Comparing Textual and Block Interfaces in a Novice Programming Environment

Thomas W. Price
North Carolina State University
890 Oval Drive
Raleigh, NC
twprice@ncsu.edu

Tiffany Barnes
North Carolina State University
890 Oval Drive
Raleigh, NC
tmbarnes@ncsu.edu

ABSTRACT

Visual, block-based programming environments present an alternative way of teaching programming to novices and have proven successful in classrooms and informal learning settings. However, few studies have been able to attribute this success to specific features of the environment. In this study, we isolate the most fundamental feature of these environments, the block interface, and compare it directly to its textual counterpart. We present analysis from a study of two groups of novice programmers, one assigned to each interface, as they completed a simple programming activity. We found that while the interface did not seem to affect users' attitudes or perceived difficulty, students using the block interface spent less time off task and completed more of the activity's goals in less time.

Categories and Subject Descriptors

K.3.2 [**Computers and Education**]: Computer and Information Science Education; D.1.7 [**Programming Techniques**]: Visual Programming

Keywords

Block programming, drag-and-drop, programming interface

1. INTRODUCTION

Programming is a challenging subject to learn, and educators have investigated many strategies for making it more accessible to students [17]. Much of this effort has been directed towards creating better programming environments for novices, resulting in many new systems [13, 18]. A common feature in many modern novice programming environments is the use of drag-and-drop blocks of code, which fit together to form a program, minimizing the possibility of syntax errors and the need to memorize procedure names. While this feature can be traced at least as far back as the LogoBlocks environment [3], it has become prevalent in many more recent environments [1, 8, 10, 16, 29], which have been evaluated in classrooms [24, 25, 26], summer camps [21, 31] and after-school programs [22].

In this paper, we will use the term Block-Based Programming Environment (BBPE) to refer to those environments that allow users to construct and execute computer programs by composing atomic blocks of code together to produce program structure. These code blocks may additionally have slots, which can be filled by other blocks; for example, a function call block may have slots for each of its parameters. These blocks may represent high-level structures, such as methods or loops, or low-level operators such as multiplication or equality comparison. An example is shown in Figure 1. There exist a variety of programming environments which use the block metaphor, but here we limit our use of the term BBPE to those that use *procedural* languages. For a more thorough introduction to one BBPE, see [29].

Much work has gone into the evaluation of BBPEs. Previous studies have identified what programming concepts students use in BBPEs [22], measured learning gains from classes based on BBPEs [24, 26], and investigated the ease of transitioning from these environments to textual programming [9, 31]. However, these studies evaluate entire programming environments, or even whole curricula, and thus it is difficult to attribute success or failure to any specific aspect of the environment.

This study seeks to isolate the effects of a block interface on the experience of novices when learning to program. To do this, we created two instances of a programming environment, differing only in that one uses a textual programming interface, and one uses a block interface. We collected data from novice, middle-school programmers as they used one of the two interfaces, and analyzed it to answer the following research questions. When compared to a textual interface, how will a block interface:

RQ1. Affect students' attitudes towards computing?

RQ2. Affect their perceived difficulty of programming?

RQ3. Affect their performance on a programming activity?

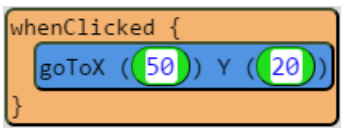

Figure 1: An example of a simple block program, consisting of a procedure call with two literal arguments, nested inside of of an event block.

ICER'15, August 9–13, 2015, Omaha, Nebraska, USA.

ACM 978-1-4503-3630-7/15/08 ...$15.00.
DOI: http://dx.doi.org/10.1145/2787622.2787712 .

To the best of our knowledge, this is the first study to directly compare block and textual programming interfaces in an otherwise controlled setting. Our results contribute to a better understanding of the role that block interfaces play in students' experiences in BBPEs. The following sections cover related work, detail the methodology of this study, and present results, analysis and discussion of the data.

2. RELATED WORK

2.1 Block-based Environments

There are a variety of BBPEs available, many appearing within the last 5-10 years. While most of these environments are developed in academic settings, others, such as Google's Blockly [1] and LEGO Mindstorms [2] emerged from industry. A full review of BBPEs is beyond the scope of this paper, but we highlight two examples, Scratch and Alice, which capture the common characteristics of BBPEs and have been evaluated in a number of setting.

Scratch [29], developed by researchers at MIT, is one of the best-known BBPEs. It was designed to be "more tinkerable, more meaningful and more social" than previous novice programming environments, such as LOGO [27]. To that end, Scratch features primarily graphical output, allowing users to create and manipulate 2D sprites, while adding music, animation and interactivity. The Scratch website allows users to upload, share and remix each other's programs, adding a social element to the environment. Scratch is also notable for its use of executable program fragments. Scratch programs can be built in small chunks, and any piece of Scratch code can be individually executed, with its effects immediately visible. Scratch has become a widely used programming language. As of June 2015, it ranks 25th on the TIOBE index[1], which measures programming languages' popularity based on search engine results.

