The Sounds of Sorting Algorithms:

Sonification as a Pedagogical Tool

Joel C. Adams
adams@calvin.edu

Bryce D. Allen
bryceallen121@gmail.com

Bryan C. Fowler
bryan.fowler42@gmail.com

Mark C. Wissink
markcwissink@gmail.com

Dept. of Computer Science
Calvin University
Grand Rapids, MI, USA

Joshua J. Wright
wright.jjw@gmail.com

ABSTRACT

Much work already exists on algorithm visualization—the graphical representation of an algorithm's behavior—and its benefits for student learning. Visualization, however, offers limited benefit for students with visual impairments. This paper explores **algorithm sonification**—the representation of an algorithm's behavior using *sound*. To simplify the creation of sonifications for modern algorithms, this paper presents a new *Thread Safe Audio Library* (TSAL). To illustrate how to create sonifications, the authors have added TSAL calls to four common sorting algorithm implementations, so that as the program accesses a value being sorted, the program plays a tone whose pitch is scaled to that value's magnitude. In the resulting sonifications, one can (in real time) *hear* the behavioral differences of the different sorting algorithms as they run, and directly experience how fast (or slow) the algorithms sort the same sequence, compared to one another. This paper presents experimental evidence that the sonifications improve students' long-term recall of the four sorting algorithms' relative speeds. The paper also discusses other potential uses of sonification.

CCS CONCEPTS

• Human-centered computing ~Auditory feedback • Social and professional topics ~Computer science education

KEYWORDS

Accessibility, algorithm, audio, graphics, hearing, media, sonification, sorting, sound, visualization

ACM Reference format:

Joel C. Adams, Bryce D. Allen, Bryan C. Fowler, Mark C. Wissink, and Joshua J. Wright. 2022. The Sounds of Sorting Algorithms: Sonification as a Pedagogical Tool. In *Proceedings of the 53rd ACM Technical Symposium on Computer Science Education V. 1 (SIGCSE 2022), March 3–5, 2022, Providence, RI, USA.* ACM, New York, NY, USA, 7 pages. https://doi.org/10.1145/3478431.3499304

SIGCSE 2022, March 3–5, 2022, Providence, RI, USA

ACM ISBN 978-1-4503-9070-5/22/03…$15.00
https://doi.org/10.1145/3478431.3499304

1 Introduction

In 1950, Claude Shannon (the "father of information theory") visited Alan Turing in London. In their biography of Shannon [20], Jimmy Soni and Rob Goodman relate the following anecdote Shannon told after visiting Turing:

"*So I asked him what he was doing. And he said he was trying to find a way to get better feedback from a computer so he would know what was going on inside the computer. And he'd invented this wonderful command. See in those days, … the idea was to discover good commands.*

And I said, what is the command? And he said, the command is to put a pulse to the hooter, put a pulse to the hooter. Now let me translate that. A hooter … in England is a loudspeaker…

Now what good is this crazy command? Well … if you're in a loop, you can have this command in that loop and every time it goes around the loop it will put a pulse in and you will hear a frequency equal to how long it takes to go around that loop. And then you can put another one in some bigger loop and so on. And so you'll hear all of this coming on and you'll hear this 'boo boo boo boo boo boo boo' and his concept was that you'd soon learn to listen to that and know when it got hung up in a loop or something else or what it was doing all the time, which he'd never been able to tell before."

The remarkable thing about this story is that years before the development of high-level languages, compilers, debuggers, graphics, or any of the other modern programming conventions, Turing was creating a machine instruction that he could insert into a program to *hear* it executing in real-time. The result would be a new sonic language by which he could listen to a program's behavior. Fluency in this new language would let Turing hear if a program was running correctly or incorrectly.

This paper builds on Turing's idea by exploring *algorithm sonification*—incorporating sound into a program to hear the behavior of its underlying algorithm—as a tool for CS education. Section 2 discusses related work. Section 3 describes a library the authors have created to support sonification. Section 4 illustrates the use of this library by presenting sonifications of select sorting algorithms. Section 5 presents an assessment exercise and its results; Section 6 presents the authors' conclusions and thoughts for future work.

2 Related Work

Algorithms operate at a level of abstraction that can make them difficult for students to understand. Recognizing this, computer science educators have invested much effort in creating mechanisms to improve student understanding. For example, authors routinely use *figures* (e.g., flowcharts, finite state machine diagrams, UML diagrams) to visually depict computing abstractions within the static medium of a paper book.

