph, 2004; Seymour Papert, 1988; Piaget & Piaget, 1973)

The popular *Exploring Computer Science* (Goode, Chapman, & Margolis, 2012; Goode & Margolis, 2011; Outlier Research and Evaluation, 2015) curriculum places a great deal of emphasis on social implications of computing, problem solving, and off-computer activities while *Computer Science Unplugged* (T. Bell, Alexander, Freeman, & Grimley, 2009; T. C. Bell, Witten, & Fellows, 1998; Fellows, Bell, & Witten, 2002; Henderson, 2008) does not use computers at all.

"The activities introduce students to Computational Thinking through concepts such as binary numbers, algorithms and data compression, separated from the distractions and technical details of having to use computers. Importantly, no programming is required to engage with these ideas!" ("CS Unplugged," 2016)

This statement from the Computer Science Unplugged website draws focus to several shortcomings of this approach.

1. The fallacy that you cannot enjoy dessert without eating your vegetables first.
2. The debatable idea that binary numbers, algorithms, or data compression are appropriate or important curricular topics for young people, especially if there will no actual coding involved.
3. The implicit suggestion that these topics are easier to learn without programming or are even relevant in the absence of coding.
4. When "distractions and technical details" are used as an excuse for coding without computers, the writer presents an incorrect notion of cognitive development. Introducing these topics without the context of computing makes them more abstract, not less.

Code.org commissioned a study to measure the effectiveness of its outreach and curriculum development efforts. High school students reported dissatisfaction with "off computer" or "unplugged" activities compared to the actual programming in their *Exploring Computer Science* classes.

"Students preferred the programming activities to the unplugged lessons.
Many teachers felt that the programming units, especially those that included emphasis on HTML and Scratch, were of the greatest interest to students. Teachers felt that programming allowed students to "just try things out, make mistakes, explore things." Teachers described how the opportunity to create things engaged students, with one teacher noting, "What got them all excited was being able to create in HTML, in Scratch, in making a robot do something."

… Several teachers remarked that students were frustrated because they were not doing what they thought of as "real" computer science. As one teacher said, "I heard more than a few times early on in the year, 'Isn't this a computer science class? Where are the computers?' That's part of the ECS shtick. You can kind of play it off and explain it away, but when it comes down to it I also agree with them.'" (Outlier Research and Evaluation, 2015)

The students also reported displeasure with waiting several units of study before computers were used. (Outlier Research and Evaluation, 2015)

You should avoid a computer science curriculum that delays actual programming for weeks or months while content is covered. Another sign of poorly conceived curricula is when there is a great emphasis on vocabulary memorization such as algorithm, binary numbers, or data compression at the expense of actual coding.

#6 The standards are at best premature
Educators, publishers, and policy-makers too often resort to the cliché, "We need to teach teachers how to assess this stuff." When the "stuff" is associated with new and powerful ideas like computer science, making, tinkering, or engineering, this desire becomes highly problematic.

Ask any decent fourth grade teacher to show you student writing samples representing below grade level, at grade level, or exemplary work and they can do so with ease and alacrity. Why? Because they have seen thousands of examples of student writing and there are decades worth of experience evaluating writing. This is not so for Scratch programming, Arduino projects, or digital fabrication.

Therefore, it is premature to create benchmarks for judging student work at a specific moment in time since we have no idea of what kids are capable of doing with these emerging technologies. A teacher's energies would be much better spent supporting her students in creating hundreds of unique and wondrous projects. Only then may we begin to assess such efforts.

This putting of the assessment cart before the learning horse results in the laundry lists of hierarchical skills dictated to teachers by anonymous committees of bureaucrats. Turn to any page in a K-12 digital technologies framework and read a few paragraphs. It will not take long to find empty rhetoric, preposterous sequences, low-level project suggestions, and mountains of nonsensical word salad that fails to drive progress. A lack of imagination and vision is often rooted in ignorance. Too often in education, such ignorance is cloaked in a few hundred pages of bullet points, charts, and graphs that no one bothers to read. This leaves teachers doubting themselves and without any real guidance about how to teach coding.

