Bonk: Accessible Programming for Accessible Audio Games

Shaun K. Kane, Varsha Koushik, and Annika Muehlbradt
Department of Computer Science
University of Colorado Boulder
Boulder, CO 80309 USA
{shaun.kane, varsha.koushik, annika.muehlbradt}@colorado.edu

ABSTRACT
Introductory computer programming presents a number of challenges for blind and visually impaired screen reader users. In addition to the challenges of navigating complex code documents using a screen reader, novice programmers who are blind are often unable to experience fun coding projects such as programming games or animations. To address these accessibility barriers, we developed Bonk, an accessible programming environment that enables the creation of interactive audio games using a subset of the JavaScript programming language. Bonk enables novice programmers to create, share, play, and remix accessible audio games. In this paper, we introduce the Bonk programming toolkit and describe its use in a week-long programming workshop with blind and visually impaired high school students. Students in the workshop were able to create and share original audio games using Bonk, and expressed enthusiasm about furthering their programming knowledge.

Author Keywords
Audio games; accessibility; blindness; K12; computer science education.

ACM Classification Keywords
CCS → Human-centered computing → Accessibility → Accessibility technologies.

INTRODUCTION
Learning to code is now considered a fundamental step in developing one's ability to think and solve problems, as well as opening up opportunities for learning, creative expression, and employment. As US President Barack Obama commented during 2014's Hour of Code [22],

> *No one's born a computer scientist, but with a little hard work, and some math and science, just about anyone can become one.*

IDC '18, June 19–22, 2018, Trondheim, Norway © 2018 Copyright is held by the owner/author(s). Publication rights licensed to ACM. ACM ISBN 978-1-4503-5152-2/18/06…$15.00
https://doi.org/10.1145/3202185.3202754

Figure 1. Blind and visually impaired students design and program accessible video games in a programming workshop. Image (c) 2017 National Federation of the Blind.

As ever, there is value in identifying who may not be counted within this definition of "just about anyone," and to understand what barriers that those individuals may face.

As we consider how to move from computing for just about everyone to computing for all, we encounter barriers related to the accessibility of programming tools for people with disabilities [18]. Learning how to code may have additional benefits for people with disabilities, who can empower themselves to solve some of the accessibility challenges that they may encounter in the world [17]. However, just as with any complex tool, programming languages and environments may create accessibility barriers if they are not designed to support people of all abilities.

One approach that has been shown to increase engagement and retention for novice programmers is *media computation* [9], which situates the process of learning about programming within the context of creating and sharing media such as video games [19] or animated 3D movies [14].

Teaching programming through media has been shown to increase engagement with programming tasks, including for underrepresented groups such as young girls [14]. However, these representations can present significant accessibility challenges for novice programmers who cannot access video media, including blind and visually impaired people.

When introductory programming activities are inaccessible to some learners, this not only means that they cannot participate in those activities, but may also reinforce the idea that the entire field of computer programming is inaccessible. Thus, there exists a need to create programming experiences that can be both created and enjoyed by people of all abilities. Ideally, these experiences would not be designed as a "patch" on an existing inaccessible system, nor as an isolated community that is only of interest to people with disabilities, but would instead support and engage people of all abilities [35].

To explore the potential of accessible programming tools with accessible output, we developed *Bonk,* a programming toolkit that enables novice programmers to explore computer science through developing and sharing interactive audio games. Bonk provides a simplified, scaffolded interface for creating complex audio interactions, abstracting away challenges of playing sounds and rendering speech output. Bonk programs can be rendered in various forms: as text, as audio rendered in a screen reader, or through an embedded "self-voicing" text-to-speech system. By supporting robust forms of output, Bonk embodies the approach of *accessible programming tools for accessible media.*

In the following sections, we introduce the Bonk programming toolkit and present example games and reflections from an evaluation of Bonk with ten blind high school students. The contributions of this paper are:

1) Design rationale and implementation of Bonk, a "built-accessible" programming toolkit;
2) Observations about this approach from a week-long programming workshop with blind high school students;
3) Proof-of-concept accessible games, developed by blind students, that demonstrates the types of games that can (and cannot) be made with Bonk;
4) As a secondary contribution: a case study analysis of group coding between blind programmers, enabled by Bonk's support for code sharing and remixing.

RELATED WORK

Learning Environments for Programming

Research about how to effectively teach computer science has developed for nearly 50 years [8]. The canon of computer science education research addresses many pedagogical methods, programming languages, and development tools. Our present work, however, is motivated by several key threads of CS education research.

As previously mentioned, the present work is motivated by a media computation approach [9]. This approach has been shown to increase engagement from novice programmers. In this work, we not only explore how to support computational creation of accessible media, but explore the implications of using accessible and inaccessible media in introductory programming exercises.

Our research is also motivated by the design of constructionist programming environments such as Scratch [21]. While much attention has been paid to Scratch's use of visual blocks to construct programs, Bonk draws on Scratch's approach to creating and sharing artifacts. As with Scratch, Bonk aims to be tinkerable, meaningful, and social [26].

Finally, our work is motivated by narrative-based programming tools such as Storytelling Alice [14] and Looking Glass [10]. Engaging in storytelling creates opportunities to explore topics such as object-oriented design, functions, and loops [13]. Furthermore, programming a story does not inherently require any specific medium: thus, stories can be universally designed and presented in multiple formats to engage people with a range of abilities.

Accessible Programming Tools

Researchers have explored a variety of approaches to improving the accessibility of programming tools, including programming languages that are optimized for accessibility, tools to address specific programming challenges, and software libraries for producing accessible content.

