# 6

# Platforms and Tools:

# *Anything You Can Do, I Need To Do Cheaper*

“... there is nothing like making music and messing with sound to inspire people to learn how to program.” [[19](#_ENREF_19)]

## Getting Into the Game

On their page intended to woo prospective graduate students, the Georgia Tech School of Music website says [[8](#_ENREF_8)]:

Successful design and development of music technology systems must be supported by knowledge of music theory, perception, composition, and performance, as well as digital media, computing, electrical and mechanical engineering, and design.

We don’t disagree, but that’s an awful lot to know! What’s more, requiring students to have even a subset of these skills before they can “get in the game” deprives a huge percentage of them of the opportunity to learn valuable computing skills through the engaging power of music.

There is no end to the money you can spend on technology to gain the ability to design and create. For certain types of projects, professional or “prosumer” software applications boasting the latest bells and whistles might in fact make total economic sense in terms of functionality and time. However, we don’t feel that it’s necessary to jump into the higher end of the software market at the beginning stages of learning computational skills. In addition, such costs are prohibitive for most undergraduates and even graduate students. Of course, one could outfit a computer lab available to students with this level of software, but then they would have to do all their assignments in the lab, which is simply not a practical solution.

We think it is important that students can run the same software on their own systems that is demonstrated in class and with which they are expected to do their assignments. We therefore suggest that you adopt software platforms that you can download freely from the web, but that still allow you to explore broad computing and music concepts that are common to the higher end platforms. We don’t contend that such software is as sophisticated or as polished as its professional or prosumer cousins, but it is most likely fully sufficient for your teaching purposes. *Remember:* Your goal is to make the excitement of music technology accessible to students so that you can teach computing and music together in engaging ways. This approach to computing+music is not about training audio engineers.

The argument for low cost of entry is particularly compelling in elective courses at the introductory level (like ours), because most of these students will not go on to take another course in which they might use the same tools. Thus, we will readily trade a bit of quality or a lack of advanced features for free cost. In addition, “good enough” is a compelling criteria for evaluating what platforms and tools to use, especially when one considers students’ pocketbooks. We also find that “good enough” does not hinder creativity or impede learning. On the contrary, it might actually enhance those outcomes.

|  |
| --- |
| Figure 6-1. CD cover for the cast   recording of *Jersey Boys*. |

Consider this little anecdote. Jesse and his wife Bonnie went to see *Jersey Boys*, the story of the iconic Four Seasons, in Boston. The show was terrific, and on the way out Bonnie wanted to buy the CD. It was $25 at the theater, and Jesse said that they could probably buy it from Amazon.com at half that price. Bonnie acknowledged that, but said that she wanted to listen to all those great songs on the drive home. As you can probably guess, the CD was bought at the theater, and indeed they enjoyed it almost as much as the show as they sang along in the car.

Once they got home, Jesse went on Amazon.com and bought four original Frankie Valli and the Four Seasons CDs for the same $25. When they came a few days later and Jesse listened to them in the car, something seemed to be missing. The original Four Seasons weren’t as good as the actors portraying them, at least not on their respective recordings. Why?

Well, there are two reasons. The first is simply 50 years of improvements in recording technology. Even though the old recordings had been re-mastered, they still couldn’t compare to the brilliance of the show recording. But the second reason is actually the more interesting. Just who were Frankie Valli and the Four Seasons, anyway, and what was their musical training?

|  |
| --- |
| FrankieValliFourSeasons_500x500.jpg  Figure 6-2. CD cover for a recording  of the original Frankie Valli  and the Four Seasons. |

Before we answer that question, consider the backgrounds of the actors who play the Four Seasons in the national touring company of *Jersey Boys* that played in Boston. Matt Bailey (*who played the role of Tommy DeVito*) has two B.F.A. degrees from the University of Arizona, one in Acting/Directing and the other in Musical Theatre [[1](#_ENREF_1)]. Joseph Leo Bwarie (*Frankie Valli*) has a B.F.A. in Acting from Boston’s Emerson College [[5](#_ENREF_5)]. Josh Franklin (*Bob Gaudio*) has a B.F.A. in musical theater from Webster Conservatory for the Theater Arts in St. Louis, Missouri, and has studied music and theory with Linda Weiss, a graduate of the Juilliard School of Music and founder of the Colorado Springs Conservatory [[7](#_ENREF_7)]. Steve Gouveia (*Nick Massi*) is the odd man out in this regard, having taken only “a few voice lessons here and there,” and claiming that he was simply “blessed with a good singing voice” [[9](#_ENREF_9)].

The original Frankie Valli and the Four Seasons, on the other hand, were just — as they say near the end of the show — “four guys under a streetlamp” making “that sound, our sound.” They had little to no formal training, but they sure had style. And heart. And listening to them even on compressed MP3s using tiny earbuds is enough to make even the most hardened audio engineer smile.

So while the best recording equipment and the best software may be financially out of reach for most of us, — especially our students — we can all still make music. Good music. What’s more, it is entirely feasible to teach a lot *about* music and explore the computational thinking techniques that complement the musical concepts we introduce in this book using “good enough” tools. And even more, when assignments are created to reinforce those concepts, or to get students to extend them by writing programs, they can do so on their own systems in their own homes or dorm rooms. For these purposes, accessibility trumps professional quality any day.

## Sound Editing

### Audacity

We have already discussed Audacity (audacity.sourceforge.net, free) at considerable length. This is the main sound editor we use in our classes and that we highly recommend, not only because it is free, but because it will be more than adequate for your purposes and runs well under both Windows and Mac OS. Most Music students are already quite familiar with Audacity, but few have explored its capabilities thoroughly. They may have used it to record themselves and then cleaned up those recordings. For many, the most sophisticated thing they may have done beyond simple cut-and-paste editing and trimming is to normalize the waveform of a soft recording to increase its volume. Few at the beginning GenEd level understand the ramifications of clipping or the relationship between volume and decibels.

