SpeechGames: Voice Programming in Computer Science Education

April 10, 2018

Abstract

We introduce the SpeechGames platform for exploring topics in computer science education.

1   Introduction

There is tremendous interest in computer science education. SpeechGames is a platform that may be used to explore topics in computer science education for children who may be suffering from disabilities by means of a voice interface to the Blockly coding environment. The may help children who are unable to use a mouse and keyboard interface.

In this technical report, we shall first describe our design principles for SpeechGames, following guidelines proposed by cognitive load theory worked out in educational psychology and computer science education. Then we will go on to describe components of our architecture, from the programming environment to the speech recognition interface. After we've described the building blocks, we will detail algorithmic work to improve usability. And then we will proceed to discuss related and future work.

2   HCI Principles

Cognitive load is the total amount of mental effort being used in the working memory [1]. Reducing cognitive load has been linked to improved educational outcomes [2], and hence was a guiding principle in designing our educational platform.

Cognitive load theory distingusihes between three different types of load:

- **Intrinsic load.**
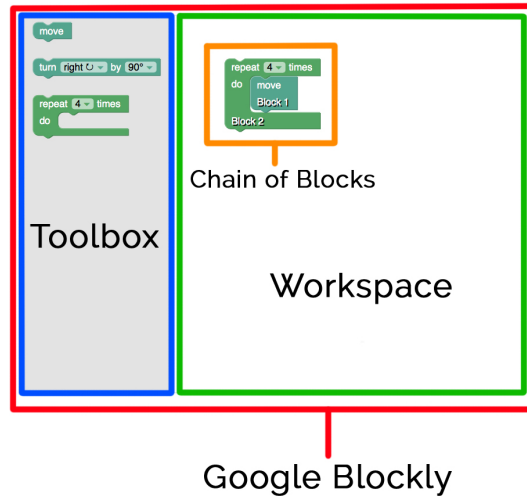
- **Extraneous load.**

- **Germane load.**

Figure 1: Google Blockly workspace diagram

[3] further identifies a series of effects to serve as guidelines for creating learning materials:

- **Redundancy effect**

Speech recognition presents particular challenges for designing effective educational materials. [4] argues that speech input/output consumes cognitive resources in short-term and working memory, these effects being mitigated in applications with a limited vocabulary and constrained semantics.

3   Blockly

Google Blockly is a JavaScript library that creates a visual block programming environment where core computer science logic can be taught, such as conditionals and looping. Blockly is run in the web browser, which gives the opportunity for mobile use. We chose Google Blockly over other block based languages due to its easy to work with and has a customizable API.

Let us introduce some terminology here that would be used extensively in the paper moving forward. Blockly has the concept of a **workspace**, as shown in **Figure 1**. This is the space where all the blocks that need to be executed in the program are located. A **chain of blocks** refers to a series of vertically connected blocks in the workspace. Vertically connected blocks are executed sequentially. In Google Blockly, it is also possible to several unconnected chains of blocks in the workspace. These chains are executed in order of the coordinates of their topmost block. A menu to the left of the workspace holds the available block types for a program. We will refer to this menu of available block types

as the **toolbox**. In order to move blocks from the toolbox to the workspace, users must click and drag using a mouse in a traditional Blockly setting.

The toolbox is where all the allowed blocks for a particular program can be found. Google Blockly has a large library of blocks, which can be grouped as:

- **Logic**: if statements and equality testing

- **Loops**: while loops, for loops, for-each loops and break statements

- **Math**: incrementing variables, summing numbers, randomizing numbers and other basic math and geometry operations.

- **Text**: displaying text, appending text, searching through text, substring, printing and prompting

- **Lists**: creating a list with variables, length, testing if empty and searching through a list

- **Color**: changing the color and random coloring

- **Variables**: creating a variable, initializing it and modifying its value

- **Functions**: a chain of blocks with a name and a possible return value

4 Speech

To reduce cognitive load, we focused on creating a concise and fixed grammar, meaning there is only one way to say a command, which is a term we use to describe a manipulation performed by speech. While speech recognition can be used for a conversation style of programming, we wanted to mimic the standard idea that computer languages have rigid grammars.