Much current research around programming for blind and visually impaired users focuses on the Quorum programming language [31], which has been designed in collaboration with, and extensively tested by, blind and visually impaired people. Quorum offers support for programming audio-based programs, although the focus is on creating more advanced programs than we explore here. APL [29] is a programming language designed by blind programmers, which offers code structures that would be intuitive for blind programmers. Torino [32] is a tangible programming toolkit that enables children to create programs by assembling physical blocks. Other researchers have explored how to create accessible versions of block-based programs, typically by adding non-visual information through audio [15,24]. Bonk complements these approaches: while our current prototype was built using JavaScript, our approach can be adapted to any language.

A second thread of research has focused on identifying challenges experienced by current blind programmers, and developing tools to address these challenges. Albusays and Ludi surveyed 69 blind programmers and found many common problems, including inaccessible IDEs, diagrams with no non-visual alternative, and difficulties navigating code and debugging output [1]. Sodbeans [30] and StructJumper [2] are tools that provide audio feedback to increase understanding while navigating program code. As our focus is on creating compelling output from users' code, our work complements this prior research.

Some prior research has explored activities that can be made accessible to blind programmers through applications such as robotics [20], chatbots [4], and data science [12]. Bonk explores the use of audio games as a medium for learning programming, and adds the additional feature of collaboratively writing and sharing code.

Audio Games

Speech, nonspeech audio, and spatial audio have been used as alternatives to visual information in video games [36]. Audio Battleship [28] and Finger Dance [23] are games that were designed by researchers to explore how to create accessible audio games, but were not targeted at a mainstream audience. Blindsight[1] and Papa Sangre[2] were audio-only mobile games that were popular among blind gamers. More recently, some audio games (e.g., Earplay[3]) have targeted a mainstream audience, focusing on contexts such as gaming while mobile and interacting with voice agents such as Amazon Alexa. These games tend to support more narrative gameplay types such as interactive stories [27]. Bonk works similarly to these prior games, but enables audio games to be created by blind novice programmers.

Accessibility and Collaborative Work

In recent years, some researchers have moved away from focusing on accessibility issues that occur between a user and computing device, and instead have focused on how technology can help people with different abilities work together. Prior research has shown that collaborative tasks may be hindered by a lack of visual feedback, or due to overhead from using a screen reader [5,6,34]. While these studies have typically focused on interactions between blind and sighted individuals, our current research complements this work by illustrating the opportunities and challenges of collaborative programming between multiple blind coders.

TOWARDS ACCESSIBLE MEDIA COMPUTATION

Our primary goal in this work is to explore programming tools for creating engaging, accessible, shareable media. In developing Bonk, we were guided by the following goals, drawn from prior research on accessible programming environments and the prior experiences of our research team:

Easy deployment. Setting up software environments can be difficult for novices, and may be especially difficult when using assistive technology [6]. Thus, the programming tool should be available on a variety of computing devices without requiring software to be installed.

Universal playback. Individuals may use a variety of assistive technologies, including screen readers, magnifiers, or Braille displays [11]. Thus, program output should be playable on different devices (PCs, mobile devices, assistive technology) and in multiple media (text-to-speech, Braille).

Expressive output. Integrating creative expression into programming tasks can improve engagement [9,26]. In fact, prior research with blind novice programmers found that the programmers altered their text-to-speech device settings to produce interesting sounds [12]. Thus, the programming language should enable users to express creative control over the program output.

Easy sharing and remixing. Novice programmers may benefit from examining and building upon the code of others [26]. Furthermore, these programmers may be motivated by the ability to show off their work to friends [9]. Thus, programmers should be able to easily share their code and view code created by others.

Learn programming concepts in context. Introductory programming environments should emphasize computational thinking concepts such as sequences, loops, conditions, and events [7]. These concepts should be tied to the novice's goals so that mastering these concepts will help the learner achieve her goals.

DESIGN OF BONK

We developed Bonk, a programming toolkit that supports accessible programming of accessible media. The main components of Bonk are its audio game programming framework, web-based development environment, and HTML5-based game engine.

Audio Game Programming Framework

Bonk offers a scaffolded framework, based on JavaScript, that is optimized for creating interactive audio games. This framework abstracts away the complexities of text-to-speech audio, sound playback, and processing user input.

What Can You Make with Bonk?

We originally developed this tool with the intent of supporting interactive audio stories, similar to classic text adventure games, Choose Your Own Adventure books, and modern text-based game systems such as Inform[4] and Twine[5]. Developers can create an audio story, including speech and environmental sounds, and provide multiple paths that a player can follow. Bonk programs can react to specific keypresses, or specific strings typed by a player. Bonk also provides support for timed events, allowing developers to create simple action games.

As shown in our formative user study, this relatively simple set of tools allows for the creation of various types of games and content, including audio stories, interactive fiction, trivia games, and action games.

Program Structure

Bonk code is written in JavaScript, with an extensive set of convenience functions that reduce the overhead of creating programs. We chose to build Bonk on top of JavaScript so that the resulting programs could be run on any device with a modern web browser. Bonk uses the HTML5 Web Speech API for text-to-speech output, and the Web Audio API for sound effects. Figure 2 shows an example program.