This is OK, of course, and we repeat that the point of this work is not to train audio engineers. But Audacity’s many effects and filters provide wonderful opportunities to teach CT while students are, in the words of Dan Truman, “messing around with sound” [[19](#_ENREF_19)]. (Dan Truman [[14](#_ENREF_14)], along with Perry Cook , is also the founder of the Princeton Laptop Orchestra, “PLOrk” [[17](#_ENREF_17), [18](#_ENREF_18)].)

For example, when attempting to learn a song from a recording it is often useful to slow the tempo down. Audacity has an effect that allows you to change tempo without changing pitch (see Fig. 6-3). This dialog box is very nice because it allows you to specify the change by specifying either a positive or negative percentage or a beats per minute value. If you enter numbers into the “Beats per minute” fields, the percentage change is automatically calculated for you. The resultant length (the time to play the entire recording) is also calculated automatically. One thing you can do with this dialog box is to have students compute the values themselves and then use Audacity to verify their answers. Or you can turn the problem on its head by slowing a recording down or speeding it up and then asking students to figure out how to reverse that change using only the “Percentage Change” field. The answer is that you have to use a reciprocal. That is, if you slow it by 50% (1/2), to get back to the original tempo you have to speed it up by 100%. But is that 2/1? No, it’s not. The arithmetic gets a bit tricky here. To slow by 50%, you’re really using (1/2) – 1 = –1/2 = –50% (note the negatives). To get back to the original you use (2/1) – 1 = +1 = 100% (positive values). This makes it easier to see the reciprocal relationship, but some students will obviously have trouble understanding the computations. Things get even more interesting if you only want to slow it by 1/3, which is 2/3 of the original speed. This computation becomes (2/3)–1 = –0.33 = –33%. To return to the original speed, you need to use (3/2)–1 = +0.5 = 50%. You can then contrast changing tempo without changing pitch with simply changing the speed, which is another Audacity effect, but one that affects both tempo and pitch. Students will clearly hear the difference between the two.

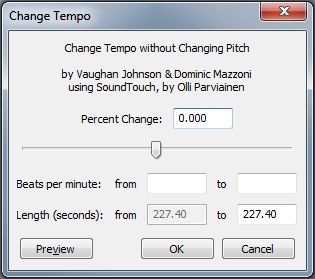


Figure 6-3. Audacity dialog box for   
“Changing Tempo without Changing Pitch.”

Students in other Arts majors are much less likely to be familiar with Audacity. Like Science and Engineering majors, many of these students may have never used any sound editing software at all, including the limited applications that come with their own systems.

Thus, Audacity is a perfect choice for students’ first sound editor. What’s more, given that many Music majors at least know the concepts that some of Audacity’s effects implement — such as changing pitch without changing tempo and vice versa — they get an opportunity to teach the Science and Engineering majors. When we get into programming later in the course, the Science and Engineering majors typically teach the Music and Arts majors. This is truly reciprocal learning [[2](#_ENREF_2)] at work, and we consider that the best way to engage the entire class.

### DVDVideoSoft Free Studio and AVS4YOU

We have also mentioned the DVDVideoSoft suite of tools (dvdvideosoft.com, free, see Figure 6-4). These tools are mostly converters, not editors, but they are invaluable for capturing songs from YouTube and other sources and converting them to formats that Audacity can work with. There are 45 programs in the entire package, all free!



Figure 6-4. DVDVideoSoft Free Studio Manager.

AVS4YOU (www.avs4you.com, free with logos and watermarks that can be removed for a small registration fee) is a full suite of 16 editing and associated audio, video, and image editing tools (see Figure 6-5**)**. The frequently asked questions (FAQs) section of their website states:

The non-activated [*free and unregistered*] programs do not have any feature or time limitations. The only thing is that they have a voice logo in the output audio files (that is true for audio programs, such as AVS Audio Editor, AVS Audio Recorder, etc.) or a watermark in the output video files (AVS Video Converter, AVS Video Editor, etc.). To remove logos and watermarks you need to activate your programs [*by registering them and paying a small fee*] and reconvert your source audio or video files.

This will be fine for most school based purposes, as long as you don’t mind seeing these logos and watermarks on student assignments. As usual, these tools are not as sophisticated as the expensive programs, but they are very impressive for the price.[[1]](#footnote-1) In addition, they are very easy to use, which is of course important for students at the level we teach and useful for everyone.

Some of the DVDVideoSoft programs are available for the Macintosh, but as of this writing the AVS4YOU programs explicitly specify “no Mac OS/Linux support.” However, you can find similar tools for the Mac with straightforward Google searches. For example, we found the free “Enolsoft Free YouTube Downloader HD for Mac” at www.enolsoft.com/free-youtube-downloader-hd-for-mac.html and “SoundConverter for Mac” at soundconverter.en.softonic.com/mac. And of course, there are a number of tools that come with Macs, such as GarageBand, or that are specifically designed for the Mac with which you can also do basic sound editing.