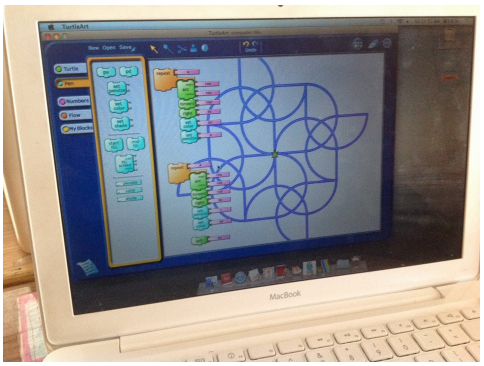
#7 Computer Science is a context for constructing mathematical knowledge
If your educational goals are no more ambitious than increasing understanding of the existing math curriculum, then you should teach children to program. After observing a 6th grade teacher lead a forty-five minute lesson on absolute value, she was asked, "When would you *use* absolute value?" She shrugged and answered, "Perhaps in 7th grade." Absolute value comes in quite handy when you're trying to land your rocketship on a planet in the video game you programmed or to teach a robot to navigate unfamiliar terrain.

Absolute value, probabilistic behavior, negative numbers, angle measure, rate of speed, coordinate geometry, Boolean logic, modulo arithmetic, and even inequality are confusing abstractions without the context of computer programming. In other words, much of the math curriculum has little, if any, application or relevance outside of coding.

While coding, these concepts have meaning and application. Learning them becomes more natural and students engage in the practice of mathematicians rather than being taught math. (Harel & Papert, 1990; Kafai, 1995; Seymour Papert, 1971, 1972a, 1972b, 1972c, 1980a, 1980b, 1993, 1997, 1999, 2000a, 2000b, 1991; S. Papert, Watt, D., diSessa, A., Weir, S., 1979; G. S. Stager, 1997)

Josh Burker, a teacher in a K-5 school in the United States created a lesson for grade four students who were studying Islamic tiling patterns as part of a world culture unit. The students used Turtle Art to create patterns following the geometry of Islamic tiles. A few years ago, this project would have ended

with this two dimensional representation. Perhaps the tiles could have been printed out or displayed as part of a back to school night PowerPoint slideshow. But instead, this project continued by importing these patterns into a 3D Computer Aided Design (CAD) software program and printing them out on a 3D printer. Pressed into firing clay and hand painted by students, these became a class project that could be shared with family and friends. This project represents a learning journey from culture, through geometry, into art, creating sharable artifacts that could not be created without programming.



Covering curriculum

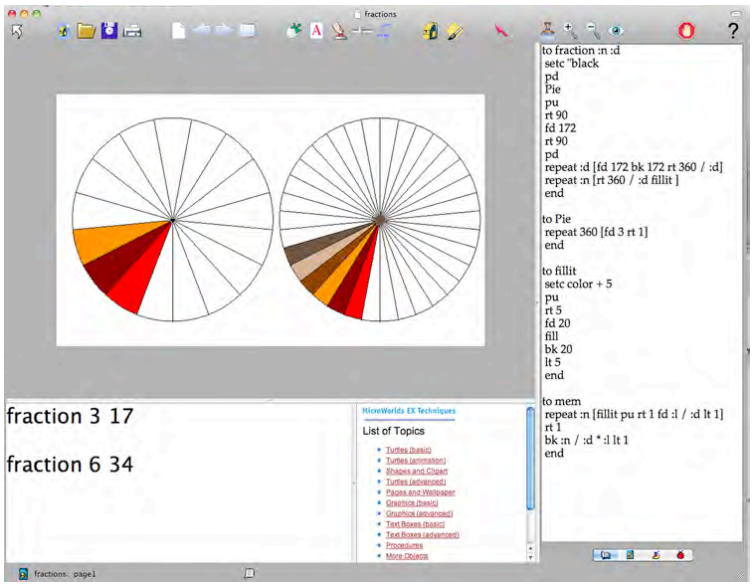
Most teachers are responsible for "covering" a vast list of content that makes up every subject. Although teachers have more flexibility in how curriculum is covered than some like to admit, they are indeed responsible for students learning specific things. Learning to program can lead to greater understanding of traditional topics too.

For example, take fractions (please)! Fractions are taught to kids all over the world, over multiple years and with great disparity in achievement. Fractions are one of the things we teach kids over and over again, yet they don't stay taught. Idit Harel-Caperton's American Educational Research Association award-winning research demonstrated that if you ask fourth graders to program a computer "game" to teach younger children fractions, the programmers gain a much deeper understanding of fractions, plus a host of other skills, than children who were taught fractions in a more traditional fashion (Harel, 1991).