In contrast to static paper, the dynamic medium of computer graphics makes it possible to create algorithm *visualizations*, which are visual models of an algorithm's behavior. A few samples of the work that has been done in this area include [1] [8][9][17][19]. Since much of a typical person's brain is devoted to visual processing, it has been logical for CS educators to use visuals to model program behaviors for most students.

However, visualizations offer limited benefits for students with visual impairments [16][18]. To better serve these students, we might explore ways to model computing abstractions using senses other than seeing. For these students, the sense of hearing is a reasonable place to begin exploring the non-visual representation of algorithmic behavior.

In the early 1990s, Digiano and Baecker explored creating programs that play sounds, which they called *auralization* [10]. Their subsequent work produced a new version of Logo called *LogoMedia* [11]. The work in this paper differs from their work by exploring the alternative area of auditory display called **sonification**, which Dictionary.com defines as "*the production of sound*" but Kramer, et al. define as "*the transformation of data relations into perceived relations of an acoustic signal for the purposes of facilitating communication or interpretation*" [15]. This work also differs from the work of Digiano and Baecker in: (i) using a compiled language (C++) instead of an interpreted language (Logo), (ii) using an object-oriented approach, and (iii) supporting both single-threaded and multithreaded computing.

Bingmann [3,4] has developed excellent sorting algorithm "audibilizations" [3]and visualizations. The library presented here is more general, since it is not limited to sorting algorithms.

Human-computer interface researchers have explored the use of sound in making program interfaces more accessible. For example, some researchers have explored the use of *earcons*—the auditory equivalent of icons—that provide sonic feedback when the user moves the mouse over menu choices, desktop icons, and other GUI items [5][12][13]. The work in this paper differs from such work by focusing on the use of sound to provide feedback on *algorithmic behavior*, as opposed to user behavior.

CS educators have also used sound in CS1 by having their students write programs that process multimedia sound files. For example, many novice students find that processing the notes in a sound-file is a motivating way to learn about loops [14]. Processing sound files is clearly different from Turing's idea of adding sound to a program to hear its algorithmic behavior. However, we find it encouraging that students find sound to be a motivating factor in learning about programming. Perhaps students will also find program sonifications to be more interesting and engaging than programs that are silent?

3 A Sonification Library

The lead author first had the idea of using sound to represent algorithmic behavior after answering a student's question. Whenever this student ran a multithreaded program, she noticed that her laptop's fan would start running, so she asked, "What is this program doing to cause my laptop to make all that noise?" Answering her question involved explaining that this program was using more of her computer's cores than a sequential program, generating more heat than normal, causing her laptop to turn on its fan to exhaust that heat.

In this case, the sound being generated by the student's computer when she ran the program was an *unintended side effect* of the program's behavior, caused by the laptop's hardware engineering. But this incident led the first author to begin wondering: Could one *intentionally* add sound to a program in a way that creates a meaningful sonic representation of the program's behavior? After subsequently reading of Shannon's visit to Turing, the author decided to explore this idea.

The first step was to examine existing sound libraries to see if they could be used to safely add sound to multithreaded programs. Some libraries offered an application programmer's interface (API) that was too low-level (e.g., designed for sound engineers to interact directly with the sound hardware). Others offered an API that was too high-level (e.g., not useful for adding sound to programs). Very few guaranteed thread-safety. After much fruitless searching, the authors decided to create a new library, which we describe next.

3.1 Design Goals

The design goals for this library were:

- *Thread-safety*: If a library is thread safe, then multiple threads can use the library simultaneously without producing any race conditions or deadlocks. Multicore processors are ubiquitous, and multithreading is the most common technique for fully utilizing such processors, so it is imperative that a modern library be thread safe.
- *Object-oriented*: Object oriented programming (OOP) supports the creation of highly maintainable and reusable code. Since C++ supports OOP, is commonly used for parallel / high performance computing, and allows one to directly interact with a computer's hardware, the authors chose it as the implementation language.
- *Easy to use*: The library's API should provide intuitive abstractions that simplify the task of creating sonifications.
- *Portable*: The library should be useable on Linux, MacOS, and Windows.

The authors decided to name this library the *Thread Safe Audio Library* (TSAL), which is descriptive, if not very creative.