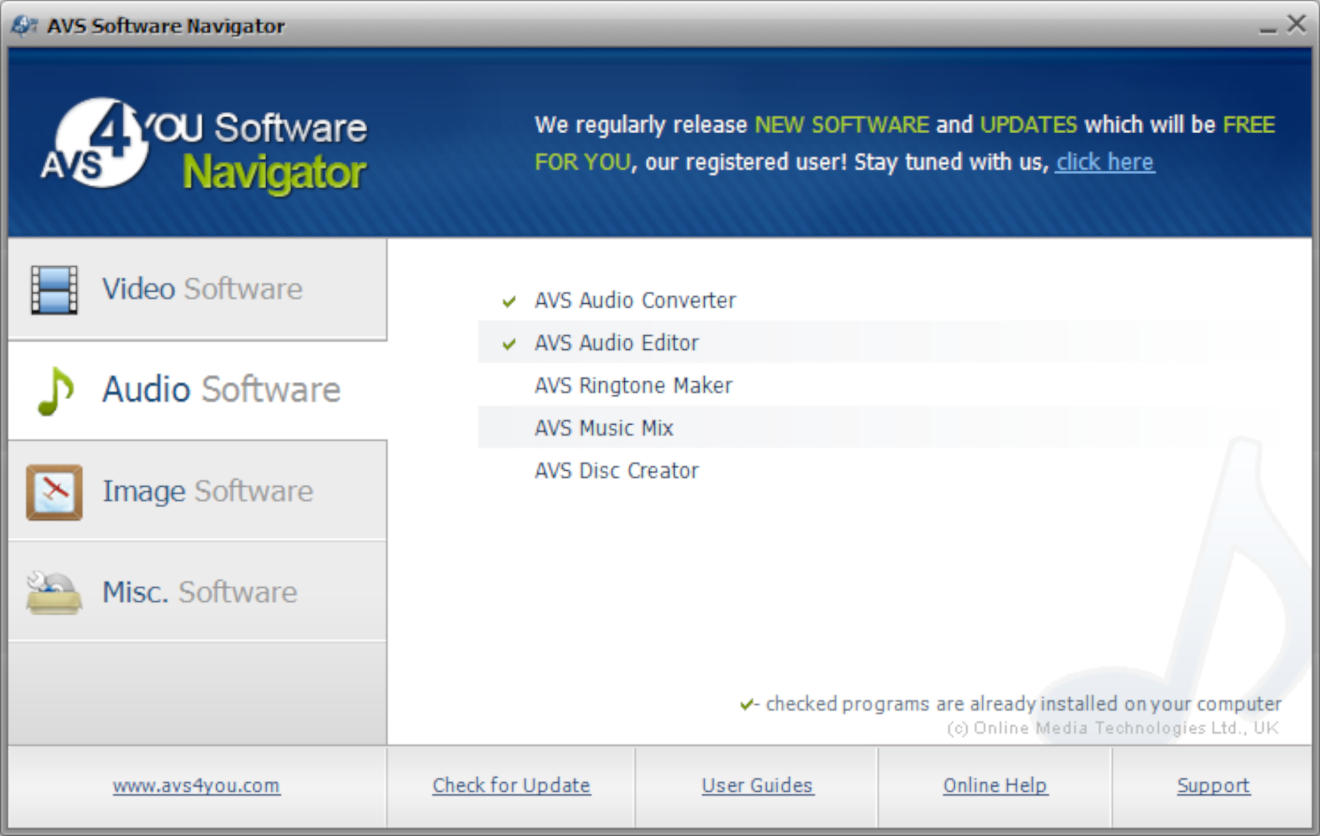


Figure 6-5. AVS4You Software Navigator.

The bottom line is that in addition to Audacity, there are numerous other options for creating and editing existing or newly recorded sounds. To repeat, however, your goal in selecting tools should be to use those with a low threshold, both in terms of cost and learning curve. While computer music is the *vehicle* for teaching computational thinking, you don’t want to become so tool-centric that your students “can’t see the forest for the trees.” We suggest that you use tools that your students can pick up quickly, even if they don’t use all of the tools’ features. After all, how many of us, even those who consider themselves advanced, really use or are even familiar with *all* the features of Word or Excel?

## Sound Programming

The same philosophy applies to our choice of sound programming software. To Jesse, programming is a delight. He sees it as a highly creative activity in which he gets to express himself through hundreds of small decisions about how to design and implement an algorithm and how to present information to a user. He takes real pleasure in coding elegant solutions to sometimes complex problems.

To Gena, programming is at best a black art, and at worst a black hole! She doesn’t enjoy becoming engrossed in the rigid structure of computer languages as Jesse does. She has spent countless hours achieving the skill required to produce precise sounds on her violin, but she doesn’t relish spending similar hours teaching a computer to do likewise. Gena believes that software for the average person should be intuitive, logical, and user-friendly. And most important, it should work, plain and simple. When it doesn’t, there’s little joy in the effort it takes to figure out why.

When we talk about these differences in professors’ perspectives, you might consider them cute. But when you face an interdisciplinary class full of students from disciplines that are, at least on the surface, near polar opposites of each other, they present a real challenge to keeping all students engaged. If we program in a language or system that is comprehensible only to Computer Science majors, the Music majors will tune out, regardless of how patient we are in explaining what’s going on. If we program in a language or system that is easily grasped by the Music majors but is so limited as to be seen as a “toy,” not only will the CS majors tune out, but the Music majors will quickly become bored, as well. The trick is to find a compromise.

### Good Tools that Just Don’t “Work” for Our Students

There are, of course, many programming tools and languages and libraries and systems for programming music. Some are built on standard programming languages, such as jMusic [[3](#_ENREF_3)], which is built on Java, while others are built completely from scratch. Some are corporate products that are professionally maintained, while others are homegrown. The latter are less likely to have comprehensive documentation and rarely have any technical support.

As always, we look for free tools. As mentioned earlier, the problem is to find one that can be used by all of our students. As an example of a good tool that just doesn’t “work” for our students, consider the SuperCollider [[10](#_ENREF_10), [20](#_ENREF_20)] program in Figure 6-6. Believe it or not, this program plays the famous guitar riff in Led Zeppelin’s *Kashmir*. (To hear this riff, visit www.youtube.com/watch?v=hAzdgU\_kpGo.)

It’s important to stress that we are not knocking SuperCollider. On the contrary, it’s a great system that some of our colleagues have used to produce very cool music. It’s incredibly powerful in the hands of experienced users. Our only issue is that the large learning curve involved in understanding the SuperCollider syntax makes it inappropriate for an entry level interdisciplinary course in computing+music. Even with the level of documentation provided in the previous figure, our students would simply be blown away by the complexities of this code.[[2]](#footnote-2)

*// Select all between the parentheses and*

*// press [Ctrl+KeyPad Enter] to load the synthdef*

( *// The actual synth and envelope*

**SynthDef**( "kashmir",

**arg** sound, freq ;

**var** sin, env\_gen, env, freq\_env ;

env = Env.triangle( 0.2, 0.2 ) ;

*// Env.triangle has ( duration, level [, peak] ) ;*

env\_gen = **EnvGen**.kr( env, doneAction: 2 ) ;

*// EnvGen plays back the envelope*

*// when done, it frees up the space used by the synth*

sin = **SinOsc**.ar( freq, 0, env\_gen ) + **Saw**.ar( freq, env\_gen ) ;

