BlockyTalky: A Physical and Distributed Computer Music Toolkit for Kids

R. Benjamin Shapiro
Annie Kelly
University of Colorado Boulder
Boulder, CO, USA
ben.shapiro@colorado.edu
annie.kelly@colorado.edu

Matthew Ahrens
Tufts University
161 College Ave
Medford, MA USA
matthew.ahrens@tufts.edu

Rebecca Fiebrink
Goldsmiths, University of London
London, SE14 5PP
United Kingdom
r.fiebrink@gold.ac.uk

ABSTRACT

NIME research realizes a vision of performance by means of computational expression, linking body and space to sound and imagery through eclectic forms of sensing and interaction. This vision could dramatically impact computer science education, simultaneously modernizing the field and drawing in diverse new participants. We describe our work creating a NIME-inspired computer music toolkit for kids called BlockyTalky; the toolkit enables users to create networks of sensing devices and synthesizers. We offer findings from our research on student learning through programming and performance. We conclude by suggesting a number of future directions for NIME researchers interested in education.

Author Keywords

NIME, computer science education, distributed systems

ACM Classification

• **Applied computing~Sound and music computing** • *Social and professional topics~Computing education programs*

1. INTRODUCTION

Inherent in NIME research is the vision that computational practices can be rich with creative expression as well as technological ingenuity. The NIME community investigates and demonstrates how new computational systems can enable musical expression and interaction. Though this community has impacted the state-of-the art of music composition and performance, its innovations have not yet become a widespread part of computer science education.

We believe that computing education could benefit in several ways from integrating topics central to NIME into computer science (CS) curricula, and from adopting the NIME community's understanding of computing as an expressive, creative domain.

First, computer music is a uniquely strong application domain through which we could modernize computer science curricula. Computing education research scholars as well as industry have bemoaned the absence of contemporary computing topics like distributed systems and concurrency from computer science education [15]. Recent standards documents like the ACM-IEEE Computer Science Curricula 2013 [1] mandate these topics' inclusion within CS education pathways, but compelling examples of how to do so are currently lacking. Yet is is rare to find computer music performances that lack timing-sensitive distributed computing, concurrency, and networking. Games also have many of these computational facets, but past research shows that games disproportionately motivate boys to program [11]. Computer music offers underexplored potential to motivate people from all backgrounds to learn computer science [7].

Next, computer music also offers us a chance to combat pervasive and insidious misperceptions of computer science and computer scientists. American students and adults have negative stereotypes of science and scientists, often believing that they are socially distant, dangerous, workaholic, peculiar, irreligious, and missing fun in their lives [12, 14]. And they tend to hold similar stereotypes of computer scientists [3,10,13]. These stereotypes can be challenged by educational approaches that bring scientific methods together with topics that are relevant to youths' lives and that showcase the wide range of possible pro-social impacts of STEM, including computing [10]. Music is often both highly social and creative. CS education experiences based upon computer music could drastically change students' perceptions of computing.

Integrating music into computer science education may likewise be helpful in broadening participation in computing, which refers to increasing the currently anemic participation by women, ethnic minorities, indigenous people, and the poor in CS. Broadening participation is one of the foremost goals of the computing education research community. Integrating computer music into computing curricula could improve participation both by combatting the misperceptions described above and by creating more pathways into computer science—as well as computer music itself—for students who are already musicians.

In this paper, we describe the creation of a new distributed and physical computer music systems-building and performance toolkit for kids aged 10 and up. We have designed this toolkit to enable youth to create new digital musical instruments and other interactive music systems, with the aims of engaging them with a variety of introductory computing concepts as well as challenging them to think more positively about the creative potential of computers and their own ability to do computing. We also describe the outcomes of several educational activities in which youth have used this toolkit, and the implications of these outcomes for understanding and improving education.

2. DESIGN CONSIDERATIONS

We set out to create a toolkit for kids to create computer music systems that they can use in collaborative performance. We believed that the toolkit should simultaneously enable users to construct a variety of physical interfaces (e.g., using different combinations of sensors), as well as support the combination of different technologies in distributed systems (e.g., sensors communicating wirelessly with software sequencers that control sound synthesis algorithms).