Bonk programs are scaffolded to avoid the complexity of handling web page events. Each Bonk program has a function, called `run`. Code inside this function will run one time

[1] blindsidegame.com

[2] papasangre.com

[3] earplay.com

[4] inform7.com

[5] twinery.org

once the web page, speech engine, and sound files have loaded. This approach is similar to that used by Processing [25] and Arduino [16]. To reduce the ambiguity of identifying function names when using a screen reader, all built-in functions are written in lowercase, and multi-word functions are separated by underscores (e.g., `add_text_box`).

```
function run(game) {
    game.speak("You walk down a narrow stairwell.");
    var sound = new Sound("stairs");
    sound.play();
    game.speak("There is a big crowd here. On your left is the concession stand.
The line is very long. On the right is the way to the restroom. It is completely
packed.");
    var crowd = new Sound("crowd");
    crowd.play();
    game.speak("You probably don't have time for either of these, so you better
get to your seat! Another stairway ahead of you goes to the seating area.");
    game.speak("Press w to walk up the stairs, and s to go back.");
}

function on_key_press(key) {
    if (key == 'w') {
        game.go_to_room("Either_one_seats");
    } else if (key == 's') {
        game.go_to_room("Either_one_first_choice v.2");
    }
}
```

Figure 2. Example program created by a blind programmer. This example features text-to-speech, sound effects, key input, and a branching story structure.

Text-to-Speech

The key component of Bonk's interactive audio games is its text-to-speech component. This component is designed to simplify control of the text-to-speech engine, and to provide the developer with extensive control over speech output.

Creating an audio-based user interface can be challenging, especially for novice programmers. On many platforms, including the web, speech and audio output must be controlled using multithreaded programming. Incorrectly programming the speech threads could either cause the program to freeze up, or would cause all audio to play at once.

Bonk abstracts away the need to manage audio playback threads by queueing all audio commands, and managing their playback in a background thread. As an example of this challenge, the program in Figure 2 alternates between reading out text-to-speech messages and playing sound effects. Using the standard HTML5, this code would either freeze the browser until all audio had played (if run synchronously), or would play all sounds at once (if run asynchronously). To solve this problem, Bonk's `speak` and `play` functions add each action to a queue. At runtime, Bonk manages the background thread to play the audio output with proper timing, allowing the developer to create audio output as easily as text output.

Bonk's text-to-speech engine is designed to provide developers with control over voice parameters, including the specific voice, pitch, playback speed, and volume. In a prior programming workshop for blind students, the students entertained themselves by setting their text-to-speech settings to extreme values, such as a very fast and high-pitched voice, or a very slow and deep voice [12]. Bonk provides simple functions for changing text-to-speech parameters. Bonk also provides the ability to create multiple character profiles, each with a distinct voice, enabling developers to easily create audio stories with multiple characters.

Sound Effects

The other major component of Bonk's audio game is support for sound effects. As with text-to-speech, Bonk automatically manages the playback of multiple sounds in sequence. Furthermore, Bonk provides an integrated sound effect search feature, enabling developers to add sound effects to their program by simply typing the name of the desired effect. Bonk's sound effect library automatically finds an appropriate sound effect and plays it back.

When the user enters the name of a sound effect, Bonk automatically searches the Freesound[6] open source sound library for sounds matching the search terms that are close to 5 seconds in length; by default, Bonk plays the best matched sound. The purpose of this feature is to enable novice developers to sketch out an audio game as easily as they might sketch out a character for a video game. During our workshop, students added a variety of sound effects to their games, including *bite, boing, bonk, cheer, club party, computer, crowd, dance music, dance party, elevator, fanfare, jazz, laughter, loop, monster, music, rain, running, siri, stairs, storm, warzone,* and *water.*

Handling User Input

As mentioned previously, handling web page events can be extremely challenging for novice developers. Once again, Bonk provides simplified access to user input handling. Bonk supports two types of input events: key presses and text input. As in Processing and Arduino, Bonk developers can add event handling to their code simply by adding a specially named function to their code. To respond to single keypresses, the developer adds the function `on_key_press` to their code. To handle string input, such as text command, the developer initializes a textbox by adding a call to `add_text_box` in their main function, and adding the `on_text_box` to their code. This structure allows programmers to easily add input handling despite the web's complicated event model. Bonk also offers `start_timer` and `on_time_out` functions for timed input.

Branching Structure

Managing a game's state can quickly become complicated. Bonk supports more complex games by allowing developers to link different code snippets together. For example, to create a maze game in which the player can move north, south, west, or east, the developer can create a separate "room" for each direction, and link them together using the `go_to_room` command. This feature allows the developer to separate a game into smaller, manageable chunks, and enables multiple coders to work on a single game in parallel.

[6] freesound.org

Collaborative Coding Environment

Bonk programs are written, shared, and run via a web application, written in Node.js and using a MySQL database. This design was chosen to support our goals of easy deployment and easy sharing/remixing. Bonk is similar to Scratch [26] in that, at any time, a user can view the code of a running program, create a copy of that code, and remix the code. All code is entered via an HTML form (Figure 3).

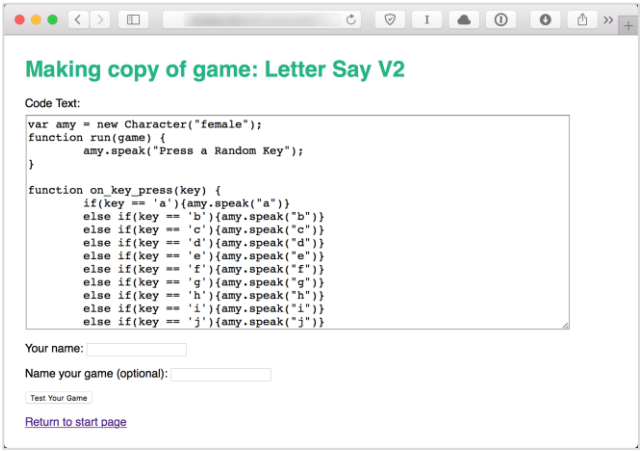

Figure 3. Bonk coding environment. Bonk programs are hosted on a central web repository where they can easily be shared and remixed.