*// SinOsc takes 4 arguments ( freq, phase, mul [, add] )*

*// Mul is the wave multiplier, to change the amplification*

*// Saw takes ( freq, mul [, add] )*

**Out**.ar( [0,1], sin )

*// Array enables stereo ( 0 left, 1 is right )*

*// Out.ar takes 2 arguments, the busses to write out to*

*// and the source of the sound*

).load(s);

*// The load message loads this SynthDef to the server*

)

*// Select all between the parentheses again and*

*// press [Ctrl+KeyPad Enter] to play*

(

**var** x = 45, a = 6 ;

p = **Pseq**([ 45, 46, 47, 48 ], inf).asStream ;

*// Pseq needs "asStream", or it won't play in sequence*

*// this Pseq sequences through the notes*

q = **Pseq**([ 0.2, 0.2, 0.8 ], inf).asStream ;

*// this Pseq sequences the seconds to wait between notes*

t = **Task**( *// a task is a stream that can be paused*

loop( *// loops forever*

if( a < 6, a = a + 1 , a = 1 ; x = p.value ) ;

*// every 6 notes, x changes*

y = **Synth**( "kashmir", [ freq: x.midicps ] ) ;

*// calls SynthDef "kashmir" with current note*

*// midicps (cycles per second) allows you to use midi*

*// numbers, which are then converted to hertz*

q.value.wait ;

// #.wait is the number of seconds to wait

);

);

t.start ;

)

*// Press [Ctrl+KeyPad Enter] on line below to stop the task*

t.stop ;

Figure 6-6. SuperCollider program to play the Led Zeppelin *Kashmir* riff.

The same is true of Java with jMusic. Most (but not all) of our Computer Science students are familiar with Java and know how to use libraries such as jMusic, but they would have a very hard time explaining all the non-music structures to their Music student partners. You can probably think of a number of other systems that you may actually use yourself but that simply wouldn’t be appropriate for your students. The bottom line is that we need a less intimidating, more intuitive, more user-friendly, yet still powerful, solution.

### A Good Tool that Does “Work” for Our Students

Our solution is Scratch, a visual programming environment developed by the Lifelong Kindergarten Group at MIT’s Media Lab. It is freely downloadable from scratch.mit.edu. It runs on Mac OS X (version 10.4 or later), numerous versions of Windows, and Ubuntu.[[3]](#footnote-3)

Although Scratch is designed for young children, it “works” for college students for a number of reasons. First, one should not be mistaken by its stated target audience or the word “Kindergarten” in the name of the group that developed it. In experienced hands, Scratch can be used to produce very sophisticated programs. The beauty of Scratch’s visual programming paradigm is that it makes the threshold of entry into the world of programming very low. That is, complete novices can do interesting things very quickly, and the block design makes understanding the programming constructs involved considerably easier than seeing them in textual code.

The “Lifelong Kindergarten Group” is not about developing software for children in kindergarten. It is about developing software that makes it easy to experiment and joyful to learn by trial and error, just as kindergarten children do. The Director of the Lifelong Kindergarten Group is Prof. Mitchel Resnick, a proponent of Seymour Papert’s constructionist theories of learning [[12](#_ENREF_12)]. Resnick argues that “the ‘kindergarten approach to Learning’ — characterized by a spiraling cycle of Imagine, Create, Play, Share, Reflect, and back to Imagine — is ideally suited to the needs of the 21st century, helping learners develop the creative-thinking skills that are critical to success and satisfaction in today’s society” [[15](#_ENREF_15)].

Thus, Scratch is not a “toy” language or system. Resnick designed it to achieve goals that he credits to Seymour Papert, as embodied in Papert’s seminal book *Mindstorms* [[11](#_ENREF_11)], and built on concepts pioneered in the Logo programming language.

Papert argued that programming languages should have a “low floor” (easy to get started) and a “high ceiling” (opportunities to create increasingly complex projects over time). In addition, languages need “wide walls” (supporting many different types of projects so people with many different interests and learning styles can all become engaged). [[16](#_ENREF_16)]

Resnick goes on to say that his group plans “to keep our primary focus on lowering the floor and widening the walls, not raising the ceiling,” but for our purposes, Scratch already contains a sufficiently high ceiling that incorporates a number of important computer science constructs that we use to teach computational thinking: code blocks, global and local variables, lists, loops, conditionals, and subroutine-like structures.[[4]](#footnote-4)

But most importantly for us, Scratch includes some great music functionality. First, it can play a number of built-in sounds, or sounds stored in MP3 files. You can also record your own sounds and make them available to Scratch without leaving the environment (see Figure 6-7).

But Scratch doesn’t stop there. The secret to the music capabilities is that the lead programmer for Scratch is John Maloney, a gifted software developer who was a member of the team that developed Squeak (www.squeak.org), an experimental programming system for elementary school children headed by computer pioneer Alan Kay. Among other things, John had primary responsibility for Squeak’s sound and music facilities. John integrated a number of those facilities into Scratch, particularly its ability to *generate* music using MIDI. Playing prerecorded sounds is one thing, but truly generating music it is quite another. This is the part of Scratch that we use most earnestly to teach CT through music programming.



Figure 6-7. The Scratch panel from which   
one can play, import, and record sounds.

## Teaching CT with Scratch

### Working with Scratch

It is not our purpose for this book to be a Scratch programming manual. What’s more, we are quite confident that readers of this book wouldn’t even need a manual to get up and running with Scratch. You just download it, install it, and start dragging blocks around and connecting to form structures. (Gena says, “If I can program with Scratch, anyone can!”) For those who like a little hand-holding, however, we recommend the Scratch support page at info.scratch.mit.edu/Support.

The remainder of this chapter introduces the music capabilities built into Scratch that we use and demonstrates the scope of ways in which we use them to teach CT. We provide examples of our own work, that of students, and even some extensions by others’ work in this area. It is of course impossible to cover everything, however. Despite its apparent simplicity, Scratch is incredibly rich. Readers who desire to see more examples are directed to the class notes and assignments on our course websites. The website for the current semester can be accessed by going to http://soundthinking.uml.edu. Websites for previous semesters’ offerings are linked from http://teaching.cs.uml.edu/ ~heines/teaching.jsp, where they are labeled 73.212 / 91.212 -- Sound Thinking.