To achieve the thread-safety, OOP, and portability goals, the authors built TSAL as a set of thread-safe C++ classes on top of *PortAudio*, a free, open-source, cross-platform, low-level audio API for the C programming language [6]. Figure 1 presents a *partial* UML class diagram showing the relationships of the primary abstractions TSAL provides:

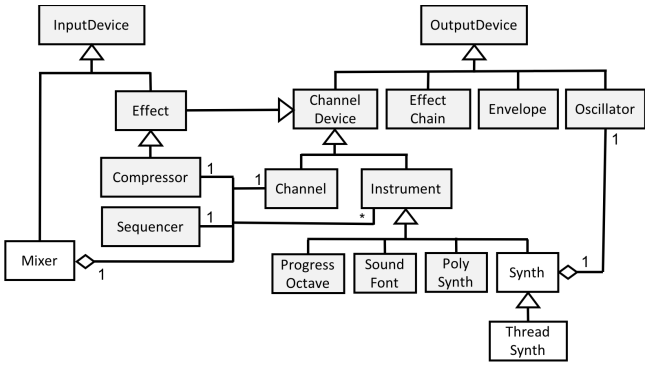


Figure 1: TSAL Class Structure Diagram

Figure 1 omits several TSAL classes and relationships that are tangential to this paper (e.g., those for handling MIDI files). The library is non-trivial, but its API allows one to create a sonification using just two classes—shown in white in Figure 1—the **`Mixer`** and **`Synth`** (or **`ThreadSynth`**) classes.

3.2 Library Use

TSAL can be used to turn an existing program into a sonification by following four simple steps: Within the program…

1. Define a **`Mixer`** object: a software representation of a multichannel mixer like those used by DJs to mix sounds.
2. Synthesizers are used to play sounds, so define a **`Synth`** object (for single-threaded programs) or a **`ThreadSynth`** object (for multithreaded programs) to generate sounds.
3. Add the **`Synth`** object to the **`Mixer`**. Different synthesizers have different sonic characteristics and capabilities, so a **`Mixer`** may mix the outputs of multiple synthesizers. For example, each thread in a multithreaded program might create and add its own synthesizer to the **`Mixer`**, which blends the sounds those synthesizers generate.
4. Use the **`Synth`** (or **`ThreadSynth`**) object to play sounds by invoking its **`play()`** method. For a sonification, the sounds played should express the program's algorithmic behavior.

If our problem involves processing the values from a data set, then as we process a given value *v*, one way we might accomplish step 4 is to use **`play()`** to play a note whose pitch is scaled to the magnitude of *v*—higher notes for larger values, lower notes for smaller values.

The next section uses this approach to create sonifications of four well-known sorting algorithms. Each sonification performs steps 1-3 as follows:

```
Mixer mixer = new Mixer();
Synth synth = new Synth(); // or ThreadSynth()
mixer.add(synth);
```

4 Sorting Sonifications

Sorting sonifications are interesting because they let one hear the behavioral differences of different algorithms that solve the same problem.

4.1 Insertion Sort

For small sequences, Insertion Sort remains one of the fastest sorting algorithms, making it an important algorithm for students to understand. Figure 2 presents an implementation of Insertion Sort, augmented with TSAL calls (shown in blue) to turn it into a sonification:

```
void insertSort(vector<int>& data, Synth& synth) {
  const int SIZE = data.size();

  for (int i = 1; i < SIZE; ++i) {
    int insertVal = data[i];
    MidiNote note1 = scaleToNote(data[i],
                                 0, MAX_VALUE,
                                 C3, C7);
    synth.play(note1, Timing::MICROSECOND, 50);

    int j = i;
    while (j > 0 && data[j-1] > insertVal) {
      MidiNote note2 = scaleToNote(data[j-1],
                                   0, MAX_VALUE,
                                   C3, C7);
      synth.play(note2, Timing::MICROSECOND, 50);

       data[j] = data[j-1];
       --j;
    }
    data[j] = insertVal;
  }
}
```

Figure 2: Insertion Sort Sonification

In Figure 2, the parameter **`data`** is a vector of N integer values, randomly generated from the range 0 to **`MAX_VALUE`**. In keeping with MIDI notation, **`C3`** is a TSAL-defined constant for the C-note one octave below middle C (C4), and **`C7`** is a constant for the C-note 3 octaves above middle C. The function call:

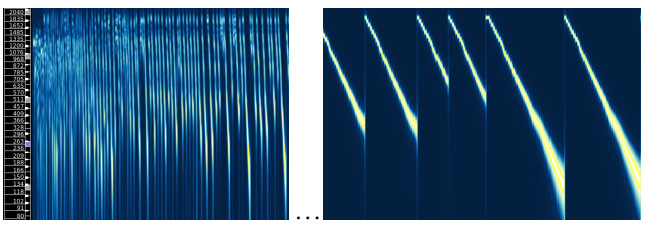
```
scaleToNote(data[i], 0, MAX_VALUE, C3, C7)
```

returns a MIDI note from the range **`C3..C7`**, whose pitch is scaled to the position of **`data[i]`** in the range **`0..MAX_VALUE`**. Once that MIDI note has been generated, the method call:

```
synth.play(note1, Timing::MICROSECOND, 50);
```

plays that note for 50 microseconds. Within the inner loop, a similar approach is used to play a note for **`data[j-1]`**.