When creating a new game, the developer enters the code into a code window, adds their name, and adds the name of their game. If the developer does not specify a name for their game, Bonk automatically assigns a name from a list of common words, as in URL shortening services like shoutkey.com. These automatically assigned names are easy to remember and easy to verbally share.

The Bonk web site contains a number of additional features: a list of recently created games, documentation for each function, a set of example games and source code, and a discussion forum.

Game Playback

Games created in Bonk are written in HTML and JavaScript, and can be played in any modern desktop or mobile browser. Games are shared with short, memorable URLs.

By default, Bonk presents the game content both as text-to-speech and on-screen text (Figure 4). Bonk games are self-voicing, using the HTML5 Web Speech API to support a variety of voice settings. It is possible to disable self-voicing mode: in this case, Bonk does not create its own speech output, but allows the user to render text using their own screen reader. While this mode reduces the expressive capabilities of Bonk's text-to-speech, some users may prefer to use their own customized voice settings.

EXPLORATORY CODING WORKSHOP

To understand the strengths and limitations of Bonk's approach, we tested the initial version of Bonk as part of a week-long coding workshop with 10 blind and visually impaired high school students.

Setting

The coding workshop took place as part of a larger STEM-learning camp hosted by a national organization serving blind and visually impaired high school students. Students chose one subject track, which met for half of each day for five days. Students participated in mini activities in the afternoons, and all of the students attending the camp met up on the last day to show off their work.

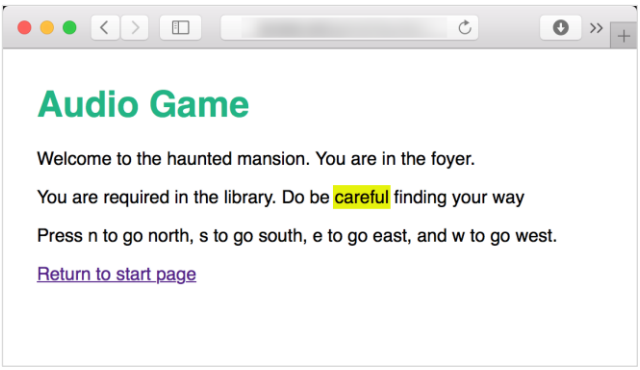

Figure 4. Bonk games are played in the web browser. Bonk's player provides various accessibility features, including high-contrast text and customizable text-to-speech output.

Participants

The computer science workshop track featured 10 students, ranging in age from 14 to 18. These students used a variety of assistive technologies to access their computing devices. A few students had some prior programming experience, but most of the students had never programmed before, and some had limited experience using computers and screen readers.

#	Gender	Assistive Tech.	Programming?
S1	F	screen reader, Braille	✓
S2	M	screen reader	✓
S3	M	screen reader	
S4	M	screen reader	
S5	M	magnifier	
S6	M	screen reader	✓
S7	M	screen reader	
S8	M	screen reader	✓
S9	M	magnifier	
S10	M	magnifier	
TA1	M	screen reader	
TA2	M	screen reader	✓
TA3	F	screen reader	✓

Table 1. Students and teaching assistants who participating in the coding workshop, their preferred assistive technology, and their prior programming experience.

The workshop was managed by three members of the research team. Additional support was provided by three teaching assistants, who were blind and low vision adults, although only two of the teaching assistants had any prior experience with computer programming.

Event Schedule

The workshop took place over 5 days, from approximately 8AM until noon. Each meeting included a variety of activities, including group discussions, solo and group programming. Because students came in with a wide range of experience with computers, the researchers provided written tutorials and exercises that students could follow independently, with other students, or with help from a TA or researcher. Each participant was provided with their own Windows laptop with the JAWS Screen Reader. Table 2 shows the daily schedule:

Day	Activities
1	Introduction to computer science concepts, acting out algorithms (as in CS Unplugged [3]), getting set up with assistive technology, basics of writing code.
2	Programming tutorials: expressions, variables, functions, objects, text-to-speech, sound effects.
3	Finish programming tutorials. Brainstorm game ideas. Form project groups.
4	Programming and testing games.
5	Final game testing. Group discussion and feedback. Project expo.

Table 2. Workshop schedule.

On Days 1, 2 and 4, the class discussed computing careers via Skype with blind computing professionals.

STUDENT EXPERIENCES WITH BONK

Here we report on the use of our accessible programming tools over the course of the week-long workshop.

Student Games

Students developed a variety of games over the course of the week. Although each student progressed at a different pace, each student uploaded at least one working program over the course of the week. During the third day, students formed into four project groups, and worked with their group for the rest of the week.

The four games are summarized in Table 3. Students designed the following games:

CS Mad Lib. Students developed a humorous game based on Mad Libs, in which the player is asked to name a set of words that fit certain criteria (e.g., adjective, place name), which were then inserted into a story.

WebNote. WebNote is a music-themed game, which originally began as two separate games, an interactive piano keyboard and a music trivia, *name-that-tune* game. The two students developed their games independently on the first day, and worked together on the second day to combine them via a unified game menu.

Either One. Students in this group developed a Choose Your Own Adventure-style game. This game was the most complex, with 13 different scenes. The group divided the work between multiple students, and assembled the complete game on the final day.

Labyrinth. One student chose to create a game on his own. This game was a maze-style game that required the player to navigate the maze and answer riddles.