*Note to the Editor:* These links will eventually be replaced by simply directing the reader to the book’s website, where there will be links to our course websites as well as other resources.

Handouts and sample programs from our *Making Music with Scratch* workshops are also available at www.performamatics.org.

### Generating Music with Scratch

Figure 6-8 shows the Scratch Sound panel, from which you can select blocks to add to your program. When you drag a  block into the Scripts area, you can change the note’s MIDI value to be played by clicking the ⯆ next to the note number. This brings up a piano keyboard as shown in Figure 6-9. You can also change the note value manually by clicking it and then entering a number as shown in Figure 6-10. Using only these basic capabilities, we are able to program a linear first version of *Frère Jacques* as shown in Figure 6‑11. As you can see from this example, each note is entered manually input including the notes that repeat.

|  |  |  |
| --- | --- | --- |
| Figure 6-8. The Scratch Sound panel. |  | Figure 6-9. Changing a note’s  MIDI value by clicking the ⯆  and then selecting it from the  popup keyboard.    Figure 6-10. Changing a note’s  MIDI value by clicking the value  and then typing a new value. |

|  |  |  |
| --- | --- | --- |
| Figure 6-11. First version of *Frère Jacques* with each phrase repeated in straight linear fashion. (The Scratch version is coded an octave lower than the score.) |  |  |

Coding music linearly is rather tedious, to say the least, and it isn’t necessarily the way musicians think about music. Musicians will more often than not think about music at the phrase level and look for patterns. Figure 6-12 takes advantage of the “looping constructs” in the Scratch Control panel (Figure 6-13) to make the coding more efficient.[[5]](#footnote-5) As you will notice, the first four notes in our example in Figure 6-11 repeat. Placing those notes within a repeat block allows us to repeat that pattern as many times as we wish. We can then create subsequent loops with the next series of patterns we hear, as you can see in Figure 6-12.

The real power to generate music with Scratch comes when we replace the “hard-coded” values in the play note block with variables. “Hard-coded” means that actual numbers are specified in the program, such as the MIDI values 60, 62, and 64. Programmers typically avoid hard-coding whenever possible so that when they need to make a change to a program, such as changing a song’s starting note, they only need to make that change in one spot. To do this, you need to use variables. If you set a variable to the starting note and each time you need that note you refer to the variable, it will be correct everywhere throughout the program.

|  |  |  |
| --- | --- | --- |
| Figure 6-12. Second version of *Frère Jacques* with each phrase repeated using a Scratch repeat loop. (The Scratch version is coded an octave lower than the score.) | Figure 6-13. The Scratch Control panel. |  |

To do this in Scratch, you display the Variables panel and click the Make a variable button (Figure 6‑14a). This brings up a dialog box in which you enter the name of your variable (Figure 6‑14b). Once a variable has been created, Scratch makes a number of operations available that you can use to manipulate that variable (Figure 6‑14c).

|  |  |  |
| --- | --- | --- |
| (a) | (b) | (c) |
| Figure 6‑14. Creating a Scratch variable named note and the operations that Scratch makes available after at least one variable has been created. | | |

In the program in Figure 6‑15, you can use the note variable, which is the variable you just created when you clicked on the Make a variable button in the Operator panel. Notice that the various types of procedures one can perform in Scratch are color coded based on how each series of blocks functions. For example, Variables blocks are orange, while Sound blocks are magenta. Most Western music is based around some kind of key structure, whether it’s major, minor, etc. Therefore, in the *Frère Jacques* example, we create a variable to represent the tonic, or key, note. Using this variable with the + operator available in Scratch’s Operator panel (see Figure 6‑16), you can program each play note block to play a note that is some offset from the tonic. That is, once you establish what the tonic of the composition will be, you can program notes based on the difference in semi-tones between the tonic and the note you want played. This allows you to program a second version of *Frère Jacques* that can play the song in any key and octave you choose.

Not only does this program introduce Music majors to variables and the use of mathematical expressions (such as tonic+7) in programming, but it also introduces Computer Science students to the musical concept of transposition. Thus, the process of developing this single program demonstrates not only the power of Scratch, but also an approach to teaching computational thinking in an interdisciplinary fashion.

|  |  |
| --- | --- |
| Figure 6-15. *Frère Jacques* programmed using a variable for the tonic (or key) note, with each note to be played coded as an offset from the tonic. This version allows the song to be played in any key. | Figure 6-16. The Scratch  Operators panel. |

**Please Note**

Scratch is built for animation. For this reason, Scratch *intentionally* *slows down* processing so that animations do not run too quickly on today’s fast computers. However, “fast” is of course a relative word. What’s too fast for animations is sometimes too slow for music, where precise timing is necessary. Therefore, to ensure that Scratch processes blocks as quickly as it can so that your music plays as smoothly as possible, make sure to set Scratch’s “single-step speed” to “turbo speed” by selecting the Set Single Stepping… option from the Edit menu and then clicking Turbo speed as shown in Figure 6‑17.

|  |  |
| --- | --- |
| (a) | (b) |
| Figure 6‑17. Setting “turbo speed” to improve music timing. | |

### Going Further with Scratch Music

A more advanced approach uses the Scratch list capabilities, which are series of values that can be referred to by an index value. For example, if we have a list named CMajorNotes that contains the notes of a C major scale (60, 62, 64, 65, 67, 69, 71, 72), we can refer to the third note as “item 3 of notes.” The Scratch item *n* of *list* block  in the Variables panel is intended precisely for this purpose.

Lists can be also used to teach numerous computational thinking concepts. For example, in addition to the basic concept of an indexed data structure, a great deal can be done with computing the index of the note you want to play. When combined with loops, the structure becomes even more powerful. It is interesting to write code that plays the scale from the bottom note upwards and then again from the top note back down. Or you can play every other note. Or you can pick out notes that make a chord. All such programs require CT.