The static medium of a conference paper does not permit the reader to hear a sonification; a **spectrogram** is a sound-chart that graphs sonic pitch (Hz, y-axis) against time (x-axis). Figure 3 shows segments of the spectrogram (made with Sonic Visualizer [7]) generated by the sonification in Figure 2; Figure 3a shows its first few seconds; 3b shows its final few seconds:



(a) First Two Seconds **(b) Last Two Seconds**
Figure 3. Insertion Sort Spectrogram Segments

Insertion Sort works by keeping a sorted subsequence (initially length 1) at the beginning of the array. For each other item, it repeatedly: (i) finds the next value to be inserted (`data[i]`), and (ii) moves it backward until it reaches its correct position within that sorted subsequence, which takes linear time in the length of that subsequence. When the algorithm begins, that subsequence is very short, so step (ii) is on average very fast, as indicated in Figure 3a. But as the algorithm runs, the length of this subsequence increases, making step (ii) on average take longer, as can be seen in Figure 3b. The sonification lets a person *hear* this change in behavior. Put differently a sonification lets a person hear why Insertion Sort is a good algorithm for short sequences, but not a good algorithm for long sequences.

Using a Linux workstation equipped with a 3.0-GHz Intel i5 CPU, the sonification in Figure 2 took 117 seconds to sort 2500 integers, because the sonification plays each note for 50 microseconds. By contrast, if the same function is run without the sonification code, it takes roughly 0.2 seconds to complete. The sonification's lengthy time may seem like a disadvantage: Why would one want the sort to proceed more slowly?

When *teaching* students about sorting, the sonification's longer runtime offers a significant *pedagogical advantage*: Instead of `insertSort()` sorting the sequence in a fraction of a second—as does every other sorting algorithm—the sonification's prolonged runtime has a student *experience* the algorithm's speed relative to other sorting sonifications. Put differently, a slow sonification creates a visceral learning experience by leveraging a typical student's impatience, potentially improving the student's learning about algorithmic (in)efficiency.

4.3 Quick Sort

For comparison, Figure 4 presents an implementation of the Quick Sort algorithm, augmented to produce a sonification:

```
void quickSort(vector<int>& data, int lo, int hi,
               Synth& synth) {
  if (lo < hi) {
    int pivotVal = data[lo];
    int pivotIndex = lo;
    MidiNote note1 = scaleToNote(pivotVal,
                          0, MAX_VALUE, C3, C7);
    synth.play(note1, Timing::MICROSECOND, 50);

    for (int i = lo+1; i < hi; ++i) {
      MidiNote note2 = scaleToNote(data[i],
                          0, MAX_VALUE, C3, C7);
      synth.play(note2, Timing::MICROSECOND, 50);

      if (data[i] < pivotValue) {
        ++pivotIndex;
        swap(data[i], data[pivotIndex]);
      }
    }
    swap(data[lo], data[pivotIndex]);

    quickSort(data, lo, pivotIndex-1, synth);
    quickSort(data, pivotIndex+1, high, synth);
  }
}
```

Figure 4: Quick Sort Sonification

Quick Sort chooses a "pivot" value (here the first value); reorganizes the array into the subarray of values less than the pivot, followed by the pivot, followed by the subarray of values greater than the pivot; recursively sorts the first subarray; and then recursively sorts the second subarray. In Figure 4, the sonification plays a note whose frequency is scaled to the pivot value near the start of each recursive call. Then each iteration of the for loop compares the pivot value to `data[i]`, so the sonification plays a note scaled to `data[i]`. Figure 5 shows a complete spectrogram for the Quick Sort sonification:

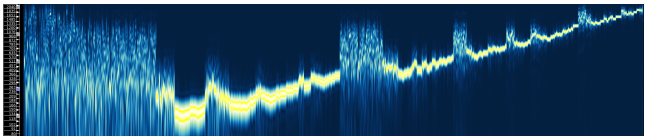


Figure 5. Quick Sort Spectrogram (Complete)

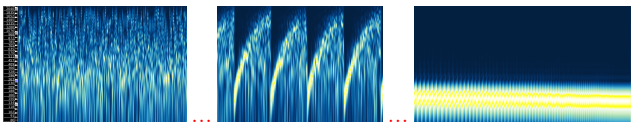
If one compares Figures 5 and 3, Quick Sort's behavioral differences can be clearly *seen*, but when a student runs the sonifications, the differences—descending tones vs. ascending tones—can be clearly *heard*. The chaotic sections of Figure 5 are the different recursive calls, as each call compares its pivot value against the other values in its `lo..hi` subsequence.