Scratch has been evaluated in a number of contexts. Meerbaum-Salant et al. [24] designed a two-hour Scratch curriculum and observed its implementation in two ninth grade classrooms. An analysis of student scores on a pre- and post-test of CS concepts showed significant improvement after using Scratch, though students did struggle with more abstract concepts such as initialization, variables and concurrency. Maloney et al. [22] describe their experience using Scratch in an urban after-school center and their analysis of the programs created. Scratch was popular, with students using it voluntarily and more frequently than any other available design software. While around 20% of the projects included only media manipulation without code, about half of the remaining programs employed loops and user interaction, with another quarter using conditional and synchronization statements. Students produced programs of increasing complexity over time.

Alice 2 [8], and its successor Alice 3 [9], are BBPEs which allow users to program within a 3D environment. Alice was one of the first novice programming environments to adopt a drag-and-drop interface. It employs an event-based, object-oriented paradigm, allowing users to add objects to their scene, manipulate the objects' attributes and call their methods. Alice offers a library of 3D objects, animations and sounds, making it a media-rich experience. Alice uses drag-and-drop controls for manipulating lines of code and some expressions, but relies more heavily on menus to give the user access to the many manipulable properties of each object. Alice's interface shares the goal of many BBPEs of simplifying programming by removing the capacity for syntax errors.

Moskal et al. [26] developed an introductory college curriculum using Alice. They compared students who underwent this course prior to, or concurrently with, their first CS1 course to students who took only the CS1 course. They found that the Alice course significantly improved students' grades in the CS1 course, as well as retention in CS over a two year period. They found these trends more apparent with "high-risk" students, who had less math experience and no programming experience prior to college. A similar study by Cooper et al. [8] supports these findings, showing that an Alice-based curriculum helped achieve improved grades and retention. Kelleher et al. [19] found that Alice could be further adapted to young female students by adding features to facilitate storytelling programs, such as easier animations and more character-driven methods.

Other popular BBPEs include MIT App Inventor [28], which allows users to design and program Android apps in a web application. It has been evaluated in K-12 classrooms and summer camps, suggesting it is a powerful, motivational and accessible tool [25], which can serve as a bridge to textual coding in Java [31]. The block interface of App Inventor was turned into a standalone project called Google Blockly [1], which was designed to allow developers to create visual interfaces for their applications. LEGO Mindstorms NXT are customizable LEGO robotics, which can be programmed using a simple block programming environment that employs both procedural and data-flow paradigms. LEGO Mindstorms have been used to teach programming to middle school students [6] and in introductory undergraduate CS courses [7]. Harvey and Mönig created Snap (formerly BYOB) [15], a web-based BBPE based on Scratch, which is being used as the environment of choice for a pilot of an upcoming AP CS Principles course [14].

In addition to their block interfaces, many BBPEs share the following characteristics as well:

- They target novice programmers [8, 15, 28], often younger children during primary or secondary education [9, 19, 25, 29, 31].
- Their programs reflect the syntax and structure of existing programming languages [8, 15, 16].
- They situate programming in a multi-media context, with a focus on cultural relevance [30]. Users can integrate art, music and interactivity into their projects [21, 22], leading to the creation of games [31], stories [19] and apps [28].

We present these *characteristics* separately from our definition of BBPEs, which is concerned only with the block interface. There exist textual programming environments which emphasize these features (e.g. Greenfoot [20]), and BBPEs which do not (e.g. Blockly [1]).

2.2 Comparing Block and Textual Languages

Some studies do compare block and textual programming environments to each other. Lewis [21] compared two groups of 5th grade students participating in a computing summer

[1]http://www.tiobe.com/

camp, over the course of 6 days. One group was taught using Scratch, and the other was given similar lessons using Logo, a textual language. The course was designed to teach "making music, movies and games using computers," and as such its lessons were media-rich. Contrary to the author's hypothesis that Scratch's lack of syntax errors would make learning programming easier, students found the exercises equally difficult in both groups. Students in the Logo condition also expressed more confidence in their computing ability after the activities. Both languages seemed better suited to teaching specific constructs, with Logo students showing a better understanding of loops and Scratch students showing a better understanding of conditionals.

Booth and Stumpf [5] studied adults as they learned to program for Arduino, an electronics platform, through two 20-minute exercises. They compared two conditions, which respectively used a Java-based textual editor and a block-based editor called Modkit, which the authors compare to Scratch. Their results suggest that the block-based editor may have improved completion rates, and that these effects were more pronounced in the activity in which participants were modifying an existing program, rather than creating a new one. They found that the Modkit group found their experience more user friendly and had a lower perceived workload and higher perceived success. While the authors also support their conclusions with quotations from participants, they had a small sample size and did no statistical analysis.

McKay and Kölling [23] used predictive modeling to compare a variety of block and textual programming environments used for education, including Scratch, Alice, Greenfoot, LEGO Mindstorms NXT and Python. Using a prototyping tool called CogTool, they modeled the execution time of a variety of programming tasks in each environment. The results suggest that textual languages are better suited to some tasks, such as insertion and replacement, while block languages are better suited to deletion and movement. Importantly, they also show that there is large variance among block languages, and features such as how the language handles instantiating literals can have a large effect on task time. The authors note that their model does not account for time spent thinking or designing the program, and it is possible that some languages facilitate this better than others.