To create a rigid grammar, we focused on making the grammar as simple as possible. We employed a guiding principle of **syntactic parsimony**. That, there ought only be one command in the grammar per action in the workspace. For this reason, the grammar includes no optional or choice elements. No synonyms are allowed for the main verbs (get, delete, change, connect, separate) of the sentences of the grammar. We also disallowed variability on prepositional phrase location. For example, we only allow "Change left in block 1 to right" and not "Change left to right in block 1." We were also guided by a principle of **semantic parsimony**. The commands "Move block 2 under block 1" and "Move block 1 under block 2" for example might be equivalent operations, but due to semantic parsimony, we only allow moving blocks *under* one another.

The user has the following commands:

- Add - "get a _____ block"

- Connect - "connect block ___ under block ___"
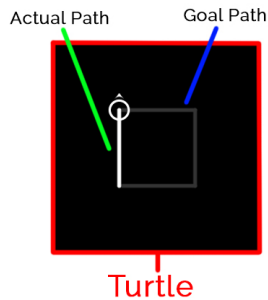
- Connect inside - "connect block ___ inside block ___"

Figure 2: Turtle workspace diagram

- Separate - "seperate block ___"

- Change field - "change ___ in block ___ to ___"

- Delete - "delete block ___"

- Run program - "run the program"

- Next level - "go to the next level"

- Stay on level - "stay on this level"

4.1 How do the user interface commands described earlier map to the commands in the grammar?

5 Turtle

In order to implement the voice-programming interface using Blockly, we decided to use Turtle. Turtle is a Blockly game, that contains a turtle that can be moved around using a small subset of commands (blocks). The objective of the game is to make the turtle trace a particular path shown to the user, using only the provided blocks in the toolbox. Turtle is one of many Blockly games that we could have chosen in combination with Google Blockly and our speech recognition, but chose it for its simplistic design.

- **move**: moves the turtle forward

- **turn**: turns the turtle, can decide between left or right turning and angle of turning 1, 45, 72, 90, 120, 140 degrees

- **repeat**: repeatedly executes the blocks inside it, can decide to repeat either 2, 3, 4, 5 or 360 times

- **pen**: can decide between up/down, up stops the turtle from drawing a line, down makes the turtle's movements draw lines again

4

Turtle evenly divides computer science concepts into levels in order to reduce the cognitive load of the learner. Each individual level reinforces previously learned topics, while introducing new ones simultaneously. An example of a couple of consecutive levels would be as follows:

1. User is exposed to how to move the turtle forward and also turn the turtle to make a square. In order to make the square, the user is only give move and turn blocks, meaning they need four move blocks and three turn blocks to complete the task.

2. User is given a repeat block in the toolbox. Now the user is tasked at completing the same task as before, but with only using one repeat block, one move block and one turn block.

A user has now, in two levels, been first introduced to the mechanics of moving and turning the turtle in the Google Blockly environment and then naturally transitioned into loops. Since the task is simply "create a square", it is not too difficult to utilize the new blocks provided, given the knowledge of previous levels.

6 Speech recognition

We perform speech recognition using the Webkit Speech API in Google Chrome. The API provides us with the user's speech, as understood by the speech recognition engine, in text form. We chose this option over CMU's Sphinx because we wanted to get better recognition and the Google Speech API has been trained on a lot more data compared to CMU's Sphinx engine which we could train on a limited and non-representative set of data that we would generate. Google's Speech API is also easier to use and available in other web browsers (since this is primarily a web application).

7 Suggestions

We provide the user a list of suggestions to teach them the grammar. We do this because the grammar is a rigid, fixed grammar. Furthermore, we provide these suggestions because we don't give any training in the grammar to the user. Instead, we hope that they can pick it up by following the suggestions. The suggestions mechanism provides generic suggestions, not specific to the block IDs or the workspace. We do not suggest how the user should solve the program, as the user might mistakenly believe. Instead, we simply teach them how to use the grammar. However, the appearance of particular suggestions is triggered by certain states of the workspace. For example, when an empty repeat block is on the canvas, we suggest that the user "Connect block 2 inside block 1" with these exact block IDs, regardless of what the block ID of the repeat block is. Finally, we provide an example in Table 1 of the suggestions box with the corresponding workspace before and after adding the first block. We modify the

Table 1: Example of suggestions before and after adding the first block to the workspace.

when	workspace	suggestions
before		"Get a move block"
after	move Block 1	"Get a move block" "Delete block 1" "Run the program"

suggestions list when we believe it could be useful for the user. For example, when there is a repeat block on the canvas, we suggest that the user connects a block inside of the repeat block, to make use of the repeat block. However, our current suggestions system is quite limited.