Game Features

The students' games used several of Bonk's advanced features. All games included some text-to-speech output and interactivity. Three out of the four games used sound effects, custom voices, and branching paths.

In two cases, student groups decided to add features to their games that were not directly supported by the Bonk programming framework. The group that made CS Mad Lib wanted to add several text boxes to a single page, which was not originally supported by Bonk. By working with the teaching staff, the students were able to add this feature to their game. The group that made WebNote wished to add audio playback of specific music notes to their game. One student from that group, who had some prior programming experience, researched several ways to add musical note playback, and worked with the instructors to add this feature to the game.

Working in Groups

At the start of the workshop, students mostly completed tasks alone or with the help of one of the instructors. Over the course of the workshop, with some encouragement from the instructors, students began working together to write code, debug each other's code, and test games. Because there were more students than instructors, students sometimes sought

	Team Size	Lines of code	Num. scenes	Speech output	Keyboard navigation	Text input	Sound effects	Custom voices	Branching paths	Other features
CS Mad Lib	4	63	2	✓		✓	✓	✓		Form input
WebNote	2	78	3	✓	✓				✓	Music
Either One	3	102	13	✓	✓	✓	✓	✓	✓	
Labyrinth	1	92	9	✓	✓	✓	✓	✓	✓	

Table 3. Student teams created four games for the end-of-workshop project demonstration. These projects included several of Bonk's language features, including various forms of user input, customized user output, and branching story paths.

help from their peers rather than waiting for an instructor to become available.

On the third day, the instructors led the students through a brainstorming activity in which each student came up with several game ideas. Based on these ideas, students formed loose groups around broad topics: music games, interactive stories, and a Mad Libs-style game. After further discussion, the students formed their final project groups, and worked with their group for the remaining two days.

Collaboration took several forms within the project groups. In the CS Mad Lib group, students rotated roles over the course of their work. One student, who had been assigned a programming task by the *de facto* project leader, finished his coding early. He then took on the role of debugger, helping the other students in his group with their code. The group that created Either One assigned each member a specific role: lead programmer, lead tester, lead designer, and lead debugger (shared between the three students). Their game ends with a credits scene. The group that created WebNote mostly worked independently, occasionally sharing programming tips, until they combined their two games into a larger game on the final day of the workshop.

Creativity and Play

In addition to their coordinated group work, students occasionally participated in informal social interaction and play. The CS Mad Lib Group "performed" their game to the classroom, showing off the humor of their game. One student from the Either One group started a side project of his own, creating several audio stories featuring characters that he developed, each with different voices and personalities (Figure 5). One of the teaching assistants created a series of small puzzle games and shared them with the students; these games were intentionally designed to be frustrating, confusing, or impossible to "win."

```
var forever = new Character("female, 1,9");
forever.speak("hello guys, i am forever")
var isabellelu = new Character("female, 1.9");
isabellelu.speak("forever i was  just there, i saw you and we should go to a
performance of classical music")
var jenny = new Character("female, 1.4");
jenny.speak("yeah lets go and bring hyeong jeong she really wants to go i used to
play classical music its very fun")
var renali = new Character("female, 1.3");
renali.speak(" yeah i play in our schools orchestra with vy li and qyuhn")
var brent = new Character("male, 1.4")
brent.speak(" oh my god its rena stone come here its rena li she sent me a text
shes so awesome as you know i really like rena")
var stone = new Character("male, 1.5")
stone.speak(" yes i know you used to really like her do you still?")
```

Figure 5. Sample code from a student's side project, which uses multiple voices to create a sophisticated audio story.

Usability and Accessibility Issues

While all students were able to create some code, and were able to contribute to their final group projects, students occasionally encountered usability or accessibility issues using this initial version of Bonk. These problems are briefly summarized here.

Problems with assistive technology. Students in the workshop used several assistive technology devices, including screen readers, screen magnifiers, Braille displays, or a combination of several devices, and use of these devices sometimes caused errors. A software bug in the HTML code editor occasionally caused some error messages to be hidden, making it difficult to debug problems. This problem was fixed in the code, but caused some frustration early on. The screen reader software also sometimes captured the game player's keyboard input such that the game code did not detect the input. This issue could be overcome by entering a "pass-through" command to the screen reader; however, some students did not know about this pass-through feature, and instead turned their screen reader off to play the game, which sometimes caused problems if the student did not know how to reactivate the screen reader.

Another issue arose for students who brought their own assistive technologies, as some devices could not connect to the workshop network or to the laptops provided by the workshop organizers. One student brought her own refreshable Braille display, which she preferred to use when reading source code, but could not connect the device to the network. As a result, she chose to copy the relevant files from the laptop to the Braille device via a USB drive, and to copy them back when she was finished.

Barriers to collaboration. In some cases, students had difficulty working together due to problems integrating their assistive technologies. Several students had very little experience in using the JAWS screen reader, and required extensive help from the teaching assistants. However, in some cases, the teaching assistants were not familiar with the student's screen reader configuration, and therefore had difficulty supporting the student. In a few cases, students were able to help each other when the teaching assistant was unfamiliar with a specific issue.

In another case, a teaching assistant experienced difficulty in helping a student because she could not follow along with the student's screen reader output. Eventually the teaching assistant was able to acquire a headphone splitter, and was then able to follow along more easily.

Difficulties sharing and versioning documents. The Bonk programming environment was intentionally designed to avoid problems related to file management. Instead of managing a file system, each Bonk game was assigned a unique URL, and files cannot be deleted. Students sometimes had difficulty editing a document, and instead made copies of their documents, which resulted in duplicate documents with confusing names such as *game, game v1,* etc. Likewise, sometimes students had difficulty remembering and sharing the names of their games, especially multi-word names, where students might confuse names that are delimited by spaces, dashes, underscores, or camel case.