This basic design concept is common at NIME yet different from any which animates other programming tools designed for youth. For example, consider Scratch, a hugely popular software system for kids to use to create games, stories, and animations. It is hugely popular: 185,000 unique users created new Scratch projects in December 2015 [16]. Scratch runs in a browser and enables users to write expressive programs by dragging pieces of code onto sprites in order to define their behaviors. Scratch allows users to play sounds and even to build modest physical interfaces (via MaKey MaKey), but the sound programming features are much more geared toward adding sound to games than they are to composing and performing music. EarSketch [7] is another educational programming

environment, and one that combines music with CS education. It too is browser-based and enables users to programmatically sequence and manipulate synthesized sounds and samples. However, it is not designed for collaborative real-time performances or for using physical input to shape sound synthesis.

Though there were aspects of tools like Scratch and EarSketch that we wished to replicate (e.g., browser-based interfaces, which drastically reduce the complication of system use in schools), we also recognized that we would need to borrow heavily from design and engineering patterns that are common in NIME but that have not yet been adapted for younger users. As engineers, our goal was to create the first computer science education toolkit that would adapt core NIME approaches for young learners. That is, we wanted to create systems that could use usable by youth 10 years old and older to create:

- software synthesizer programs that offer real-time control over their parameters
- sequencers that sequence those synths
- hardware, especially sensors that get information about people in the world
- networked systems: distributing sensing and sound-making, but also supporting multiple people making sound together on multiple devices that can communicate and synchronize.

As educators, we wanted to empower those kids to:

- make choices about hardware design
- write network protocols to link system elements together
- work with constraints in systems that shape their activity, such as latency in sensing, communication, and synthesis
- experiment with different types of expressive control
- improvise during performance through use of physical inputs or modification of code.

3. IMPLEMENTATION

Over the past two years we have created and iteratively refined the BlockyTalky toolkit, which meets all of the above goals. BlockyTalky is open source and runs on low-cost single-board computers like the Raspberry Pi. Typically these devices are equipped with "shields" or "capes" that allow use of child-friendly sensing hardware (e.g., LEGO Mindstorms sensors).

Each BlockyTalky device runs a server that provides users a complete web interface for configuring and programming its hardware. Drop-down menus enable users to declare what kinds of sensors are connected to which input port and the system provides real-time sensor readings to help users to plan, monitor, and troubleshoot their designs (see Figure 1).
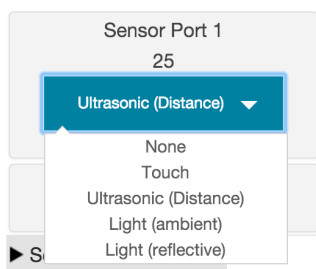

Figure 1 Input selection menus with real time sensor readings

A variety of programming blocks enable users to define musical motifs (including synthesized notes, samples, and effects), to send messages between devices, and to create event-handlers for inputs received from physical sensors or messages received over the network. The programming model assumes that users will typically compose and enact performances using several devices at once, with some devices handling sound synthesis and others handling physical inputs from users; user-created asynchronous messaging protocols

coordinate activity across these devices. Figure 2 shows a typical configuration and what code for that configuration looks like.
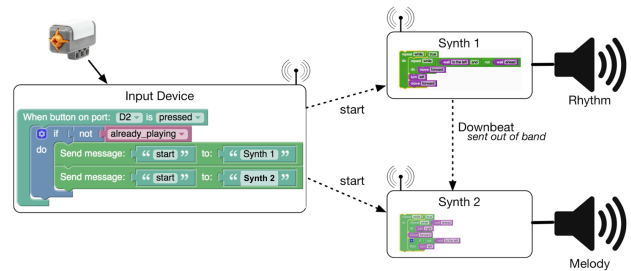

Figure 2 Common project architecture and code

Users can configure BlockyTalky synthesizers to synchronize their clocks to one another with a single programming block (*Sync my clock to Foo*). Then they can use *wait for* blocks to specify the timing of note synthesis or sample playback; common values of *wait for* (e.g., a downbeat or the first note in a 4-count) across multiple synthesizers result in synchronized playback.