8 Corrections

As previously described, we use the Google speech API built into Google Chrome to capture user commands. However, the API has a hard time understanding commands from our grammar. We hypothesize that the API expects ordinary English, and as a result, phrases like "get a turn block" or "change 4 in block 3 to 5" are often recognized incorrectly by webkitspeechcrecognition. We developed Algorithm 1 to correct incorrectly recognized phrases ("recognitions") to what we hope are the intended commands ("utterances").

The correction algorithm first takes in a recognition r, a workspace W, and a max modification factor (described later) λ. First, the recognition r is converted into a phoneme sequence ρ using a modified version of CMU's pronouncing dictionary [5]. Then we generate a set C of all possible commands that a user can specify given the workspace. For each command c, we convert it into a phoneme sequence γ and compute its edit distance e to ρ. We iterate over C to find the minimum edit distance e^* (breaking ties arbitrarily) and corresponding minimum edit distance command c^*.

Furthermore, if μ is too far from the original recognition sequence ρ, we reject the correction μ and notify the user that we didn't understand their command. We added this feature to avoid using a "correction" from a completely unrelated sentence picked-up or recognized by the speech API. Our notion of "too far" is satisfied when

$$e^* > \lambda * length(\rho)$$

6

As such, there is no fixed maximum number of edits, as this could give different performance for different string lengths.

Algorithm 1 Correction Algorithm

1: **procedure** CORRECT(r, W, λ)
2:     $\rho \leftarrow stringToPhoneme(r)$
3:     $C \leftarrow generatePossibleCommands(W)$
4:     $e^* \leftarrow \infty$
5:     $c^* \leftarrow r$
6:     $\gamma^* \leftarrow \rho$
7:     **for** c in C **do**
8:         $\gamma \leftarrow stringToPhoneme(c)$
9:         $e \leftarrow findMinEditDist(\rho, \gamma)$
10:        **if** $e < e^*$ **then**
11:           $e^* \leftarrow e$
12:           $c^* \leftarrow c$
13:           $\gamma^* \leftarrow \gamma$
14:        **end if**
15:     **end for**
16:     **if** $e^* \leq \lambda * length(\rho)$ **then**
17:        **return** c^*
18:     **else**
19:        **return** r
20:     **end if**
21: **end procedure**

8.1 Correction Algorithm Parameters

The correction algorithm requires the following parameters to run:-

- **Recognition**: The recognition of the utterance by the speech API.

- **Workspace**: The state of the workspace that contains information about the numbers and types of blocks currently present on the workspace.

- **Maximum Modification Factor**: The maximum percentage of modification (i.e. correction) allowed to the original recognition. The methodology to get calculate this number, is described in section 8.2.

8.2 Calculation of Maximum Modification Factor

8.2.1 Data Collection

The data was gathered manually by speaking into the speech engine and recording the utterance, recognition and correction (if it exists) in text form. The correct and incorrect utterances, along with their recognitions and corrections were stored as CSV files.

8.2.2 Statistical Procedure

The data containing the accuracy corresponding to each threshold value from 0.00 to 1.00 (with a step size of 0.01) was collected by running the analysis on both positive and negative data examples.

For the correct examples, the accuracy was obtained by counting the total number of correct corrections. In other words, we need to count the total number of times the recognition was corrected to the original utterance. For the incorrect examples, the accuracy was obtained by counting the total number of times corrections were not made - since corrections are always valid, mapping an incorrect utterance to a valid command is not correct.

Lastly, the maximum modification factor was calculated by using the threshold value that corresponded to the maximum average accuracy of the positive and negative examples. It is important to note that the average was calculated by weighing the categories equally, and not the examples.