When run without the sonification code, the `quickSort()` function in Figure 4 sorts the same 2500 values in about 0.008 seconds. This is clearly faster than `insertionSort()`'s 0.17 secs, but sub-second times like 0.008 and 0.17 are too similar to make an impression on many students (see below).

However, when run as a sonification, the function in Figure 4 sorts those 2500 values in about 2 seconds. To typical students, this short execution time is amazingly different from the 117 seconds required by Insertion Sort. In addition to hearing the algorithms' behavioral differences, sonification lets students directly experience just how fast Quick Sort is.

4.4 Other Sorting Algorithms

The authors also created sonifications for Bubble Sort and Merge Sort. Space limitations prevent us from presenting their code here, but they follow the same approach as Figures 2 and 4: when the algorithm accesses a given data-value, the sonification plays a note whose pitch is scaled to that value's magnitude. Figure 6 presents segments from Bubble Sort's spectrogram:



(a) Initial (b) Middle (c) Final Seconds

Figure 6. Bubble Sort Spectrogram Segments

Bubble Sort is inefficient, but its sonic behavior is interesting: Initially (Figure 6a), it sounds just like its name, as the larger values "bubble" to the end of the sequence. As the algorithm loops through the remaining values, those large values disappear from consideration, the smaller values "sink" toward the front of the sequence, and a "heartbeat" sound appears (6b). As the sequence of unsorted values gets shorter, this "heartbeat" gets faster and lower

in pitch (6c). Without the sonification code, the Bubble Sort program sorts the 2500 values in 0.49 seconds; with the sonification code, it takes an excruciating 324 seconds.

Figure 7 presents a complete spectrogram for Merge Sort:

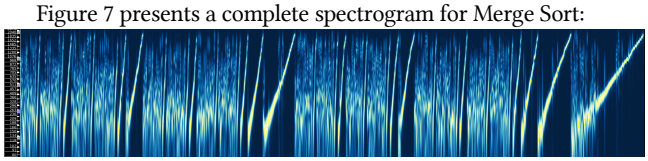


Figure 7. Merge Sort Spectrogram (Complete)

In Figure 7, the chaotic-looking sections are short subsequences being merged; the arcs that ascend from left-to-right are the longer subsequences being merged. As the lengths of the subsequences increases, the merges take longer and can be clearly heard. Without the sonification code, the Merge Sort program sorts the 2500 values in 0.013 seconds; with the sonification code, it sorts them in about 3 seconds. These four sorting algorithms thus have completely distinct sonifications or *sonic signatures*, as can be seen in their spectrograms.

5 Assessment and Discussion

To assess the effectiveness of sonification as a learning tool, the authors developed these research questions:

RQ1: Do sonifications improve students' long-term recall?
RQ2: Do visualizations improve students' long-term recall?
RQ3: Do sonifications improve student engagement?
RQ4: Do visualizations improve student engagement?

5.1 Experiment

To answer these questions, the authors designed and conducted an experiment: CS2 students (*Introductory Data Structures*, where sorting algorithms are *not* covered) were invited to participate in the experiment for extra credit. 24 students volunteered and were randomly assigned to four groups of 6 students each (*Control*, *Audio*, *Graphics*, and *Audio+Graphics*). One student from the *Control* group mistakenly attended the *Audio+Graphics* session, resulting in group sizes of 5, 6, 6, and 7, respectively.

Each session was held in a computer lab and consisted of four 15-minute segments, on Bubble Sort, Insertion Sort, Merge Sort, and Quick Sort, respectively. In each segment, students were:

a. Given a 30-second "elevator pitch" introduction to that sorting algorithm (1 minute).

b. Directed to read a detailed online tutorial about that algorithm from *www.tutorialspoint.com* (5 minutes).

c. Directed to run a program implementing that algorithm. A pseudocode version of the algorithm was displayed / projected on a screen at the front of the lab, and the students were invited to use it as a reference (4 minutes).

d. Directed to re-read the online tutorial for that algorithm to better understand the observed behavior (5 minutes).