Asking 10-year-olds to write a computer program that represents any fraction as a part of a circle leads to a greater understanding of fractions, as well as a working knowledge of variables, division, the geometry of a circle, and the knowledge that comes from design and debugging.

While some would claim that "we don't have time" to add programming to the curriculum, devoting several class periods to one programing project may be much more efficient than spending several years of instruction on the same topic.

A seventh grade girl completing an assignment to write a program to solve a linear equation will come to understand the math topic, perhaps better or quicker because she was able to add computer graphics or an animated story or musical composition to her program. Programming supports a range of expression and learning styles. Therefore, a willingness to engage with someone else's assignment may become more palatable when motivated to do so in your own voice.(Martinez & Stager, 2013)



Fraction program written by 4th graders in Logo using MicroWorlds (LCSI)

"Why then should computers in schools be confined to computing the sum of the squares of the first twenty odd numbers and similar so-called 'problem-solving' uses? Why not use them to produce some action? There is no better reason than the intellectual timidity of the computers in education movement, which seems remarkably reluctant to use the computers for any purpose that fails to look very much like something that has been taught in schools for the past centuries. This is all the more remarkable since the computerists are custodians of a momentous intellectual and technological revolution." (Seymour Papert & Solomon, 1971)

#8 Physical computing is a critical context for learning computer science

In our book, *Invent to Learn: Making, Tinkering, and Engineering in the Classroom,* we assert that we have entered a historical period of significant technological change. We identified three categories of game-changing technology; digital fabrication, physical computing, and computer programming. Physical computing may be thought of simply as robotics, but more broadly as adding interactivity and intelligence to a variety of materials. Microcontrollers, such as Arduino, the Hummingbird robotics kit, and even LEGO's WeDo or Mindstorms kits offer the ability to design things like interactive robots, prototype complex systems, and build instruments capable of conducting scientific experiments. Coding supercharges a range of projects that can exist on and off the computer screen.

Most physical computing projects require careful sequencing, timing, logic, and sensory feedback. Regardless of the materials being used, all sensors return a range of values based on external stimuli. Your invention then behaves in a particular way based on that sensor data.

The first thing one needs to do when working with a sensor is to determine the kind and range of data it provides. This requires writing a simple program to continuously display data as you change the conditions impacting the sensor. (Some programming environments have a "watcher" built-in for observing sensor behavior) Once you can "read" the data, you need to determine its behavior as it responds to the world. For example, as the light sensor sees more light, do the numbers it reports increase or decrease? Inverse relationships are not uncommon (a nice context for more mathematical thinking). Once you get a sense of the data, you need to determine the threshold at which your program

tells your machine to trigger a specific behavior. This almost always occurs when the data is within a particular range, relying on a working understanding of inequality more solid than trying to remember which way the crocodile's mouth is pointing on a worksheet.

The coding strategy just described is employed in countless scenarios from making your cardboard robot dog bark at an intruder to turning on an air-conditioner when a room gets too warm. Despite the importance and ubiquity of this skill, it cannot be found in any of the K-12 "coding" standards we have surveyed.

#9 Instrumental coding, additive teaching

Two fifth grade boys in Vermont were inspired by The Blue Man Group to build a marimba out of PVC pipe. Since they wished to compose music for the marimba and play it, tempering the pitches was important. They found an equation for determining the length of the tube, based on its diameter and the frequency of the pitch they were trying to reproduce. Then they found tables of musical pitches and their frequencies on the Web, but now some mathematics was required. The boys explained that they were capable of performing the calculations, but they wanted all of their classmates to be able to participate as well. This display of empathy or hubris led to an ingenious plan. The boys wrote a Scratch program that asked users for the diameter of their tube and the frequency of the pitch they wished to reproduce. It then told you how long to cut the tube.

This is a quite elegant example of what one might call instrumental programming. The result of coding wasn't a computer program or app, as much as it was a marimba. Programming to solve problems en-route to making something else, analyzing data, or conducting an experiment is a very powerful idea.