To make a list, you begin by clicking the Make a list button in the Scratch Variables panel as shown in Figure 6‑18a. Just like when we created a variable, this brings up a dialog box for you to name your list (Figure 6‑18b). The Variables panel then displays a list of operations that you can do on the list (Figure 6‑18c), and an empty list display appears in the stage area (Figure 6‑18d).

|  |  |  |
| --- | --- | --- |
| (a) | (b)    (d) | (c) |
| Figure 6‑18. Creating a Scratch list named notes and the operations that Scratch makes available after at least one list has been created. | | |

There are several ways to get data into a list. The most straightforward is to enter values manually by clicking the + in the lower left-hand corner of the list display and then entering the values one at a time, pressing the Enter key after typing each one (see Figure 6‑19). For music, however, this approach is not optimal. With a lot of MIDI note values to enter, which is the usual case, it is easy to miss a note or to enter a note twice. Editing the list in Scratch is a little tricky, so we prefer to create our lists of notes in external text files and then import those lists into Scratch.



Figure 6‑19. Clicking the + sign on the list display to enter note values manually.

The list to be imported into Scratch must be a simple text file with one value per line. This can easily be created using the built-in Notepad application on Windows systems or the built-in TextEdit application on Macintosh systems. If you use TextEdit, however, be sure to save your list of numbers as plain text by selecting the Plain Text option in the File Format dropdown list when you first save the file (see Figure 6‑20). If you save the file as Rich Text Format, which is the default, you will not be able to import the list into Scratch.



Figure 6‑20. Creating and saving a list of numbers to be imported into a Scratch list using the TextEdit application on a Macintosh system. Note that the file must be saved in Plain Text format for Scratch to be able to read it. After Plain Text format is selected, the entry in the Save As box will change to Untitled.txt, which you can then change to whatever filename you like.

Once you’ve created your list, right-click (control-click or two-finger click on the Mac) the list display to pop up the menu shown in Figure 6‑21a and click the import… menu item. This brings up the dialog box in Figure 6‑21b. Click the Computer button to navigate to the folder in which you stored your text file, select the file, and click the OK button. Your current list values will be deleted, and then the entries in the text file will populate the list. If you make a mistake, it’s usually easier to change the text file and reimport the list values than to try to edit them from within Scratch.

|  |  |  |
| --- | --- | --- |
| (a) |  | (b) |
| Figure 6‑21. Importing a list of numbers into a Scratch list. (a) The popup menu that appears when you right-click a list display. (b) The dialog box that appears when you select the import… menu item. | | |

For *Frère Jacques*, we created two lists and one variable (see Figure 6‑22). The first list contains all the MIDI note values, while the second contains the rhythm values (in beats). Note that whole numbers are used for the MIDI values, while decimal numbers are used for the rhythm values. This is perfectly fine: Scratch can handle them both. What’s more, Scratch lists can contain strings of characters, as well.[[6]](#footnote-6) These capabilities open up a world of possibilities.

We’ll get to the use of the variable in a moment, but first we focus on the two lists: one for notes and one for rhythms (or perhaps the second list might be better named durations). Each was populated by importing values from a corresponding plain text file. It is important to note that the two lists are the same length, such that each note had a corresponding rhythm (or duration).[[7]](#footnote-7)

|  |  |  |
| --- | --- | --- |
| (a) |  | (b) |
| Figure 6‑22. (a) The Scratch Variables panel after creating two lists named notes and rhythms and a variable named counter. (b) The notes and rhythms lists after they have been populated by importing values from corresponding plain text files. | | |

The program that accesses these lists to play *Frère Jacques* is shown in Figure 6‑23. This program uses a variable named counter to keep track of which loop it’s on. That is, the first time through the loop the counter variable is 1, the second time it’s 2, and so on. The counter variable is then used to provide access to the individual entries in the lists.

Note that you have to initialize and increment this variable yourself, because Scratch does not provide access to its internal loop counter. This is not a big deal, and as a matter of fact it is a blessing to arts majors because it makes the way the loop works more explicit. However, it is often a stumbling point for science majors who are experienced programmers, who know that there *must* be an internal counter somewhere, and who spend considerable time trying to figure out how to access it. So in this case, programming experience is a negative, not a positive! Happily, though, the science majors’ confusion seldom lasts very long, and they get the loop working pretty quickly. For both groups of students, the trick is to remember that you have to increment the variable each time through the loop. Even we sometimes forget to do that, but the beauty of learning to program through music is that one clearly *hears* the error. That is, if you don’t increment the variable, the first note will play over and over again ad infinitum. That’s clearly not what you want.



Figure 6‑23. Program to play *Frère Jacques* from MIDI values and rhythms stored in the two parallel lists shown in Figure 6‑22.

### Playing Multiple Parts

As most children know, *Frère Jacques* is meant to be sung as a round. To do that, we need to play multiple parts simultaneously using combinations of the  ,  , and  blocks to create subroutine-like structures. Using these blocks, you can “broadcast” a message that the when I receive block is listening for. When the latter block “hears” the message it’s interested in, the blocks that it sits above will be executed. The difference between the broadcast and broadcast and wait blocks is that the former sends its message and then allows the program to continue immediately on its way, kind of like when you send an email message. That is, you send an email and then go on to do something else without waiting for the recipient to respond. The latter, the broadcast and wait block, is more like a telephone call. It sends its message and does not go on until the stack of blocks under the corresponding when I receive block finishes executing. This is like waiting for the person on the other end of the line to answer before you can communicate with him or her.

To play *Frère Jacques* as a round, we first put each phrase into a separate block as shown in Figure 6‑24. Each of these phrases is also stored in a separate Scratch sprite so that they can have different volume (loudness) and timbre (instrument) characteristics, which are “local” to each sprite rather than “global” to the entire program. Each of these stacks of blocks is executed when it receives the message it is waiting for from one of the broadcast blocks. Thus, we can play the song “straight” using the code in Figure 6‑25.

|  |  |  |
| --- | --- | --- |
| (a) |  | (b) |
| (c) |  | (d) |
| Figure 6‑24. Each phrase of *Frère Jacques* broken out into its own block. | | |



Figure 6‑25. Control code to play each phrase of *Frère Jacques* shown in Figure 6‑24.

|  |
| --- |
| Figure 6‑26. Control code to play *Frère Jacques* as a round. |

Now, with the phrases separated and taking advantage of the fact that with the tempo set to 60 beats per minute each twice-repeated phrase is exactly 4 seconds long, we can code *Frère Jacques* as a round as shown in Figure 6‑26. This is not the most elegant way to code the round, to be sure, but it is a good starting point for understanding timing and sending and receiving signals and the CT concepts embodied in doing so.

