Voice Programming in Computer Science Education

February 13, 2018

1 Introduction

Our goal is to create a voice enabled platform that uses machine learning to allow anyone to learn the fundamentals of computer science. Currently, we seek to reduce the amount of cognitive load required to be introduced to programming concepts, which would give more students opportunities in computer science. Voice programming looks to benefit both new and experienced programmers. Google Blockly is a simple language that allows us to create a robust programming grammer that can be used of people of all ages. In addition to discovering solutions to voice recognized programming, our project will help teach people who are unable to use a traditional mouse and keyboard.

1.1 What are the results or outcome of the project?

2 HCI Principles

2.1 What HCI principles were kept in mind in the design of our system?

2.2 How did those HCI principles influence our design decisions?

3 Blockly

3.1 What is the Blockly environment?

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

3.2 How are blocks arranged horizontally? What language do we use to describe this?

3.3 How are blocks arranged vertically? What language do we use to describe this?

In Blockly, there is the ability to allow several chains of blocks to be run while in the workspace. The order of execution causes the topmost chain of blocks (topmost being indicated by the top block in the chain's y location). Normally, users interact with Blockly through looking at the workspace and adding blocks to the workspace by using the toolbox. In order to move blocks from the toolbox to the workspace, users must click and drag using a mouse.

Available blocks

- **move**: moves the tutle forward
- **turn**: turns the turtle, can decide between left or right turning and angle of turning 1,45,72,90,120,140 degrees
- **repeat**: repeats the blocks found inside, 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

Users must refer to blocks as the block number that is shown on the block
Users can move blocks in 2 ways:

- **Connect block <ID> under block <ID>**: allows user to move blocks and chains of block under any block or chain of blocks
- **Connect block <ID> inside block <ID>**: allows user to move block and chains of blocks inside of a repeat block

delete block <ID>: deletes a block, unless the block is the head of a chain of blocks, then this command deletes the entire chain of blocks.

3.3.1 How does separating blocks work?

4 Turtle

4.1 What is the turtle game?

4.2 What set of blocks are available to complete the game?

4.3 What do these blocks do?

4.4 Why is the game divided into levels?

4.5 How do the levels serve to teach different programming concepts?

5 Speech

5.1 Why did we decide to model speech using a

5.2 What choices did we make as we designed that grammar?

5.3 What commands are available in that grammar?

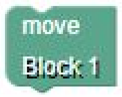
5.4 How do the user interface commands described earlier map to the commands in the grammar?

6 Speech recognition

6.1 How do we perform the speech recognition? What do we use?

6.2 What tradeoffs are there in our decision by comparison to alternatives? What was gained and what lost?

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

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

7 Suggestions

7.1 Why do we give suggestions to the user? What HCI principle is behind this?

We modify the 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. It provides generic suggestions, not specific to the block IDs. 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.

7.2 What is basis on which a speech suggestion is made? What circumstances trigger it?

7.3 What are the limitations of the current method and what future work might improve upon it?

7.4 Show an example of the suggestions box and workspace before and after an operation

8 Corrections

Utterance: A single command given by the programmer
Recognition: What the speech reognition software recogizes an utterance as
Correction: The proposed utterance given a recognition, calculated using cor-

rection algorithm

As previously described, we use the webkitspeechrecognition speech API 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 a novel algorithm to correct incorrectly recognized phrases (”recognitions”) to what we hope are the intended commands (”utterances”). The algorithm finds the command with minimal phoneme edit distance from the set of possible commands for a given workspace.

Algorithm 1 Correction Algorithm

```
1:  procedure CORRECT(recognition, workspace)
2:      $recognitionPhoneme \leftarrow stringToPhoneme(recognition)$
3:      $allPossibleCommands \leftarrow generateAllPossibleCommands(workspace)$
4:      $minimumEditDistance \leftarrow \infty$
5:      $minimumEditDistanceCommand = recognition$
6:      for $command$ in $allPossibleCommands$ do
7:          $commandPhoneme \leftarrow stringToPhoneme(command)$
8:          $editDistance \leftarrow findMinimumEditDistance(recognitionPhoneme, commandPhoneme)$
9:          if $editDistance < minimumEditDistance$ then
10:             $minimumEditDistanceCommand = command$
11:             $minimumEditDistance = editDistance$
12:         end if
13:     end for
14:     return $minimumEditDistanceCommand$
15: end procedure
```

Furthermore, if the minimum edit-distance command is too far from the original recognition, we reject the correction and simply notify the user that we didn’t understand their command. Our notion of ”too far” is more concretely defined by a maximum modification factor which defines the the number of phoneme edits that can be made as a percentage of the number of phonemes in the recognition. As such, there is no fixed maximum number of edits, as this could give different performance for different string lengths. We determined the threshold in the following way.

8.1 If there are any parameters to be fit in this correction algorithm, how were they fit?

8.1.1 What statistical procedure was use?

8.1.2 What data was gathered?

8.1.3 What assumptions were made that could be considered weaknesses of the model?

8.2 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.

Adding Blocks (1) Find the vertically lowest block on the workspace. Let y be the vertical coordinate of its lower left corner. (2) Place the new block such that its top left corner is at $(20, y)$, that is, below the lowest block on the workspace, 20 pixels right of the flyout.

Moving Blocks Suppose the user has just connected one block to another, and the resulting chain now overlaps n blocks. We repair the layout by moving the n conflicting blocks via the procedure below. For brevity, let W and H be the width and height of the entire workspace, respectively, let b be the block to be moved, and let w_b and h_b be the width and height of b and its children. Additionally define m to be a small margin.

1. Construct the set of candidate new locations:

$$C = \{(x, y) = (rm, sm) : r, s \in \mathbb{Z}, x \leq W, y \leq H\}$$

2. For every $(x, y) \in C$ check if there is no block within the box bounded by the four points:

$$(x, y), (x + w_b, y), (x, y + h_b), (x + w_b, y + h_b)$$

   Let $(\hat{x}, \hat{y})$ be the first found satisfying that condition. If none is found, set $\hat{x} = \hat{y} = m$.

3. Move b so that its top left corner is located at $(\hat{x}, \hat{y})$.

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.

10 Related Work

10.1 What is the related work in speech interfaces for programming?

10.2 What is the related work in speech interfaces for computer science education?

11 Future Work