The block-based programming interface is implemented using Google's Blockly toolkit [9]. Users' block programs are transpiled into a textual Domain-Specific Language (DSL). The DSL provides convenient abstractions around common complexities for physical computing and network programming. For example, the block program:
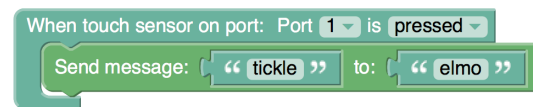
becomes

```
when_sensor "PORT_1" == 1 do
    send_message("tickle", "elmo")
end
```

The `when_sensor` macro handles management of state that is necessary to detect and dispatch events, while the `send_message` function handles peer discovery, serialization, and transmission of messages over UDP.

This functionality is implemented in JavaScript and the Elixir functional programming language, which runs on the Erlang virtual machine. Erlang's actor-based architecture enables us to quickly add new capabilities to the system, such as support for new hardware, networking protocols, and user interfaces. The Phoenix web framework (which serves both static web content and streaming realtime communication over WebSockets) enables us to provide live information to users about device and sensor states and network communications.

4. EXPERIENCES WITH YOUTH

We have used BlockyTalky with youth in a variety of different educational settings, including two multi-week computer music summer camps, short 1–2 hour workshops, and in 5- to 10-minute interactions during outreach events.

The summer camps were located on the premises of existing neighborhood-based youth-serving organizations in a large Northeastern U.S. city, and were offered as part of those organizations' existing summer activities. We chose to do our research in conjunction with these community partners in order to maximize the likelihood of including low-income minority youth within our study (low-income parents frequently report difficulty getting their children to the university campus) and involving participants who already have social relationships with one another (facilitating creative collaboration). Each of the summer camps lasted about twenty hours in total. Both

camps were taught by university faculty and students, with logistical support from the community organizations. Both camps were structured such that participants spent their time working toward a culminating computer music performance, with the university-based researcher-teachers leading tutorial activities on how to use the BlockyTalky system and offering just-in-time support as students had questions or ran into obstacles.

We documented camp sessions exhaustively, placing numerous cameras throughout the room to capture student conversation on audio and video, and configuring BlockyTalky to retain time-stamped copies of all saved changes to students' code.

4.1 Camp One

When we launched the first camp—which primarily involved African-American 11–13 year olds—we did not yet know what aspects of our computer music approach would be most and least challenging for participants. We chose to implement a lightweight curriculum, and then use our observations of student participation to guide curriculum design for subsequent iterations. Accordingly, on Day 1 we presented demos of instrument designs to the campers, then orchestrated whole-class activities around representing musical structures symbolically (e.g., clapping out rhythms and drawing them on paper). On Day 2 we handed out pre-made demo systems and encouraged students to modify them to taste. Then, beginning on Day 3, we told students to tinker and play with the BlockyTalky toolkit, creating systems to play songs that they liked. Then, as the remaining 6 days of camp progressed we increasingly urged students to start creating sets of systems that they could use in group performances. As they did so, we provided just-in-time support in response to problems they encountered or questions they had. The camp concluded with a performance day, wherein all student groups demonstrated their work. Students created a variety of projects, including:

- A 3-student Star Wars Cantina Band, including two electronic trumpets and a bass line. Each trumpet had 3 buttons, used to choose a note pitch, and a sound-volume sensor, used as a breath sensor so that the instrument would only make sound when the user blew into it. Each trumpet communicated over the network with a shared synthesizer. However, the 11–12-year-old African-American boys who built this system did not use the system's synchronization functions, and so their ultimate performance was disjointed.

- An interactive karaoke-like machine shaped like a clarinet that played all of the verses to John Legend's *All of Me*, and accompanied the girl who built the device while she sang the song and controlled the device by pressing buttons to make it change verse.