After the students shared their marimba with hundreds of adoring adults, it was suggested that since the program was about music, the user might wish to enter B Flat, instead of 466.16. Such a small intervention by a more experienced mentor is rooted in the knowledge that it is possible to modify the program in such a fashion and that *those* students were capable of doing so. It was well within their zone of proximal development.

Teaching coding like this, by scaffolding small but increasingly difficult additions to student projects, is quite different than just assigning standard problems of varying difficulty. Teachers learn to teach this way by watching and listening to students, not by following instructions in the teacher's manual. The confidence and competence to do this needs to be nurtured in teachers. Creating mentorships, critical friends networks, and ongoing professional development grounded in collaboration creates a space for teachers to gain these skills.

There's more to CS for kids than programming "video games"

There are all sorts of possible programming projects, systems to invent, and problems to solve. Sometimes, educators fall into the trap of believing that all kids love video games and that programming video games will appeal to boys and girls equally. A similar fallacy is the idea that all kids want to create apps because after all, that's how you get rich. In our experience, some kids are motivated by these projects while other attempts feel like pandering and result in unsatisfying versions of things kids otherwise enjoy. We also know that kids enjoy writing programs to graph linear equations, collect and analyze polling data, answer the question, "am I normal?," play a marimba, control a drone, generate random poetry, compose atonal music, bring a robot to life, or even do their homework. "This protean

ability to take different forms and, if you use it right, to become a kind of mirror in which you can see reflections of yourself" (Seymour Papert, 1985) makes learning to code personally valuable and useful in other aspects of life.

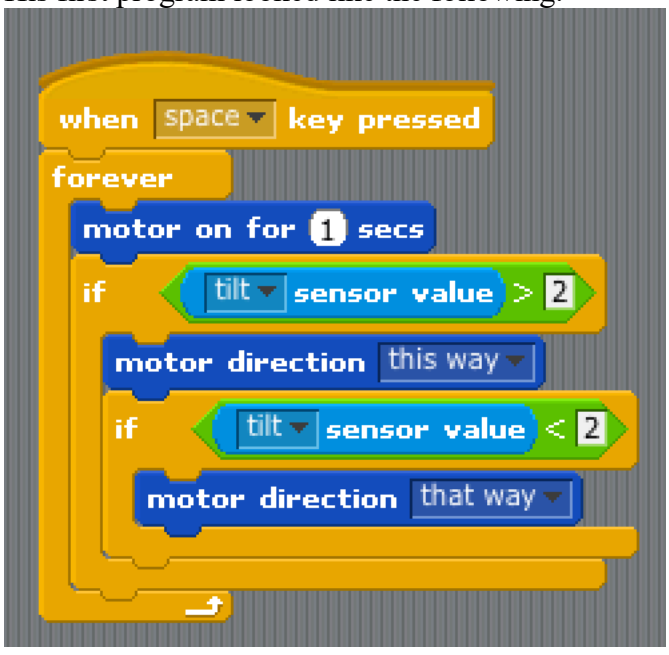
Seemingly simple projects can reveal authentic opportunities to grapple with big ideas
Connect a knob (potentiometer) to the Hummingbird Robotics controller and in Scratch or Snap!, ask students to write a program that turns the screen turtle as you adjust the knob.

The first thing you need to do is determine what sort of data you get from the knob sensor. The Hummingbird Kit has calibrated the sensor so that you get a lovely range of numbers between 0 – 100. The next thing you need to do is write a program to set the turtle's heading to the sensor value times 3.6. Simple, right? But ask 100 students from 3rd to 12th grade how to solve this problem without programming and you will see the weakness even in basic algebraic reasoning.

Too often, adults seek to assess or judge student projects without understanding the creative and intellectual processes employed. This leads them to dismiss or disrespect the thinking of their students. Simple activities like programming the Logo turtle to draw your initials includes more geometric reasoning than many students experience through high school. Creating a Scratch animation to illustrate a historical event can employ a laundry list of problem solving, mathematical, and computer science skills. The same goes for physical computing projects. Here is a seemingly simple problem using LEGO's early childhood robotics system, WeDo, and the Scratch programming language. It was observed in one of our recent "Invent To Learn" workshops for educators.