In step c, each group was shown how to run a given program from the command-line, for example:

```
$ ./quick_sort
```

Each program sorted the same C++ vector of 2500 pseudorandom integers and then displayed the *sort-time*—the time it took the program to sort the sequence.

The key difference in the treatment of each group came in step c, in *how* they were instructed to run the programs:

- *Control*: Students used no command-line switches.
- *Audio*: Students used the **`-a`** switch was used, making each program run a TSAL sonification as it sorted (see Section 4).
- *Graphics*: Students used the **`-g`** switch, making each program run a TSGL [1] visualization as it sorted.
- *Audio+Graphics*: Students used the **`-ag`** switch, making each program run a sonification and visualization as it sorted.

The *Control* group thus saw each sorting program's sort-time; the *Audio* group heard each program sort the sequence, then saw its sort-time; the *Graphics* group saw each program sort the sequence and then saw its sort-time; the *Audio+Graphics* group heard and saw each program sort before seeing its sort-time.

Note that the sort-times for a given algorithm were not uniform for the four groups. In the *Control* group, all sort-times were fractions of a second. The sonifications in the *Audio* and *Audio+Graphics* groups produced sort-times of minutes or seconds, depending on the algorithm, as described in Section 4. The sort-times of the *Graphics* group's visualizations were similar to those of the *Audio+Graphics* group but slightly shorter.

After two weeks, each subject was emailed a link to a quiz. To assess RQ1 and RQ3, the quiz had 12 multiple-choice questions, organized as three 4-question blocks: 4 questions in which students had to match an algorithm's pseudocode to its name, 4 in which students had to identify the fastest or slowest of a subset of the algorithms, and 4 in which students had to match an algorithm to its Big-Oh time-complexity. To assess RQ2 and RQ4, the quiz also had a 13th question in which each student was asked to rate how engaging she found her session on a 1 (*Boring, uninteresting*) to 10 (*I loved it*!) scale.

5.2 Results and Discussion

The students' quiz scores fit a (roughly) normal distribution. Out of 12 possible points, the maximum score was 12; the minimum score was 2, the mean was 7.125 and the median was 7.5.

For RQ1 and RQ2, the null hypotheses were that sonifications and visualizations would have no significant effect on students' long-term recall of sorting details. To analyze the responses, the authors used T-tests (2-tailed, homoscedastic) to compare the performances of the students in each experimental group to the performances of the students in the *Control* group, using a *p-value* of 0.05 as the threshold for significance.

As can be seen in Figure 8, all experimental groups did better on the quiz than the *Control* group, and the *Audio* group did the best of all groups, but none of the differences were significant (*p=0.418* [*Audio*], *0.561* [*Graphics*], *0.786* [*Graphics+Audio*]):

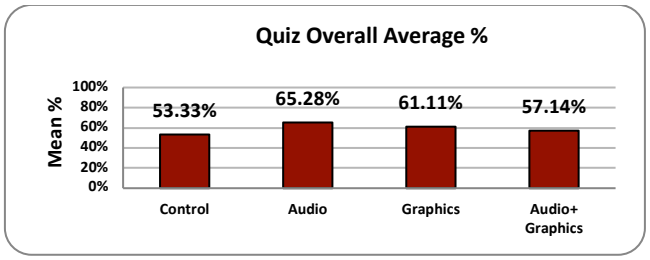


Figure 8: Comparing Groups Overall Quiz Performance

The students in the experimental groups also did better than the students in the *Control* group on most individual questions, but the differences did not meet the significance threshold. This is not surprising, given our small group-sizes (5, 6, 6, and 7).

We also analyzed the groups' performances on each of the three 4-question blocks (pseudocode, fastest/slowest, big-oh). For the second block, in which students were given subsets of the algorithms and asked to identify the fastest or slowest, the *Audio* group did significantly better than the *Control* group ($p=0.043$). The other two experimental groups also did better than the *Control* group but did not meet our significance threshold ($p=0.165$ [*Graphics*], $p=0.077$ [*Audio+Graphics*]). Figure 9 shows the four groups' performances on this 4-question block:

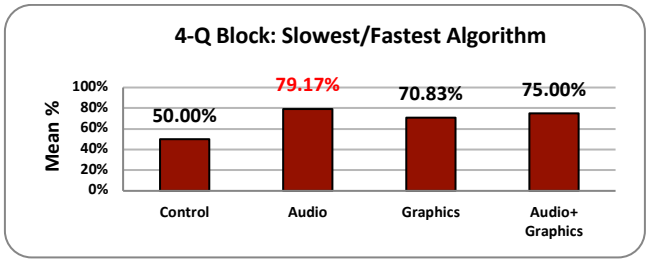


Figure 9: Slowest/Fastest Algorithm 4-Question Block

This result lets us reject the null hypothesis for RQ1, providing evidence that sonification can indeed improve a student's long-term memory. We believe that the greater differences in the sort-times that the students in the *Audio* and *Audio+Graphics* groups experienced helped those students better remember the four algorithms' speeds, relative to one another.