The *All of Me* machine was the only one of the five student projects that used programmatic features to synchronize multiple synthesis tracks. Follow-up cognitive clinical interviews [8] with students revealed that they were confused about how the synchronization aspects of the BlockyTalky system worked, even while they understood the distributed nature of the tools.

We surveyed students about their experiences, asking them to write free-form responses to three prompts: 1) The best thing about this workshop was… 2) I would improve this workshop by… 3) Describe, list, or draw 3 things that you learned at this workshop. Their responses (n=10) touched on a number of facets of the project. Through Open Coding [4] we identified several common themes: Making, Music, Programming, Learning, Products, and Social Interaction. Making, Programming, and Music were, by far, the most common themes in students' responses (two-thirds of students' responses

included at least one of these themes). For example, in response to Question 1 (The best thing was...), four surveys mentioned Making (e.g., "That we got to use legos to build any musical instrument that we wanted to"), two mentioned Programming (e.g., "learning how to program"), and three mentioned Music (e.g., "Getting my instrument to play the song it took a while but it turned out to be a great product"). The small number who mentioned music surprised us, given the content of the workshop. Responses to Question 2 (I would improve this workshop by...) were the most variable, with only five (half) of the responses involving these themes; suggestions included "adding more motor ports and sensor ports" (coded as about Making) and "making more programs to fix" (coded as about Programming). No responses to Question 2 mentioned Music, and four were left blank or said variations on "I don't know". In response to Question 3 (3 things you learned...), 9 of 10 respondents provided an answer; 8 of those mentioned Programming and 6 of them mentioned Making.

In summary, the activities successfully captured the interest of nearly all participants, and students strongly expressed a belief that they had learned some programming. All students were able to create interactive networked musical systems, and interview participants understood the basic distributed computation-related learning goals that we hoped they would. However, we saw no evidence that students understood the synchronization mechanisms that are part of the system's design.

4.2 Camp Two

A year after Camp One, we conducted a second summer program, this time with first-generation Asian immigrant girls, ages 11–14. To address the weaknesses we observed in the first camp, we took a more structured approach in the second camp, opening all but the last few days of the camp with whole-class activities, including "CS Unplugged"-style activities wherein students acted out various computational processes, including pattern matching and messaging. In the early days of the camp, we complemented these whole-class activities with small-group tutorials in how to use the BlockyTalky system, and encouraged students to tinker with it in order to make small musical demos (e.g., one group used motors to move an egg-shaker back and forth while using another motor to strike a chime). These small projects responded to students' desires to combine computer music with acoustical instruments, and helped them to learn the BlockyTalky programming environment.

After several days of this, participants began the process of developing these systems by authoring project visions during the participatory design activity described above. These vision posters described what songs the girls would program, the girls' roles in the performances, what sensors they would use, and what the role of each team member would be in the performance. The next day, students formalized these posters into project specifications that included drawings of their planned system topologies, documentation of messaging protocols, and timelines of their planned performances.

Camp Two participants designed, programmed, and performed a wide variety of pieces, including:

- A project by a group of first-generation immigrant girls to play *Flashlight* by Jessie J. It consisted of two synthesizer devices, `Synth2` and `Synth9`, each programmed with a single motif (`Motif A` and `Motif B`, respectively) plus a BlockyTalky device configured with a single push button. In their implementation, when the button was pressed the BlockyTalky would send a message to trigger `Synth2` to play `Motif A`, then at the end of the motif, `Synth2` would send a message directly to `Synth9` to trigger `Motif B`.

The girls intended to program the two synthesizers to play two additional synchronized motifs, but ran out of time.

- A system by another group of girls to play *See You Again* by Wiz Khalifa. They used two synthesizers, `Synth3` and `Synth7`, together with a BlockyTalky device that they configured with three push buttons. The girls broke the song into three parts, an **I**ntro, a **M**elody, and a **T**ag, and programmed motifs for these parts onto the two synthesizers. They then wrote code on the BlockyTalky that would send messages to the appropriate synthesizers to trigger these parts when corresponding buttons were pressed. They rehearsed a performance sequence of IMTMTT, in which one student triggered the Intro, another triggered the Melody, another triggered the Tag, and another cued each team member on their turn.