### Algorithmic Music

Our uses of variables so far have been pretty simple. Things get considerably more interesting when programs are written to manipulate those variables in ways that affect the sound. Internal variables, like the one for volume, can be manipulated, too. Figure 6‑27 implements a fade by manipulating the internal volume variable each time through the loop.



Figure 6‑27. A Scratch fader.

Manipulating one’s own variables is even more interesting, and adding a  block makes things downright fun. In Figure 6‑28 we’ve coded a loop that picks a random note out of a C major scale, which is coded in a list with MIDI values: 60, 62, 64, 65, 67, 69, 71, 72. The result isn’t very musical, but it’s interesting to hear how it changes each time the program is run. In addition, this little program can open up fascinating discussions about *why* the result isn’t very musical and how you might manipulate the algorithm to make it more musical.



Figure 6‑28. A program that plays ten notes chosen randomly from the C major scale.

### Connecting to External Devices

Scratch also supports connections to external devices, specifically the PicoBoard [[13](#_ENREF_13)] and the IchiBoard [[6](#_ENREF_6)]. These devices contain a number of sensors and components that Scratch knows about and has built-in capabilities for reading data from. The various sensors and components send messages and data values to Scratch that it can use within programs just like values stored in variables.

Figure 6‑29 shows two views of the IchiBoard, one a photograph (a) and the other a block diagram (b). The main components are:

* a slider that returns position values of 0-100
* a sound that returns volume values of 0-100
* a light sensor that returns brightness values of 0-100
* a button that returns true if it is pressed
* X, Y, and Z axis accelerometers that return values of -50 to +50
* four ports that take mini RCA plugs for additional connections, such as probes that measure resistance

|  |  |  |
| --- | --- | --- |
| (a) |  | (b) |
| Figure 6‑29. The IchiBoard. | | |

Our students use the IchiBoard to build instruments. For example, one of our students built an electronic tympani from a simple drum head (Figure 6‑30a). The IchiBoard is attached to the drum head stand, and he manipulates the slider with a string. Scratch reads two of the IchiBoard sensor values: the sound (volume) sensor value and the position of the slider (Figure 6‑30b). The sound sensor controls when Scratch triggers a note whose pitch is based on the slider position. To hear the result, listen to the first part of the video at youtu.be/9Sec6tAZsuM.

|  |  |  |
| --- | --- | --- |
| D:\Pictures\UMLCS\2010-04-29_SoundThinking_119Gallery\images\byAnne\0040_IMG_4418.jpg  (a) |  | (b) |
| Figure 6‑30. An IchiBoard tympani (a) and the code to drive it (b).  (*Photo courtesy of Anne Ruthmann Photography.*) | | |

In the second part of that video, another student uses simple alligator clip probes attached to the IchiBoard external sensor inputs to play different tones based on the measured resistance across different pieces of fruit (Figure 6‑31a) and even a classmate’s face (Figure 6‑31b)! Her code is very similar to the tympani code, but the change in sensor and the small changes to the parameters make a world of difference in the sound produced (Figure 6‑31c).

|  |  |  |
| --- | --- | --- |
| (a) |  | D:\Pictures\UMLCS\2010-04-29_SoundThinking_119Gallery\images\byAnne\0050_IMG_4434.jpg  (b) |
| (c) | | |
| Figure 6‑31. Using the IchiBoard sensors to play fruit (a) and a classmate’s face (b), along with the code being run (c). (*Photos courtesy of Anne Ruthmann Photography.*) | | |

### Live Coding

Far more complex music can be created, of course, with multiple parts each playing different instruments, mixing recorded and generated music, and incorporating changes that occur based on user input. For example, a sound might get softer or louder as the mouse moves left and right or up and down, or its pitch may increase or decrease when certain keys are pressed.

These types of interactive capabilities are the basis of live coding, the ultimate use of Scratch to generate music. Brown and Sorensen [[4](#_ENREF_4)] define live coding as follows:

The practice of live coding involves writing and modifying computer programs that generate music in real-time. Often this music making activity occurs in a live performance situation with the code source projected for the audience.

Scratch makes live coding possible because the blocks are interpreted as the program runs. Therefore, adding or removing a block or changing its parameters in a running program that is playing music will affect the sound being produced in real time. Live coding is extremely fluid, making it nearly impossible to demonstrate in the confines of a book. We therefore refer the reader to the following URLs, where videos of live coding in Scratch are posted.

http://www.youtube.com/watch?v=HyMXCZXjwWg

* code and live performance by UMass Lowell Prof. S. Alex Ruthmann

http://www.youtube.com/watch?v=rDyo4p1qLuE

* code and live performance by MIT Media Lab graduate student Eric Rosenbaum using a prototype of Scratch 2.0

### Scratch, Music, and CT

We hope that our short tour of Scratch’s music generation capabilities has demonstrated the richness of what can be done with a system that appears, on the surface, to be relatively simple. To put the extent of the concepts we try to teach with our assignments in perspective, we created the lists in Table 6‑1 that enumerate all the computer science and music concepts our students are exposed to.

|  |  |
| --- | --- |
| Table 6‑1. Computer science and music concepts covered using Scratch. | |
| **Computer Science**   * statements * sequential control flow * iteration * conditional execution * arithmetic operators * Boolean operators * objects * concurrency * variables * lists * event handling * user interaction * optimization | **Music**   * pitch * rhythm (as duration) * melodic fragments * modes and scales * polyphony * synchronization * harmony * composing * performing * transposition * balance and dynamics * digital audio (as sound files) * MIDI notes and timbres * tempo * form and structural analysis |

Many students easily make the jump from these concepts to CT skills, but some do not. The types of CT we are referring to include breaking problems down into their components and attacking those one at a time, analyzing alternatives when things go wrong or don’t produce the expected results, and creating reusable code and data structures so that they can be used again in another program. Rather than confront these deeper (or higher level) thought processes, some students just do their assignments in a mechanical way, just trying to do what they’re told so that they can get the assignment finished as quickly as possible. The student reflections we have quoted throughout this book demonstrate our approach to combatting these common student tendencies and getting them to think about why a particular assignment was made or what they got out of it. We’re sure that virtually all teachers will agree that it’s a constant battle.

