

# Voice Programming in Computer Science Education

February 20, 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 grammar 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 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.

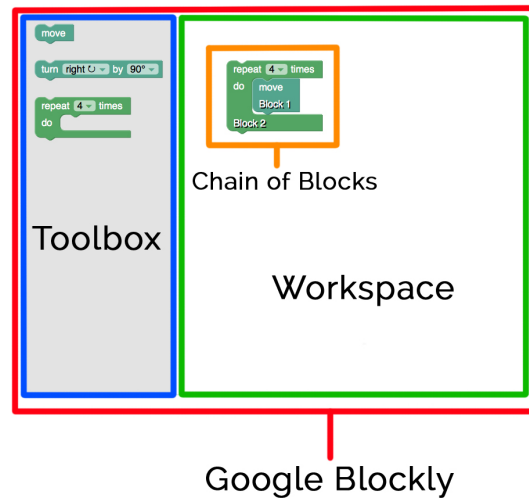


Figure 1: Google Blockly workspace diagram

**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?**

Blockly has a “workspace” (See figure 1), which is where all of the blocks in the program are located. A “chain of blocks” (See figure 1) is used to refer to a series of connected blocks in the workspace. In Google 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. The topmost chain is computed based off the highest vertical location of the first block in a chain. Normally, users interact with Blockly through looking at the workspace and adding blocks to the workspace by using a menu that holds the allowed blocks for a program, which we will refer to as the “toolbox” (See figure 1). In order to move blocks from the toolbox to the workspace, users must click and drag using a mouse.

#### Available blocks

- **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:** 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 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 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.

- 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**

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 `webkitspeechrecognition`. We developed Algorithm 1 to correct incorrectly recognized phrases (“recognitions”) to what we hope are the intended commands (“utterances”). First, we convert the recognition into a phoneme sequence using CMU’s pronouncing dictionary. Then we generate all of the possible commands that a user can specify given the workspace. We convert all of these commands into phoneme sequences and find the command whose corresponding phoneme sequence has the minimum edit distance to the recognition’s phoneme sequence (breaking ties arbitrarily) and return that command.

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. 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 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.

---

**Algorithm 1** Correction Algorithm

---

```
1: procedure CORRECT(recognition, workspace)
2:    $rphon \leftarrow stringToPhoneme(recognition)$ 
3:    $commands \leftarrow generatePossibleCommands(workspace)$ 
4:    $minEditDist \leftarrow \infty$ 
5:    $minEditCommand \leftarrow recognition$ 
6:   for  $command$  in  $commands$  do
7:      $cphon \leftarrow stringToPhoneme(command)$ 
8:      $editDist \leftarrow findMinEditDist(rphon, cphon)$ 
9:     if  $editDist < minEditDist$  then
10:       $minEditCommand \leftarrow command$ 
11:       $minEditDist \leftarrow editDist$ 
12:     end if
13:   end for
14:   if  $minEditDist \leq maxModification * length(rphon)$  then
15:     return  $minEditCommand$ 
16:   else
17:     return  $recognition$ 
18:   end if
19:   return  $minEditCommand$ 
20: end procedure
```

---

**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  $W$  where the origin is the top left corner, the  $x$  axis ranges from 0 to  $\text{WIDTH}(W)$ , and the  $y$  axis ranges from 0 to  $\text{HEIGHT}(W)$ . 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, exactly  $m$  pixels right of the flyout. This is summarized by Algorithm 2.

**Moving Blocks** Suppose the user has just connected one block to another, and the resulting chain now overlaps  $n$  blocks. 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  $y_b + h_b \geq \ell$  then
       $\ell \leftarrow y_b + h_b + m$ 
    end if
  end for
  return  $(m, \ell)$ 
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  $\text{EPHEMERALMOVE}(b, (x, y))$  and no conflicts then
         $\text{MOVE}(b, (x, y))$ 
      end if
       $y \leftarrow y + m$ 
    end while
     $x \leftarrow x + m$ 
  end while
   $\text{MOVE}(b, (m, m))$ 
end procedure

```

---

## **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**