8.2.3 Limitations

1. We had approximately 100 positive examples and 200 negative examples, and this is not the expected distribution of real-world data. In practice, we would expect a lot more positive examples than negative.

2. The negative examples were mostly random utterances that might be picked up by the microphone in a loud environment. In other words, these utterances didn't even come close to being recognized as a valid speech command. While this data might be useful when a lot of vocal activity is going on around the user, this does not serve any purpose in adapting the speech recognition to a user that is using the speech interface in a quiet environment (which is the expected behavior).

3. Sample size of approximately 300 examples is low.

8.3 Give an example of a common case that the corrections algorithm, as currently implemented, fails on

9 Layout

As the user writes a program, certain locations on the workspace fill with blocks. This presents a challenge: what should happen when blocks overlap with one another? For our system to be practically useful, it must avoid introducing such visual impairments, which may prevent the user from issuing commands (e.g. if they cannot see a block ID) or which may introduce unnecessary cognitive load.

Blockly by default places new blocks at the top left of the workspace, even if the new block will overlap an old block. Similarly, when Blockly connects one block to another, the resulting chain might overlap another chain.

We devise simple layout algorithms for each case. To make this formal, we view the workspace as a 2D plane W where the origin is the top left corner, the x

axis ranges from 0 to WIDTH(W), and the y axis ranges from 0 to HEIGHT(W). Let B denote the set of blocks on the workspace and let m denote a small, constant margin. In our implementation, we choose $m = 20$ (pixels) because it is small enough to preserve space on the workspace, and large enough to prevent Blockly from automatically connecting nearby blocks.

Adding Blocks We place new blocks by finding the vertically lowest free position on the workspace and placing the new block there, exactky m pixels right of the toolbox. This is summarized by Algorithm 2.

Moving Blocks Suppose the user has just connected one block to another, and the resulting chain now overlaps n chains. We repair the layout by moving the n conflicting blocks individually via Algorithm 3.

Algorithm 2 Place New Block

---

procedure PLACENEWBLOCK(B, m)
    $\ell \leftarrow m$ ▷ stores lowest y on the workspace
    for $b \in B$ **do**
        if $y_b + \text{HEIGHT}(b) \geq \ell$ **then**
            $\ell \leftarrow y_b + \text{HEIGHT}(b) + m$
        end if
    end for
    return (m, ℓ)
end procedure

Algorithm 3 Relayout Existing Block

---

procedure RELAYOUT(b, B, W, m)
    $x \leftarrow m$
    while $x < \text{WIDTH}(W)$ **do**
        $y \leftarrow m$
        while $y < \text{HEIGHT}(W)$ **do**
            if CANMOVEWITHOUTCONFLICT($b, (x, y)$) **then**
                MOVE($b, (x, y)$)
            end if
            $y \leftarrow y + m$
        end while
        $x \leftarrow x + m$
    end while
    MOVE($b, (m, m)$)
end procedure

---

10 Related Work

Speech interfaces for programming tasks have a long history, dating back to the beginning of speech recognition research [6].

The bulk of work aims to improve the productivity of professional developers using IDEs through speech. The first body of work here tends to specify its method to a particular programming language. For example, [7] creates a grammar for a language, Spoken Java, which is syntactically similar to Java, with a goal of disambiguating natural language utterances. [8] uses a set of rules to trasncribe dictation results into fragments of code. [9] constructs a speech plug-in focused more on the IDE's control than its language. And [10] designs a programming language from the ground up for the purpose of being developed through a speech interface. [11] proposes a voice-activated syntax-directed editor.

A second group uses voice and other modalities, such as gesture and eye-tracking, to control aspects of the IDE which need not be language-specific. [12] maps human voice and gestures are mapped to IDE commends using hardware sensors for Microsoft Kinect. [13] adds eye-tracking control to IDEs.

The question of education for motorically challenged children has been investigated in [14] where a speech-based IDE was built atop Scratch.