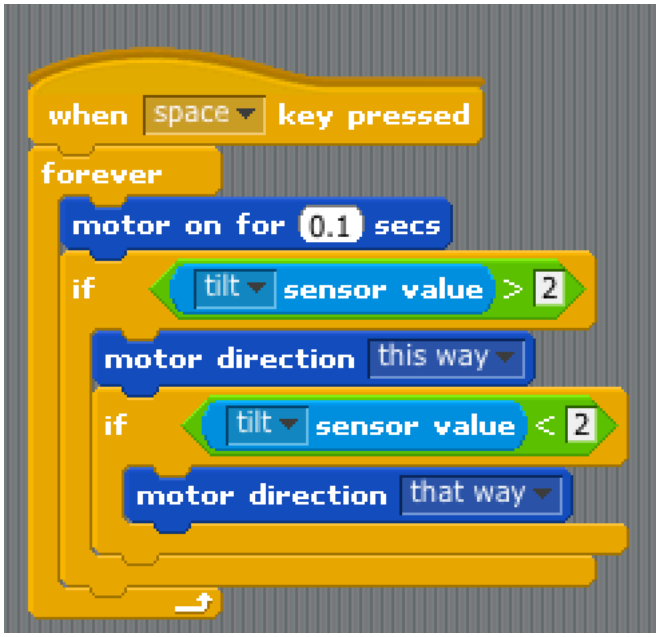
The student (in this case a teacher) connected a motor and a tilt sensor to his computer via the WeDo interface. He wanted to make the motor spin in the direction a user tilts the tilt sensor.

His first program looked like the following.



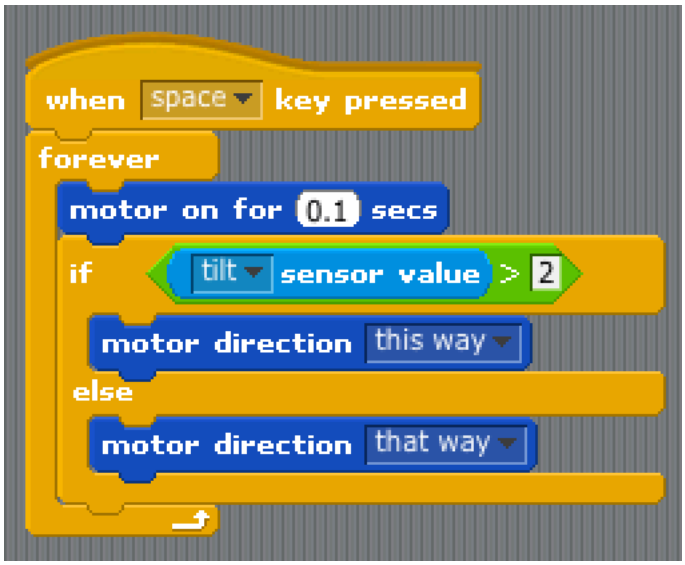
The problem was that while the motor spun rapidly, the sensor was not particularly responsive. It took a while for the motor to change direction when the sensor was tilted in the opposite direction.

The student thought that changing the length of time the motor was on for, from 1 second to 0.1 second might improve performance.



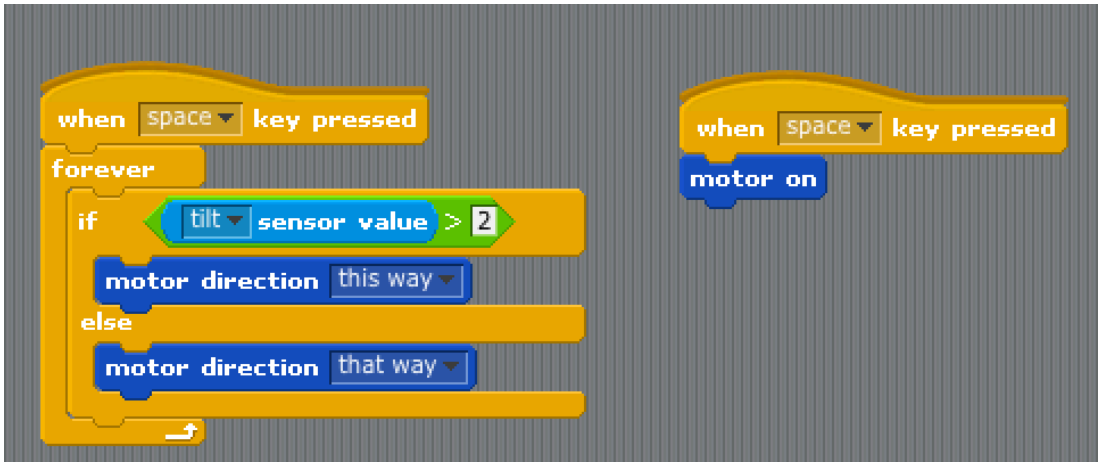
Well, the sensor performance *was* a lot more responsive, but now the motor did not spin as smoothly or as quickly. Hmmm... Tradeoffs? That's a big idea!