For RQ2, none of our results met the 0.05 significance threshold needed to reject the null hypothesis.

For RQ3 and RQ4, the null hypotheses were that sonifications and/or visualizations would have no effect on students' session-ratings in question 13. Figure 10 summarizes the groups' ratings:

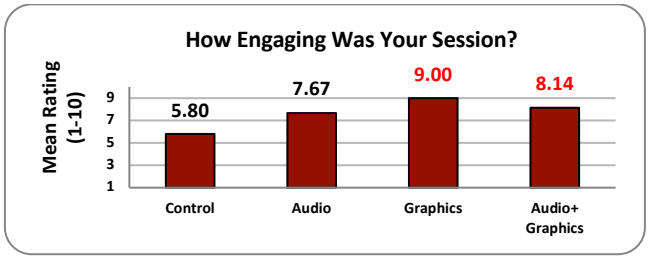


Figure 10: Engagement Ratings

While the *Audio* group rated their experience more highly than the *Control* group, the difference did not meet our significance threshold ($p=0.178$), so we were unable to reject the null hypothesis for RQ3.

However, the responses of both the *Graphics* and *Audio+Graphics* groups ratings met the significance threshold ($p=0.007$ and 0.037, respectively) when compared to the *Control* group, despite our small group sizes. This lets us reject the null hypothesis for RQ4, and it adds to the evidence that students find algorithm visualizations to be engaging learning tools.

6 Conclusions and Future Work

This paper has introduced the *Thread Safe Audio Library* (TSAL)—a new tool for adding sounds to working programs—and used it to generate *sonifications* of four common sorting algorithms. Using spectrograms, we have shown that each sorting algorithm has a distinct *sonic signature* that offers insights into the algorithms' behavioral differences.

We have presented experimental evidence that sonification can improve students' long-term recall of algorithmic details, plus new evidence that students find visualizations to be engaging ways to learn. These results were statistically significant, despite our small sample sizes and none of our students having visual impairments. We also presented other positive results (e.g., students finding sonifications engaging) that might prove significant with larger groups or among students with visual impairments. For anyone interested in repeating our experiments, our sonifications, materials, and data are freely available from the authors by request. TSAL may be freely downloaded from https://github.com/Calvin-CS/TSAL.

In adding sound to sorting algorithms, we have scarcely scratched the surface of the potential of sonification. Other possible uses of sound that we hope to explore include:

- Hearing the differences between single-threaded and multi-threaded versions of the same algorithm.
- Tracing the execution of a recursive function through its "winding" phase, anchor case, and "unwinding" phase.
- Tracing a program's execution via sonic checkpoints.
- Auditory alerts: generating distinct (and appropriate) sonic feedback when different exceptions are thrown.
- Illustrating the concept of a side-channel attack [2].

Sonification thus opens up many new possibilities, including the creation of a new sonic "language" by which a program can communicate its behavior to its users. By gaining fluency in such a language, CS students and faculty may gain new insights into a program's underlying algorithm.

Thanks to the rich set of tools that have been developed in the years since Turing first sought to "*put a pulse to the hooter*", we can now realize his idea and venture even further. We invite others to use TSAL to explore algorithm sonification and look forward to (literally) hearing their results.

ACKNOWLEDGMENTS

This work was supported by NSF-DUE#1822486.

REFERENCES

[1] J. Adams, P. Crain, C. Dilley, C. Hazlett, E. Koning, S. Nelesen, J. Unger. TSGL: A Tool for Visualizing Multithreaded Behavior, *Journal of Parallel and Distributed Computing*, Volume 118, Issue P1, Aug 2018, pp. 233-246.

[2] S. Bhunia and M. Tehranipoor, Chapter 8—Side-Channel Attacks, in *Hardware Security: A Hands-On Learning Approach*, Morgan-Kaufmann, 2019, pp. 193-218.