Syntax errors. As is to be expected, the novice programmers who participated in the workshop sometimes struggled with syntax errors in their code. These errors included common errors such as mismatching brackets and typing variable

names inconsistently. In some cases, these issues were clearly exacerbated by the student's vision impairment: some of the students who used screen magnification sometimes had difficulty reading the symbols, and had difficulty seeing structural problems with their code because they had zoomed in their screen, and thus could not see the top-level program structure. Bonk's editor did not provide syntax highlighting or line numbering, which may have caused some additional problems.

Limitations of the Framework

One issue that affected several students toward the end of the workshop was when students wished to perform some tasks that were not directly supported by the framework. In some cases, we were able to extend the framework during the workshop to increase functionality. For example, the WebNote group requested the ability to play a sound file based on a URL (rather than a search term); we were able to add this function to the sound effect library between workshop meetings.

However, some limitations of the framework were more difficult to overcome. Because Bonk provides extensive scaffolding in some areas, performing some tasks that seemed like they should be simple were instead surprisingly difficult. One example of this phenomenon was related to form input. Bonk provides the capability to generate text input boxes from JavaScript code. This feature enabled students to create form-based games without having to learn how to create HTML forms and connect them to JavaScript. This feature was designed to support only one text box per page, as we assumed that this would be used to support text commands. However, the CS Mad Lib group wished to add multiple forms to their page, which was not supported by the Bonk framework. We were able to add some limited functionality for supporting multiple text inputs, but because this was not part of our expected interaction model, the feature was not well integrated into the rest of the framework, nor was it described in the programming language documentation.

Other limitations of the current framework included the inability to set permanent game states, such as whether the player had picked up an object, which would require storing state between rooms (and HTML pages), and the ability to add background music, which would require modifying Bonk's audio queue feature. Supporting both a "low floor" and "high ceiling" [26] for accessible audio game programming presents an exciting challenge for future research.

Student Feedback

On the final day of the workshop, we facilitated a group discussion about the students' experience during the workshop, and their suggestions for improvements to both the structure of the workshop and to the Bonk framework.

We asked students what they liked most about the workshop. Several students mentioned that they were pleased with the quality of the game that they were able to create in such a short time. Other students mentioned that they enjoyed learning the basics of a "real" programming language like JavaScript. One student said, "It was our first game in JavaScript and it actually had some substance." One student praised the ability to create multiple voices and characters. Students said that they enjoyed working in groups, and appreciated the opportunity to design and develop their own games.

When asked what was difficult or frustrating about the workshop, students mentioned troubleshooting errors, connecting the system to a Braille display, and reading some of the on-screen text. Several of the students who came in with programming experience noted that they wished to go beyond the capabilities of the current framework.

Finally, we asked students what they would like to do if they had more time. One student mentioned that he would like to create more stories using the skills he had already developed. One student, who had some vision, said that he would like to develop graphical user interfaces, animations, or 3D graphics. Students also commented that they would be interested in creating web sites and forms, programming robots, working with databases, and learning more about computing fundamentals such as how data is represented in computer memory. Overall, the students expressed that they had enjoyed the workshop, and several students expressed enthusiasm about furthering their computer science education.

DISCUSSION

In this research, we developed a new programming toolkit that enables blind and visually impaired novice programmers to create accessible audio games. So, was our approach successful?

In the specific context of our week-long coding workshop, we would argue that the answer is yes. Students entered the workshop with a range of computer literacy and programming experience. In one week, all of the students were able to create their own audio games. While the experience was heavily scaffolded, most students still said that they felt they had gained "real" programming experience. Students fully engaged in the creative aspects of programming audio games, including integrating creative writing and humor into their games, and creating mini-games as a form of social play.

One challenge that was uncovered by this study is that we must ensure that Bonk provides both a "low floor" and a "high ceiling" [26]. A key component of Bonk's design is that it abstracts away some particular challenges of creating audio games, including managing speech and audio output, handling user input, and supporting sharing and remixing of games. In the future, we may explore how the various components of this system can be made modular, and can be combined and remixed with other programming tools. For example, a future descendant of Bonk could allow a student who is learning programming to create an audio story using Bonk's speech and sound libraries, embed this story into a

Quorum program, and share the result with friends via an online code portfolio.

FUTURE WORK

This work represents an initial step toward creating accessible programming tools for accessible media. Our initial deployment of Bonk has revealed numerous opportunities for improving the current programming toolkit, including providing features for more advanced programmers, supporting transfer from scaffolded introductory programming tools to more traditional programming tools, and extending Bonk's ability to create interactive stories and games.

Another topic that we are eager to explore is how tools like Bonk can be used to support collaborative work among people with vision impairments. Students in our programming workshop used Bonk as a tool to support creative expression and social interaction; this suggests that there is exciting potential in using shared production of accessible media to support social engagement.

Finally, we are excited by Bonk's potential as a platform for exploring and promoting "born-accessible" content [33]. Our approach with Bonk has not been to repair existing media computation platforms by bolting on accessibility, nor is it to develop a programming community that exists by and for blind programmers. Instead, this work is built around interactive games and stories that are not restricted to any particular representation. While this version of Bonk focuses primarily on supporting rich audio output, there is no reason that a future version could not render the same underlying source files as an animation, automatically generating visuals to match the story, or in any other format. We are excited to explore this platform not only as an accessible programming environment, but rather as a tool that is built from the ground up to create accessible media. For people who may currently be excluded from inaccessible media, Bonk could potentially empower them to bring more accessible media into the world, while for those people who are not used to encountering accessibility barriers in their own lives, this kind of tool may provide new insight on how we can all work toward a more accessible and equitable future.