Our design decision were motivated by a consideration of cognitive load theory. Both cognitive load theory and the value of block-based programming environments have been extensively discussed in the literature. Cognitive load theory was first introduced in [1] as an explanation for how problem-solving activity may be ineffective in facilitating learning. [15] and [16] explore the implications of cognitive load theory for computer science education. [17] conducts a comprehensive review on the value of blocks-based programming environments for reducing cognitive load, while [18] conducts experiments in the value of block-based programming environments for teaching computer science concepts. And [19] evaluates the benefit of teaching computer science in game-like environments.

11   Future Work

References

[1] John Sweller. Cognitive load during problem solving: Effects on learning. *Cognitive science*, 12(2):257–285, 1988.

[2] Slava Kalyuga, Alexander Renkl, and Fred Paas. Facilitating flexible problem solving: A cognitive load perspective. *Educational Psychology Review*, 22(2):175–186, 2010.

[3] Cyril Rebetez. Control and collaboration in multimedia learning: Is there a split-interaction, 2006.

[4] Ben Shneiderman. The limits of speech recognition. *Commun. ACM*, 43(9):63–65, September 2000.

[5] Alex Rudnicky. Cmudict, 2015.

[6] Richard A Bolt. *Put-that-there: Voice and gesture at the graphics interface*, volume 14. ACM, 1980.

[7] Andrew Begel. Programming by voice: A domain-specific application of speech recognition. In *In AVIOS Speech Technology Symposium SpeechTek West*, 2005.

[8] Alain Désilets, David C Fox, and Stuart Norton. Voicecode: An innovative speech interface for programming-by-voice. In *CHI'06 Extended Abstracts on Human Factors in Computing Systems*, pages 239–242. ACM, 2006.

[9] Shairaj Shaik, Raymond Corvin, Rajesh Sudarsan, Faizan Javed, Qasim Ijaz, Suman Roychoudhury, Jeff Gray, and Barrett Bryant. Speechclipse: an eclipse speech plug-in. In *Proceedings of the 2003 OOPSLA workshop on eclipse technology eXchange*, pages 84–88. ACM, 2003.

[10] Benjamin M Gordon. Developing a language for spoken programming. In *AAAI*, 2011.

[11] Thomas J Hubbell, David D Langan, and Thomas F Hain. A voice-activated syntax-directed editor for manually disabled programmers. In *Proceedings of the 8th International ACM SIGACCESS Conference on Computers and Accessibility*, pages 205–212. ACM, 2006.

[12] Denis Delimarschi, George Swartzendruber, and Huzefa Kagdi. Enabling integrated development environments with natural user interface interactions. In *Proceedings of the 22nd International Conference on Program Comprehension*, pages 126–129. ACM, 2014.

[13] Hartmut Glücker, Felix Raab, Florian Echtler, and Christian Wolff. Eyede: gaze-enhanced software development environments. In *CHI'14 Extended Abstracts on Human Factors in Computing Systems*, pages 1555–1560. ACM, 2014.

[14] Amber Wagner, Ramaraju Rudraraju, Srinivasa Datla, Avishek Banerjee, Mandar Sudame, and Jeff Gray. Programming by voice: A hands-free approach for motorically challenged children. In *CHI'12 Extended Abstracts on Human Factors in Computing Systems*, pages 2087–2092. ACM, 2012.

[15] Dale Shaffer, Wendy Doube, and Juhani Tuovinen. Applying cognitive load theory to computer science education. In *Workshop of the Psychology of Programming Interest Group*, volume 333, page 346, 2003.

[16] Keith Nolan, Aidan Mooney, and Susan Bergin. Examining the role of cognitive load when learning to program. 2015.

[17] David Bau, Jeff Gray, Caitlin Kelleher, Josh Sheldon, and Franklyn Turbak. Learnable programming: blocks and beyond. *Communications of the ACM*, 60(6):72–80, 2017.

[18] Orni Meerbaum-Salant, Michal Armoni, and Mordechai Ben-Ari. Learning computer science concepts with scratch. *Computer Science Education*, 23(3):239–264, 2013.

[19] Ben Gibson and Tim Bell. Evaluation of games for teaching computer science. In *Proceedings of the 8th Workshop in Primary and Secondary Computing Education*, pages 51–60. ACM, 2013.