Perhaps a slightly more elegant program might help?



Using If/Else rather than embedded Ifs must speed the program up a hair, but we still faced the tradeoff between sensor response and motor speed.

Just then we remembered that Scratch is object-oriented and allows parallelism, the ability to execute multiple processes simultaneously. The following program allowed for speed and sensor response!



The superficially trivial problem borne from a student's curiosity offered a context for using a number of engineering, computer science, and mathematical big ideas that might otherwise be inaccessible to students.

Flying turtles
LogoWriter, MicroWorlds, Scratch, and Snap! offer students multiple turtles that could be animated with turtle geometry commands. Animation is turtle geometry with the turtle's pen up. Tickle's ability to program low-cost drones, extends turtle geometry into three dimensions in ways Papert might never have imagined. Predicting the behavior of your drone before you make execute your flying instructions may be more revolutionary than 3D printing.

#10 There is no substitute for personal computing
Since learning to code, complete with the inevitable task of debugging, takes time, it follows that access to computers becomes critical. Coding, like writing requires different motivation, venue, state-of-mind, and time of day for each programmer. Ideas don't stop when the school bell rings. That is why students need personal computers they may use 24/7 to store ideas, collaborate with others, and work on their projects continuously. You need a computer at your disposal whenever the coding muse strikes you.

To do this, we have to expand our vision beyond low-level computer curriculum that focuses on digital citizenship, internet research, and basic workflow. We cannot paradoxically complain that students are "digital natives" (a dubious term at best) and simultaneously lower expectations about what they will do with the computer.

If a 7th grader doesn't know how to save a file, they have never created anything with a computer worth saving.

There is a complex relationship between what students want out of school and what school wants out of students. Learning to program a computer can level that playing field, but only if we expand what it means to learn to program the computer. If we rigidly define computer science and coding as only learning about algorithms, discrete mathematics, and correct syntax, we will create a generation of young people for whom computer science is boring and worthless. Of course a few will succeed, there are always people who are attuned to subjects the way school traditionally serves them. If we allow ourselves to be cheered by that limited success, nothing will change. The "CS for All" movement will be declared a failure and some "new new" education reform will take its place.

This does not have to happen. We need to champion coding as a liberal art, interesting because of what you can do with it, and connect it with student interests and passions. We need to teach teachers how to create challenges that provoke and inspire, and build teacher's confidence in themselves as creators of an ever-changing curriculum that can respond to individual student interests while at the same time tackling the big ideas of science, math, engineering, social studies, art, and all subject areas that society values.

References

Ackermann, E. (2001). *Piaget's Constructivism, Papert's Constructionism: What's the Difference?* Paper presented at the 2001 Summer Institute, Mexico City.

Beauty and Joy of Computing. (2016). Retrieved from http://bjc.berkeley.edu

Bell, T., Alexander, J., Freeman, I., & Grimley, M. (2009). Computer science unplugged: School students doing real computing without computers. *The New Zealand Journal of Applied Computing and Information Technology, 13*(1), 20-29.

Bell, T. C., Witten, I. H., & Fellows, M. (1998). *Computer Science Unplugged: Off-line activities and games for all ages*: Citeseer.

Berry, A. M., & Wintle, S. E. (2009). Using Laptops to Facilitate Middle School Science Learning: The Results of Hard Fun. Research Brief. *Center for Education Policy, Applied Research, and Evaluation*.

Cavallo, D. (2000). *Technological Fluency and the Art of Motorcycle Maintenance: Emergent Design of Learning Environments.* (Ph.D.), Massachusetts Institute of Technology, Cambridge, Massachusetts.