We have found, however, that letting students create and manipulate their own music using easily approachable tools like Scratch, having them work in interdisciplinary teams that foster interesting discussions and make help readily available for areas they’re not familiar with, and encouraging them to release their own inner creativity all contribute to creating a fertile environment in which the development of CT skills can flourish.

## Bibliography for Chapter 6

[1] Bailey, M. (2011). *Bio*. www.mattbailey.info/MBI/Bio.html *accessed* 11/14/2011.

[2] Baxter-Magolda, M.B. (1999). *Creating Contexts for Learning and Self-Authorship*. Nashville, TN: Vanderbilt University Press.

[3] Brown, A.R. (2005). *Making Music With Java*. South Bank, Queensland: Lulu (self-published).

[4] Brown, A.R., & Sorensen, A. (2009). *Interacting with Generative Music through Live Coding.* Contemporary Music Review **28**(1):17-29.

[5] Bwarie, J.L. (2011). *About Joseph Leo Bwarie*. josephleobwarie.com/bio *accessed* 11/14/2011.

[6] Engaging Computing Group (2012). *IchiBoard*. http://www.cs.uml.edu/ecg/index.php/IchiBoard/IchiBoard *accessed* Jan. 11, 2012.

[7] Franklin, J. (2011). *Resume*. www.joshfranklin.net/JoshFranklin.net/Resume.html *accessed* 11/14/2011.

[8] Georgia Tech School of Music (2011). *Prospective Graduate Students*. www.music.gatech.edu/ prospective\_students/graduate *accessed* 11/14/2011.

[9] Gouveia, S. (2011). *RE: Looking for information on your training*. Personal correspondence, 12/12/2011.

[10] McCartney, J. (2011). *SuperCollider*. supercollider.sourceforge.net *accessed* Dec. 23, 2011.

[11] Papert, S. (1980). *Mindstorms: Children, Computers, and Powerful Ideas*. New York: Basic Books.

[12] Papert, S. (1993). *The Children's Machine: Rethinking School in the Age of the Computer*. New York: Basic Books.

[13] Playful Invention Company (2012). *PicoBoard - Sensor Board that works with MIT’s Scratch*. http://www.picocricket.com/picoboard.html *accessed* Jan. 11, 2012.

[14] Princeton University (2011). *Daniel Trueman, Lewis Center for the Arts*. www.princeton.edu/arts/ arts\_at\_princeton/music/professor\_bios/trueman/index.xml *accessed* 11/19/2011.

[15] Resnick, M. (2007). *All I really need to know (about creative thinking) I learned (by studying how children learn) in kindergarten*. Proceedings of the 6th ACM SIGCHI conference on Creativity & Cognition, pp. 1-6. Washington, DC, USA: ACM.

[16] Resnick, M., Maloney, J., Monroyhernández, A., Rusk, N., Eastmond, E., Brennan, K., Millner, A., Rosenbaum, E., Silver, J., Silverman, B., & Kafai, Y. (2009). *Scratch Programming for All.* Comm. of the ACM **52**(11):60-67.

[17] Smallwood, S., Truman, D., Cook, P.R., & Wang, G. (2008). *Composing for Laptop Orchestra.* Computer Music Journal **32**(1):9-25.

[18] Truman, D., Cook, P., Fiebrink, R., & Snyder, J. (2011). *PLOrk: Princeton Laptop Orchestra*. plork.cs.princeton.edu/ *accessed* 11/19/2011.

[19] Truman, D. (2011, as quoted by Jacqui Cheng). *Musicians, Tune Your Keyboards: Playing in a Laptop Orchestra*. arstechnica.com/gadgets/news/2011/07/laptop-orchestras-what-are-they-and-where-did-they-come-from.ars *accessed* 11/14/2011.

[20] Wilson, S., Cottle, D., & Collins, N., eds. (2011). *The SuperCollider Book*. MIT Press: Cambridge, MA.

1. As of this writing, the price for a permanent license with unlimited upgrades was $59, but of course you should check the website for current pricing. [↑](#footnote-ref-1)
2. This code was developed by Brendan Reilly, one of our undergraduate computer science research assistants. [↑](#footnote-ref-2)
3. Ubuntu is a free, open-source operating system for PCs (www.ubuntu.com/download) that many of our CS students prefer to Microsoft Windows due to its speed (particularly on small computers) and the abundance of free apps. [↑](#footnote-ref-3)
4. Real subroutines are planned for Scratch 2.0 and are already available in an “advanced offshoot” of Scratch called BYOB (for “Build Your Own Blocks,” which is also known as “Snap!”) developed by Jens Mönig and Brian Harvey at the University of California Berkeley. See byob.berkeley.edu. [↑](#footnote-ref-4)
5. “Looping constructs” are programming techniques that cause groups of statements to be repeated. In Scratch, these are the blocks that enclose other blocks and cause the enclosed blocks to be executed a number of times (the repeat block in the Scratch Control panel), while a condition is true (the forever if block), or until a condition becomes true (the repeat until block). [↑](#footnote-ref-5)
6. Strings of characters are series of letters. In addition, programmers typically refer to a word or a phrase as a “string.” The fact that lists can contain strings is important because this allows programs to use data other than just numbers. For example, you might have a list that contains “do,” “re”, “mi,” etc. and display those syllables when the corresponding notes are played. [↑](#footnote-ref-6)
7. *Note to geeks:* Two lists are needed because Scratch does not support two-dimensional arrays . [↑](#footnote-ref-7)