As part of their work building such systems, we have seen students exhibit impressive evidence of understanding the relationships between the various system components that together constitute a distributed system. That is, the students were able to work backwards from performances that they envisioned in order to design assemblages of system components that could serve their expressive goals. We saw evidence of this in their final projects, in their classroom discourse, and in artifacts of their project planning activities. For example, Figure 3 shows how one group of students sketched a system diagram for one of their projects before building it, and then documented messages that would be exchanged between nodes in that system.
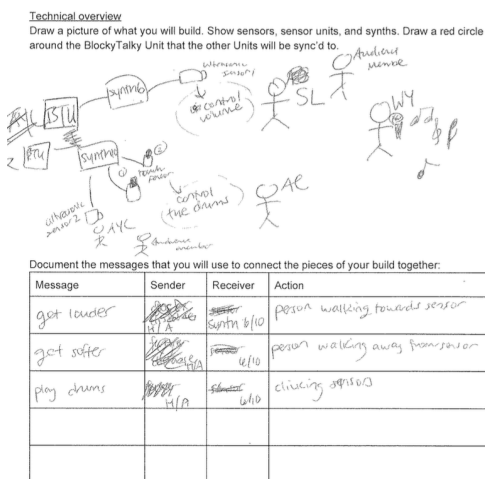
Figure 3: System diagram and protocol documentation

As is evident from the above descriptions, these projects all involved programming multiple nodes to communicate with one another and, in two of the three cases, students' designs included synchronization across multiple nodes. Because of these design and implementation details, we believe that this iteration was more successful than Iteration 1 in enabling youth to learn about concurrent and distributed computing concepts.

This assessment of student learning based upon students' projects is triangulated by interviews we conducted with students at the end of their participation in the program. These interviews focused on a card-sort and a story telling exercise. In the card sort, the girls were challenged to work with peers from different teams to group cards bearing BlockyTalky-related vocabulary into whatever arrangements seemed sensible to them, and then to explain their groupings. Next, we asked the girls to use the cards to tell stories that made sense to them. These stories offered us a rich window into students' understanding not only of the definitions of the vocabulary on

the cards, but how the functional characteristics of the concepts would be operationalized. For example, one pair of students wrote the following (card-words in boxes):

‘A’ uses a computer to program a BlockyTalky device. ‘A’ codes the NetJam to control sensors. ‘A’ designs events in a browser using blocks. ‘B’ sends a ★ message to cue motif #1. ‘C’ receives a ★ through the network to change the volume of note pitch. ‘D’ inputs a delay/lag in effect and changes the tempo output. ‘E’ uses the synthesizer for synchronization with the rhythm. ‘F’ uses communication to send a ★ to ‘G’ to play a loop beat using a Lego motor. ‘H’ taught them all using speakers and instrument in a computer science class.

This story reflects the two students' understanding of two key CS conceptual goals for our project. They described a system that is composed of multiple nodes, each running a program, and coordinated with one another via messaging. It is unclear what the precise meaning of the sentence with "synchronization" in it is, though it does seem to reflect that the two students understand the system's synchronization function as a way to have a common rhythm across BlockyTalky nodes. Other students' stories were similarly sensible.

We used the same coding scheme as in Iteration 1 to analyze students' survey responses (n=10) for this iteration. In response to Question 1 (The best thing was...), no surveys mentioned Making, two mentioned Programming (both did so in conjunction with Music, e.g., "Learning how to code and make music"), and seven mentioned music (e.g., "putting the song together"). Two responses were coded as about the Social Interaction characteristics of the activity ("working with new people" and "I was able to step out of my shell and communicate with others. I was also able to work as a team with other people.") Question 2's (I would improve this workshop by...) responses were again the most variable, with only four of the responses touching on Music, Making, or Programming; suggestions included "having all of us work together and have more time" (coded as about both Pedagogy and Social), "learning my music notes" (coded as about Music), and "learning more coding" (coded as about Programming). No responses to Question 2 mentioned Instruments, and four were left blank or said variations on "I don't know". In response to Question 3 (3 things you learned...), 9 of 10 respondents provided an answer; 8 of those mentioned programming and 6 of them mentioned Making. Social factors were also mentioned here, such as "Team work" and "Collaboration isn't always easy."