CS Unplugged. (2016). Retrieved from http://csunplugged.org

Duckworth, E. R. (1996). *"The Having of Wonderful Ideas" & Other Essays on Teaching & Learning* (2nd ed.). New York: Teachers College Press, Teachers College, Columbia University.

Erickson, Barbara (2015). Is Computing Just for Men? American Association for University Women. Retrieved from: http://www.aauw.org/2015/03/11/is-computing-just-for-men/

Fellows, M., Bell, T., & Witten, I. (2002). Computer science unplugged. *Computer Science Unplugged*.

Goode, J., Chapman, G., & Margolis, J. (2012). Beyond curriculum: the exploring computer science program. *ACM Inroads, 3*(2), 47-53.

Goode, J., & Margolis, J. (2011). Exploring computer science: A case study of school reform. *ACM Transactions on Computing Education (TOCE), 11*(2), 12.

Grasso, I., & Fallshaw, M. (1993). *Reflections of a learning community: views on the introduction of laptops at MLC*: Methodist Ladies' College.

Greenberg, G. (1991). A creative arts approach to computer programming. *Computers and the Humanities, 25*(5), 267-273.

Guzdial, Mark. "NPR When Women Stopped Coding in 1980's: As We Repeat the Same Mistakes." Computing Education Blog. N.p., 30 Oct. 2014. Web. 21 June 2016.

<https://computinged.wordpress.com/2014/10/30/npr-when-women-stopped-coding-in-1980s-are-we-about-to-repeat-the-past/>.
Harel, I. (1991). *Children designers: interdisciplinary constructions for learning and knowing mathematics in a computer-rich school*. Norwood, N.J.: Ablex Pub. Corp.
Harel, I., & Papert, S. (1990). Software design as a learning environment. *Interactive Learning Environments, 1*(1), 1-32.
Harvey, B. (1982). Why Logo? . *Byte, 7,* 163-193.
Harvey, B. (1993). Symbolic programming vs. software engineering–fun vs. professionalism–are these the same question. *Eurologo1993*.
Harvey, B. (2003). Logo.
Henderson, P. (2008). Computer science unplugged. *Journal of Computing Sciences in Colleges, 23*(3), 168-168.
Herold, Benjamin (2014) Big Race, Gender Gaps In Participation On AP Computer Science Exam. Education Week (online) Jan 10, 2014. Retrieved from: http://blogs.edweek.org/edweek/DigitalEducation/2014/01/big_race_gender_disparities_discovered_on_ap_computer_science_exam.html
Johnstone, B. (2003). *Never Mind the Laptops: Kids, Computers, and the Transformation of Learning*. Seattle: iUniverse.
Kafai, Y. B. (1995). *Minds in play: computer game design as a context for children's learning*. Hillsdale, N.J.: L. Erlbaum Associates.
Kafai, Y. B., & Resnick, M. (1996). *Constructionism in practice: designing, thinking, and learning in a digital world*. Mahwah, N.J.: Lawrence Erlbaum Associates.
Kamii, C. (2000). *Young Children Reinvent Arithmetic: Implications of Piaget's Theory. Early Childhood Education Series*: ERIC.
Kamii, C., & Joseph, L. L. (2004). *Young children continue to reinvent arithmetic--2nd grade: Implications of Piaget's theory*: Teachers College Press.
Kohl, H. R. (2012). *The Muses Go to School: Inspiring Stories about the Importance of Arts in Education*: The New Press.
Luehrmann, A. (1980). Computer Literacy: The What, Why, and How. In R. Taylor (Ed.), *The computer in the school: Tutor, tutee, and tool*. New York: Teacher College Press.
Luehrmann, A. W. (1972). *Should the computer teach the student, or vice versa?* Paper presented at the Proceedings of the May 16-18, 1972, spring joint computer conference.
Martinez, S.-L., & Stager, G. (2013). *Invent to learn: making, tinkering, and engineering in the classroom*: Constructing Modern Knowledge Press.
Outlier Research and Evaluation. (2015). *High School – Exploring Computer Science - Interviews*. Retrieved from Chicago: http://outlier.uchicago.edu/evaluation_codeorg/highschool-exploringCS-interviews/
Papert, S. (1971). A computer laboratory for elementary schools.
Papert, S. (1972a). *On making a theorem for a child*. Paper presented at the Proceedings of the ACM annual conference - Volume 1, Boston, Massachusetts, USA.
Papert, S. (1972b). Teaching Children Thinking∗. *Programmed Learning and Educational Technology, 9*(5), 245-255.
Papert, S. (1972c). Teaching children to be mathematicians versus teaching about mathematics. *International Journal of Mathematical Education in Science and Technology, 3*(3), 249-262.
Papert, s. (1980a). Computer-Based MicroWorlds as Incubators for Powerful Ideas. In R. Taylor (Ed.), *The Computer in the School: Tutor, Tool, Tutee* (pp. 204-210). NY: Teacher's College Press.