[3] T. Bingmann, *The Sound of Sorting - "Audibilization" and Visualization of Sorting Algorithms*. Online, accessed 2021-12-01: https://panthema.net/2013/sound-of-sorting/.

[4] T. Bingmann, *15 Sorting Algorithms in 6 Minutes*. Online, accessed 2021-12-01: https://www.youtube.com/watch?v=kPRA0W1kECg,

[5] M. Blattner, D. Sumikawa and R. Greenberg. Earcons and icons: their structure and common design principles. *Human Computer Interaction*, 4:1 (March 1989), pp. 11–44.

[6] P. Burk, et al. *PortAudio: Portable, Cross-Platform Audio I/O*. Online, accessed 2021-12-01: http://www.portaudio.com.

[7] C. Cannam, C. Landone, and M. Sandler. Sonic Visualiser: An Open Source Application for Viewing, Analysing, and Annotating Music Audio Files, *Proceedings of the ACM Multimedia 2010 International Conference*, October, 2010, pp. 1467-1468.

[8] S. Carr, J. Mayo, and C.K. Shene. ThreadMentor: a Pedagogical Tool for Multithreaded Programming, *Journal on Educational Resources in Computing* (JERIC), 3(1), March 2003, Article 1.

[9] A. Danner, T. Newhall, K. Webb. ParaVis: A Library for Visualizing and Debugging Parallel Applications, *Proc. of 9th NSF/TCPP Workshop on Parallel and Distributed Computing Education* (EduPar-19), in conjunction with IEEE IPDPS'19, Rio de Janeiro, Brazil, May 2019.

[10] C. DiGiano and R. Baecker. Program Aurelization: Sound Enhancements to the Programming Environment. *Proc. of the Conference on Graphics Interface '92*. Morgan Kaufmann Publishers, 1992, pp. 44-52.

[11] C. DiGiano and R. Baecker. LogoMedia: A Sound-Enhanced Programming Environment for Monitoring Program Behavior. *Proc. of the 1993 Conference on Human Factors in Computing Systems* (CHI'93), pp. 301-302.

[12] S. Garzonis, S. Jones, T. Jay, E. O'Neill. Auditory icon and earcon mobile service notifications: intuitiveness, learnability, memorability and preference. *Proc. SIGCHI Conf. on Human Factors in Computing Systems*, April 2009. pp. 1513-1522.

[13] W. Gaver. The SonicFinder: an interface that uses auditory icons. Human Computer Interaction. 4:1 (March 1989), pp. 67–94.

[14] M. Guzdial, D. Ranum, B. Miller, B. Simon, B. Ericson, S. Rebelsky, J. Davis, K. Deepak, D. Blank. Variations on a theme: role of media in motivating computing education. *Proc. of the 41st ACM SIGCSE Symposium on CS Education*, March 2010, pp. 66-67.

[15] G. Kramer, B. Walker. T. Bonebright, P. Cook, J. Flowers, N. Miner and J. Neuhoff. "Sonification Report: Status of the Field and Research Agenda" (2010). *Faculty Publications, Department of Psychology, University of Nebraska-Lincoln*. Online, accessed 2021-12-01: https://digitalcommons.unl.edu/psychfacpub/444.

[16] C. Morrison, N. Villar, A. Thieme, Z. Ashktorab, E. Taysom, O. Salandin, D. Cletheroe, G. Saul, A. Blackwell, D. Edge, M. Grayson & H. Zhang. Torino: A Tangible Programming Language Inclusive of Children with Visual Disabilities, *Human–Computer Interaction*, 35:3 (2020). pp. 191-239.

[17] T. Naps, S. Rogers, G. Rößling and R. Ross, Animation and visualization in the curriculum: opportunities, challenges, and successes. *Proc. of the 37th ACM SIGCSE Symposium on Computer Science Education* (SIGCSE'06), March 2006. pp. 328-329.

[18] A Thieme, C Morrison, N Villar, M Grayson, and S Lindley (June 2017). Enabling collaboration in learning computer programing inclusive of children with vision impairments. *Proc. of the 2017 Conference on Designing Interactive Systems*, June, 2017. pp. 739-752.

[19] C. Shaffer, M. Cooper, A. Alon, M. Akbar, M. Stewart, S. Ponce, S. Edwards. Algorithm Visualization: The State of the Field. *ACM Transactions on Computing Education*, 10(3), Aug. 2010. Article 9.

[20] J. Soni and R. Goodman, *A Mind At Play*, Simon & Schuster, 2017, pp. 108-109.