In summary, Camp Two saw a significant improvement in the richness of students' system designs, implementations, and understandings of computational ideas as compared to Camp One, while maintaining a high level of student positivity about the experience.

5. CHALLENGES AND SOLUTIONS
5.1 Challenges
Over the course of the many workshops we have run with youth, we have also learned several lessons that have informed our iterative design and development practice, and that should inform future work in this area. Here we describe some of the challenges we have encountered and how we have addressed them or plan to address them in the future. The first two challenges involve leveraging students' prior knowledge and

skills, an important dimension of any learning environment design [2], while the latter are more technical in nature.

Challenge 1: Adapting novice musicianship to programming

Some of our participants had prior experience playing music, such as through guitar classes at the community center that hosted Camp One. However, those students found it remarkably hard to draw upon that prior knowledge in order to program BlockyTalky. For instance, one pair of boys wanted to program Deep Purple's *Smoke on the Water*, a song they could sort-of play on guitar. The challenge they faced was in translating their embodied knowledge of how to play the song into the symbolic representation needed by the BlockyTalky system. That is, their knowledge of how to play the song was of the form of *first put your fingers like this, then put your fingers like that*, but the BlockyTalky system expects input in the form of *play a C, then play a G*. Resolving this challenge required teaching staff to translate between finger positions and pitch names (this case is described in detail in [5]).

Challenge 2: The pop music re-creation trap

Many students who have participated in our work have been excited to re-create pop songs that they are familiar with. We found this to be a double-edged sword: On one hand, it is motivating to students and provides an easy way to make something that sounds good even without formal knowledge about music composition. On another hand, we frequently found this to be a hugely time-consuming dead-end for students. They tended to get focused on one-to-one replication of pop music, and they did not explore re-arrangement or improvisation around melodies and rhythms in their chosen songs. Their work focused more on fidelity of re-creation rather than creative expression. Because re-creations were totally linear, students doing them tended not to explore programmatic methods for synchronization of musical motifs that would be useful in improvisation, thereby limiting their CS learning.

We have recently begun addressing Challenge 1 and Challenge 2 through a co-design partnership with two middle school teachers (one a music teacher, the other a math teacher). In this work, which is still ongoing, we are creating a computer music composition and performance curriculum that teaches students to algorithmically compose melodies, harmonies, and counterpoint, and through doing so learn about the mathematical concept of functions and the computational concept of state machines. Along with the curriculum, we are adding additional music programming blocks to enable students to program with finger numbers, rather than explicit pitches (e.g., *play 1, 3, and 5 in C-major*). This will enable students to more easily tinker with melodies at the piano and then translate them to code, as well as facilitate learning about functions (*1, 3, and 5 can map to different pitches depending upon the key, and this mapping is a function*) and state machines (a simple classical harmony can transition from a IV to a V but not from a V to a IV). Students and teachers in two middle schools will test out this curriculum beginning in late February 2016. We will evaluate how this approach empowers students to better understand the melodic structure of songs that they already know (addressing Challenge 1) and to more creatively explore musical possibilities for their performances (Challenge 2), as well as deepen connections between computer music and CS and math education.

Challenge 3: Latency

In order to minimize engineering effort, leverage prior work, and create pathways from our beginner-specific tools into more general-purpose systems, we have built as much of BlockyTalky around existing open source software as possible.

Our current iteration of the tools use Sonic Pi for sound sequencing and synthesis. In general, Sonic Pi works well, including on the relatively low-end hardware of the Raspberry Pi. However, Sonic Pi is designed for live coding, not for performances that involve realtime control over sequencing and synthesis. It maintains a long buffer in order to avoid skipping. However, this means that there is often a lengthy delay, up to a second, between when a user manipulates a sensor and when the sound they hear changes. This can be confusing to users. The Sonic Pi project has no concrete plans to address this problem.