Papert, S. (1980b). *Mindstorms: children, computers, and powerful ideas*. New York: Basic Books.
Papert, S. (1985). Different visions of Logo. *Computers in the Schools, 2*(2-3), 3-8.
Papert, S. (1988). The Conservation of Piaget: The Computer as Grist. In G. Forman & P. B. Pufall (Eds.), *Constructivism in the computer age* (pp. 3-14). Hillsdale, New Jersey: Lawrence Erlbaum Associates.
Papert, S. (1993). *The Children's Machine: Rethinking School in the Age of the Computer*. NY: Basic Books.
Papert, S. (1997, May 15, 2005). Looking at Technology Through School Colored Spectacles. Retrieved from http://papert.org/articles/LookingatTechnologyThroughSchool.html
Papert, S. (1998). Technology in Schools: To support the system or render it obsolete. *Milken Exchange on Education Technology.* Retrieved from http://www.mff.org/edtech/article.taf?_function=detail&Content_uid1=106
Papert, S. (1999). Introduction: What is Logo and Who Needs It? In LCSI (Ed.), *Logo Philosophy and Implementation* (pp. v-xvi). Montreal, Quebec: LCSI.
Papert, S. (2000a). *Climbing to Knowing - A Constructionist Ideas-Rich Activity Theme*. The Seymour Papert Institute/Learning Barn.
Papert, S. (2000b). What's the Big Idea? Toward a Pedagogical Theory of Idea Power. *IBM Systems Journal, 39*(3&4), 720-729.
Papert, S. (2002). Hard Fun. *Bangor Daily News*.
Papert, S. (Ed.) (1991). *Situating Constructionism*. Norwood, NJ: Ablex Publishing Corporation.
Papert, S., & Solomon, C. (1971). *Twenty things to do with a computer*. Retrieved from Cambridge, MA:
Papert, S., & Watt, D. H. (1977). Assessment and documentation of a children's computer laboratory.
Papert, S., Watt, D., diSessa, A., Weir, S. (1979). *Final Report of the Brookline Logo Project, An Assessment and Documentation of a Children's Computer Laboratory, Part III, Detailed Profiles of Each Student's Work*. Retrieved from Cambridge, Massachusetts:
Piaget, J., & Piaget, J. (1973). *To understand is to invent: the future of education*. New York,: Grossman Publishers.
Squires, D., & McDougall, A. (1994). *Choosing and using educational software: a teachers' guide*: Psychology Press.
Squires, D., & Preece, J. (1996). Usability and learning: evaluating the potential of educational software. *Computers & Education, 27*(1), 15-22.
Stager, G. (1998). Laptops and learning: can laptop computers put the "C"(for constructivism) in learning. *Curriculum Administrator. Received on November, 6*, 2003.
Stager, G. (2003). The Case for Computing. In S. Armstrong (Ed.), *Snapshots!: Educational Insights from the Thornburg Center*. Lake Barrington, IL: Thornburg Center.
Stager, G. (2006). Laptops: growing pains and disappointments. *Teacher: The National Education Magazine*(Aug 2006), 44.
Stager, G. (2014, October 2014). *Progressive Education and The Maker Movement-Symbiosis Or Mutually Assured Destruction.* Paper presented at the Fab Learn, Palo Alto, California.
Stager, G. S. (1995). Laptop Schools Lead the Way in Professional Development. *Educational Leadership, 53*(2), 78-81.
Stager, G. S. (1997). Logo and Learning Mathematics-No Room for Squares. *Computers in the Schools, 14*(1-2).

United States Department of Education (2010). A Blueprint for Reform: The Reauthorization of the Elementary and Secondary Education Act Retrieved from: http://www2.ed.gov/policy/elsec/leg/blueprint/blueprint.pdf