CONCLUSION

We introduced Bonk, an accessible programming environment for creating accessible media. Bonk provides low barriers to entry for creating accessible audio games, and can enable aspiring programmers to develop their skills by creating, sharing, and remixing games that are built from the ground up to be accessible. A formative evaluation of Bonk with 10 blind and visually impaired high school students showed that this approach can enable students with a range of technical ability to create and share games, and that this accessible and collaborative programming environment can support shared creative work and play for people with a range of abilities.

SELECTION AND PARTICIPATION OF CHILDREN

This activity took place as part of a larger workshop conducted by a national organization serving blind and visually impaired children. The organizers of the event recruited the students; students then chose between a number of available workshops, including our computer science workshop. All children who participated in the workshop, and their parents, completed a consent form and photo release as part of their participation in the week-long event. This consent process was managed by the national organization; all of our activities took place within their broader framework and through collaboration with the event organizers.

ACKNOWLEDGMENTS

We thank our workshop participants and teaching assistants for their help in organizing the workshop. This work was supported by AccessComputing, and by the National Science Foundation under grants IIS-1619384 and IIS-1652907. Any opinions, findings, conclusions or recommendations expressed in this work are those of the authors and do not necessarily reflect those of the National Science Foundation.

REFERENCES

1. Khaled Albusays and Stephanie Ludi. 2016. Eliciting Programming Challenges Faced by Developers with Visual Impairments: Exploratory Study. In *Proceedings of the 9th International Workshop on Cooperative and Human Aspects of Software Engineering* (CHASE '16), 82–85. https://doi.org/10.1145/2897586.2897616
2. Catherine M. Baker, Lauren R. Milne, and Richard E. Ladner. 2015. StructJumper: A Tool to Help Blind Programmers Navigate and Understand the Structure of Code. In *Proceedings of the 33rd Annual ACM Conference on Human Factors in Computing Systems* (CHI '15), 3043–3052. https://doi.org/10.1145/2702123.2702589
3. Tim Bell, Jason Alexander, Isaac Freeman, and Mick Grimley. 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.
4. Jeffrey P. Bigham, Maxwell B. Aller, Jeremy T. Brudvik, Jessica O. Leung, Lindsay A. Yazzolino, and Richard E. Ladner. 2008. Inspiring Blind High School Students to Pursue Computer Science with Instant Messaging Chatbots. In *Proceedings of the 39th SIGCSE Technical Symposium on Computer Science Education* (SIGCSE '08), 449–453. https://doi.org/10.1145/1352135.1352287
5. Stacy M. Branham and Shaun K. Kane. 2015. Collaborative Accessibility: How Blind and Sighted Companions Co-Create Accessible Home Spaces. In *Proceedings of the 33rd Annual ACM Conference on Human Factors in Computing Systems* (CHI '15), 2373–2382. https://doi.org/10.1145/2702123.2702511