We also suffer from latency problems caused by our use of electronics construction materials that are not optimized for timing. For example, BlockyTalky's support for LEGO Mindstorms sensors depends upon using the Dexter Industries BrickPi shield. The drivers and firmware for the BrickPi require constant polling of the board in order to detect changes to input state, an operation that is both slow and power-intensive (the largest source of power drain in our system).

Challenge 4: Cost

Our current hardware setup, involving a Raspberry Pi, Brick Pi, MicroSD card, USB WiFi dongle, and power supply costs about $169. The LEGO sensors needed to do anything useful with this add even more cost (a simple push button costs about $30). This price point is too high for widespread adoption of our approach.

We plan to address both of these problems by porting BlockyTalky to run on the recently released BeagleBone Green (BBG). The BBG is based on the BeagleBone Black, a board which is increasingly being used by the NIME community due to its relatively fast processor (compared to the Pi) and very low latency I/O co-processors. However, the BBG removes the Black's HDMI port in favor of two Grove connectors. The Seeed Studio Grove platform includes a massive variety of modular sensors that are well suited for building computer music devices. BlockyTalky already boots on the BBG, and we will add support for its Grove ports in Summer 2016. This will both dramatically reduce cost (<$50/device, plus sensors) and hardware latency. We will address synthesis latency by replacing Sonic Pi with a synthesis engine better geared toward realtime performance, such as ChucK.

5.2 Next Steps

As our work with BlockyTalky progresses, it will increasingly be necessary to investigate how to enable users to expand on the system's built-in capabilities through connection to other computer music software and hardware. We will soon be adding support for the OSC protocol to enable this integration.

We also plan to add support for Bluetooth communication with the BBC & Microsoft Micro:bit, a low-cost ($7) programmable board that has an ARM processor, a 5x5 LED array, accelerometers, and Bluetooth Low Energy. The Micro:bit offers an unprecedented opportunity to investigate how youth can create wearables to use to link dance and other human motion to computer music. We are currently prototyping this functionality.

6. OPEN QUESTIONS FOR NIME

The BlockyTalky hardware ecosystem is robust, cheap, and low-latency enough to support a range of rewarding and engaging music-making activities. The drag-and-drop programming environment is also a usable tool for kids with no programming background, provided that they are open to experimenting with a wide variety of sounds and effects or that they have the musical knowledge to be able to translate their ideas efficiently into symbolic representations (e.g. lists of note names and durations). However, when the educational aims are

to teach about musical instrument building, creative expression, design, and collaborative music performance—rather than about procedural programming or musical note reading—might other modes of software design be better suited to these aims? For instance, previous work has shown that building new musical instrument mappings using supervised learning—providing examples of human motions along with the musical outcomes to match those motions—can facilitate a more efficient, satisfying, and embodied approach to design, (compared to programming) for professional composers [6]. Might such techniques also allow kids to translate their musical instrument ideas into real systems? Or, might techniques for symbolic transcription of sung melodies or automatic harmony generation [17] speed up the process of "writing" programs that mimic pop songs? Might helping kids easily realize the creative limitations of mimicry at an earlier stage of their work with technology encourage them to explore new ideas?

Setting aside the question of how to embed NIME topics into computer science education, what should *NIME education* look like for youth? How could (or should) music education itself change to incorporate NIME ideas and practices? The potential benefits of expanding musical curricula to encompass computer music topics range from increasing the relevance of music education to youth who are most excited about musical genres that rely heavily on digital production practices, to facilitating music-making by youth with disabilities through bespoke digital instruments, to making a politically expedient argument for supporting music education because of its STEM content. But the risks include suggesting that music education is valuable only insofar as it aids in teaching "serious" or "economically important" STEM subjects, or exacerbating disparities between well-off schools with ample resources to invest in digital music equipment and those without them. We are excited about the potential benefits of early NIME education despite these risks, and one of our research aims is to engage the NIME community more broadly in these questions.

7. ACKNOWLEDGEMENTS

8. REFERENCES

[1] ACM/IEEE-CS Joint Task Force on Computing Curricula. 2013. Computer Science Curricula 2013. ACM Press and IEEE Computer Society Press. DOI: http://dx.doi.org/10.1145/2534860

[2] John D. Bransford, Ann L. Brown, and Rodney R. Cocking. 1999. *How people learn: Brain, mind, experience, and school*. National Academy Press.

[3] Lori Carter. 2006. Why students with an apparent aptitude for computer science don't choose to major in computer science. In *Proceedings of the 37th SIGCSE technical symposium on Computer science education* (SIGCSE '06). ACM, New York, NY, USA, 27-31. DOI=http://dx.doi.org/10.1145/1121341.1121352

[4] Juliet M. Corbin and Anselm Strauss. 1990. Grounded theory research: Procedures, canons, and evaluative criteria. *Qualitative sociology* 13.1, 3-21.

[5] Elise Deitrick, R. Benjamin Shapiro, Matthew P. Ahrens, Rebecca Fiebrink, Paul D. Lehrman, and Saad Farooq. 2015. Using Distributed Cognition Theory to Analyze Collaborative Computer Science Learning. In *Proceedings of the eleventh annual International Conference on International Computing Education Research* (ICER '15). ACM, New York, NY, USA, 51-60. DOI=http://dx.doi.org/10.1145/2787622.2787715

[6] Rebecca Fiebrink. 2011. *Real-time human interaction with supervised learning algorithms for music composition and performance*. PhD dissertation. Princeton University, Princeton, NJ.

[7] Jason Freeman, Brian Magerko, Tom McKlin, Mike Reilly, Justin Permar, Cameron Summers, and Eric Fruchter. 2014. Engaging underrepresented groups in high school introductory computing through computational remixing with EarSketch. In *Proceedings of the 45th ACM technical symposium on Computer science education* (SIGCSE '14). ACM, New York, NY, USA, 85-90. DOI=http://dx.doi.org/10.1145/2538862.2538906

[8] Herbert P. Ginsburg. 1997. *Entering the Child's Mind*. Cambridge University Press.

[9] Google Developers. 2016. Blockly. Retrieved January 29, 2016 from https://developers.google.com/blockly/?hl=en

[10] Shuchi Grover, Roy Pea, and Stephen Cooper. 2014. Remedying misperceptions of computer science among middle school students. In *Proceedings of the 45th ACM technical symposium on Computer science education* (SIGCSE '14). ACM, New York, NY, USA, 343-348. DOI=http://dx.doi.org/10.1145/2538862.2538934

[11] Caitlin Kelleher, Randy Pausch, and Sara Kiesler. 2007. Storytelling alice motivates middle school girls to learn computer programming. In *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems* (CHI '07). ACM, New York, NY, USA, 1455-1464. DOI=http://dx.doi.org/10.1145/1240624.1240844

[12] Susan Carol Losh. 2010. Stereotypes about scientists over time among US adults: 1983 and 2001. *Public Understanding of Science*, 19(3), 372-382.

[13] C. Dianne Martin. 2004. Draw a computer scientist. In *Working group reports from ITiCSE on Innovation and technology in computer science education* (ITiCSE-WGR '04). ACM, New York, NY, USA, 11-12. DOI=http://dx.doi.org/10.1145/1044550.1041628

[14] Cheryl L. Mason, Jane Butler Kahle, and April L. Gardner. 1991. Draw-a-scientist test: Future implications. *School Science and Mathematics* 91, no. 5, 193-198.

[15] David A. Patterson. 2006. Computer science education in the 21st century. *Commun. ACM* 49, 3 (March 2006), 27-30. DOI=http://dx.doi.org/10.1145/1118178.1118212

[16] Scratch Statistics. 2016. Retrieved January 29, 2016 from https://scratch.mit.edu/statistics/

[17] Ian Simon, Dan Morris, and Sumit Basu. 2008. MySong: automatic accompaniment generation for vocal melodies. In *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems* (CHI '08). ACM, New York, NY, USA, 725-734. DOI=http://dx.doi.org/10.1145/1357054.1357169