6. Stacy M. Branham and Shaun K. Kane. 2015. The Invisible Work of Accessibility: How Blind Employees Manage Accessibility in Mixed-Ability Workplaces. In *Proceedings of the 17th International ACM SIGACCESS Conference on Computers & Accessibility* (ASSETS '15), 163–171. https://doi.org/10.1145/2700648.2809864
7. Karen Brennan and Mitchel Resnick. 2012. New frameworks for studying and assessing the development of computational thinking. In (AERA 2012), 1–25. Retrieved September 18, 2017 from http://scratched.gse.harvard.edu/ct/files/AERA2012.pdf
8. Maureen Doyle. 2015. SIGCSE Symposium History. *SIGCSE Bull.* 47, 4: 7–8. https://doi.org/10.1145/2856332.2856338
9. Andrea Forte and Mark Guzdial. 2004. Computers for communication, not calculation: Media as a motivation and context for learning. In *System Sciences, 2004. Proceedings of the 37th Annual Hawaii International Conference on*, 10–pp. Retrieved September 19, 2017 from http://ieeexplore.ieee.org/abstract/document/1265259/
10. Kyle J. Harms, Jordana H. Kerr, Michelle Ichinco, Mark Santolucito, Alexis Chuck, Terian Koscik, Mary Chou, and Caitlin L. Kelleher. 2012. Designing a Community to Support Long-term Interest in Programming for Middle School Children. In *Proceedings of the 11th International Conference on Interaction Design and Children* (IDC '12), 304–307. https://doi.org/10.1145/2307096.2307152
11. Julie A. Jacko, V. Kathlene Leonard, and Ingrid U. Scott. 2009. Perceptual impairments: New advancements promoting technological access. *Human-Computer Interaction: Designing for diverse users and domains*: 93–110.
12. Shaun K. Kane and Jeffrey P. Bigham. 2014. Tracking @Stemxcomet: Teaching Programming to Blind Students via 3D Printing, Crisis Management, and Twitter. In *Proceedings of the 45th ACM Technical Symposium on Computer Science Education* (SIGCSE '14), 247–252. https://doi.org/10.1145/2538862.2538975
13. Caitlin Kelleher and Randy Pausch. 2007. Using storytelling to motivate programming. *Communications of the ACM* 50, 7: 58–64.
14. 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*, 1455–1464. Retrieved September 18, 2017 from http://dl.acm.org/citation.cfm?id=1240844
15. Varsha Koushik and Clayton Lewis. 2016. An Accessible Blocks Language: Work in Progress. In *Proceedings of the 18th International ACM SIGACCESS Conference on Computers and Accessibility* (ASSETS '16), 317–318. https://doi.org/10.1145/2982142.2982150
16. David Kushner. 2011. The making of arduino. *IEEE Spectrum* 26. Retrieved September 18, 2017 from http://utmechatronics.ir/wp-content/uploads/The-Making-of-Arduino-IEEE-Spectrum.pdf
17. Richard E. Ladner. 2015. Design for User Empowerment. *interactions* 22, 2: 24–29. https://doi.org/10.1145/2723869
18. Richard E. Ladner and Andreas Stefik. 2017. AccessCSforall: Making Computer Science Accessible to K-12 Students in the United States. *SIGACCESS Access. Comput.*, 118: 3–8. https://doi.org/10.1145/3124144.3124145
19. Scott Leutenegger and Jeffrey Edgington. 2007. A Games First Approach to Teaching Introductory Programming. In *Proceedings of the 38th SIGCSE Technical Symposium on Computer Science Education* (SIGCSE '07), 115–118. https://doi.org/10.1145/1227310.1227352
20. Stephanie Ludi and Tom Reichlmayr. 2011. The use of robotics to promote computing to pre-college students with visual impairments. *ACM Transactions on Computing Education (TOCE)* 11, 3: 20.
21. John Maloney, Mitchel Resnick, Natalie Rusk, Brian Silverman, and Evelyn Eastmond. 2010. The scratch programming language and environment. *ACM Transactions on Computing Education (TOCE)* 10, 4: 16.
22. Ezra Mechaber. 2014. President Obama Is the First President to Write a Line of Code. *whitehouse.gov*. Retrieved September 19, 2017 from https://obamawhitehouse.archives.gov/blog/2014/12/10/president-obama-first-president-write-line-code
23. Daniel Miller, Aaron Parecki, and Sarah A. Douglas. 2007. Finger Dance: A Sound Game for Blind People. In *Proceedings of the 9th International ACM SIGACCESS Conference on Computers and Accessibility* (Assets '07), 253–254. https://doi.org/10.1145/1296843.1296898
24. Lauren R. Milne. 2017. Blocks4All: making block programming languages accessible for blind children. *ACM SIGACCESS Accessibility and Computing*, 117: 26–29.
25. Casey Reas and Ben Fry. 2006. Processing: programming for the media arts. *AI & SOCIETY* 20, 4: 526–538.
26. Mitchel Resnick, John Maloney, Andrés Monroy-Hernández, Natalie Rusk, Evelyn Eastmond, Karen Brennan, Amon Millner, Eric Rosenbaum, Jay Silver, Brian Silverman, and Yasmin Kafai. 2009. Scratch: programming for all. *Communications of the ACM* 52, 11: 60–67. https://doi.org/10.1145/1592761.1592779
27. Timothy E. Roden, Ian Parberry, and David Ducrest. 2007. Toward mobile entertainment: A paradigm for narrative-based audio only games. *Science of Computer Programming* 67, 1: 76–90.

28. Jaime Sánchez. 2005. AudioBattleShip: blind learners cognition through sound. *International Journal on Disability and Human Development* 4, 4: 303–310.

29. Jaime Sánchez and Fernando Aguayo. 2005. Blind Learners Programming Through Audio. In *CHI '05 Extended Abstracts on Human Factors in Computing Systems* (CHI EA '05), 1769–1772. https://doi.org/10.1145/1056808.1057018

30. Andreas M. Stefik, Christopher Hundhausen, and Derrick Smith. 2011. On the Design of an Educational Infrastructure for the Blind and Visually Impaired in Computer Science. In *Proceedings of the 42Nd ACM Technical Symposium on Computer Science Education* (SIGCSE '11), 571–576. https://doi.org/10.1145/1953163.1953323

31. Andreas Stefik and Susanna Siebert. 2013. An Empirical Investigation into Programming Language Syntax. *Trans. Comput. Educ.* 13, 4: 19:1–19:40. https://doi.org/10.1145/2534973

32. Anja Thieme, Cecily Morrison, Nicolas Villar, Martin Grayson, and Siân Lindley. 2017. Enabling Collaboration in Learning Computer Programing Inclusive of Children with Vision Impairments. In *Proceedings of the 2017 Conference on Designing Interactive Systems* (DIS '17), 739–752. https://doi.org/10.1145/3064663.3064689

33. Brian Wentz, Paul T. Jaeger, and Jonathan Lazar. 2011. Retrofitting accessibility: The legal inequality of after-the-fact online access for persons with disabilities in the United States. *First Monday* 16, 11. Retrieved September 19, 2017 from http://journals.uic.edu/ojs/index.php/fm/article/view/3666

34. Fredrik Winberg and John Bowers. 2004. Assembling the Senses: Towards the Design of Cooperative Interfaces for Visually Impaired Users. In *Proceedings of the 2004 ACM Conference on Computer Supported Cooperative Work* (CSCW '04), 332–341. https://doi.org/10.1145/1031607.1031662

35. Jacob O. Wobbrock, Shaun K. Kane, Krzysztof Z. Gajos, Susumu Harada, and Jon Froehlich. 2011. Ability-Based Design: Concept, Principles and Examples. *ACM Trans. Access. Comput.* 3, 3: 9:1–9:27. https://doi.org/10.1145/1952383.1952384

36. Bei Yuan, Eelke Folmer, and Frederick C. Harris. 2011. Game accessibility: a survey. *Universal Access in the Information Society* 10, 1: 81–100.