Other work has investigated the *transition* from BBPEs to textual programming. Wagner et al. [31] introduced K-12 students at a summer camp to programming through MIT App Inventor, but transitioned after two days to the Java Bridge, a Java implementation of the App Inventor API. They found that by repeating exercises first using a block interface and then a textual interface, students were able to mentally map familiar block procedures to the new textual procedures. Dann et al. [9] used Alice 3 in an introductory undergraduate CS course, and transitioned from Alice's original block interface to a Java implementation of the Alice API. Students were given a test at the end of the course, which used Java code in its questions. The authors compared students' scores with those of the previous, all-Java version of the course. They found that the Alice classes performed on average at least one grade level higher than the previous pure Java classes on each section of the test. These studies are important both because they show that skills learned in a BBPE can be transferred to a textual environment, and because they serve as examples of textual environments that can offer the same media-rich features of BBPEs without a block interface.

2.3 Visual Programming Languages

The programming languages employed by BBPEs are often classified under the larger category of Visual Programming Languages (VPLs). In their taxonomy of novice programming environments, Kelleher and Pausch [18] categorize Alice 2 and other early BBPEs like LogoBlocks [3] as environments trying to "find alternatives to typing programs," specifically by "constructing programs using graphical... objects." A number of authors also refer to specific BBPEs as visual programming languages or environments [5, 21, 22, 31]. While we agree that BBPEs are visual, we find it important to distinguish them from more traditional VPLs, such as spreadsheets, flowcharts and the LabView language [32]. This distinction is important due to the body of research comparing these VPLs to textual languages, which may not be applicable to BBPEs.

Historically, the evidence supporting VPLs has been mixed [32]. For instance, Greene and Petre [11] compared programmers' ability to read and comprehend LabView with a simple textual programming language. They found that VPL comprehension was slower for all programmers, regardless of whether their past experience was with LabView or a textual language. The authors later analyzed two dataflow VPLs along a cognitive dimensions framework [12], finding them lacking in a number of dimensions compared to their textual alternatives. For example, they found that the VPLs had higher *Viscosity*, the amount of effort required to make a small change to a program, and that they forced *Premature Commitment*, making users commit to code structure before their programs are fully formulated. We reference these studies to differentiate their work on *dataflow* VPLs from more modern research on BBPEs.

3. METHOD

While many studies have evaluated BBPEs for their effectiveness in engaging students, making programming accessible, or teaching CS concepts, these studies have evaluated the environments holistically. This makes it difficult to assign success to any single aspect of the environment, including the block interface. Since many BBPEs employ a media-rich environment, for instance, perhaps their success is due primarily to this fact, and not to their novel interfaces. Even those studies which control for content when comparing BBPEs with their textual counterparts [5, 21] are still comparing two different programming environments (e.g. Scratch and Logo), which may differ in a number of ways outside of their programming interfaces that could account for different outcomes. To address this, our methodology was designed to isolate the effect of the programming interface on novice students as they learned to program.

3.1 The Environment

For our programming environment, we chose Tiled Grace [16], a web-based environment that implements both block and textual programming interfaces, and even allows the user to switch between the two when working. The "tiled" version of Grace supports the usual features of a BBPE, with drag-and-drop code blocks. These blocks correspond directly to constructs in the Grace programming language, which is also supported by the editor. A program

written with either interface will consist of the same text in the same general layout, but in the block interface this text is contained within blocks. We created two versions of the Tiled Grace environment, which were locked into either the block or textual interface, but were otherwise identical. Both versions of the environment can be seen in Figure 2.

We also performed some minor changes to the environment to make the block interface more similar to other BBPEs. The authors of Tiled Grace note that the coloring of their blocks was essentially arbitrary, so we colored blocks by functionality, as is done in other BBPEs (e.g. one color for control structures, variable manipulation, etc.). The authors also note that while other BBPEs use block and hole shapes to indicate how blocks should fit together, the authors leave this for future work. As a simple improvement, we added rounded corners to expression blocks (such as literals or functions that return a value), to indicate that these block could be placed into parameter holes. The shapes were only guidelines, and any block could still be placed in any hole, which is not true of some BBPEs. While Tiled Grace offers a palette of usable blocks, there is no equivalent for textual coding. To keep the two interfaces as similar as possible, we also added a "Code Palette," consisting of equivalent blocks of sample code. Where the blocks had holes to indicate where other blocks should be placed, the Code Palette snippets had dummy values for method parameters and comments indicating where lines of code could be added. Expression code snippets had a blue background, similar to the rounded corers of the block interface. We found these to be additions that a novice textual programming environment could reasonably implement for a specified domain.

Since our goal was to evaluate the interface in the context of novice programmers, we developed a programming exercise based on an Hour of Code activity from the Snap website [15], a tutorial designed to introduce novices to programming for the first time. The exercise had users create a simple web-based game, similar to whack-a-mole, in which users attempt to click on a sprite as it jumps around the screen to win points. Many BBPEs support the creation of similar, simple games. The exercise was split into 9 sections, with tutorial text introducing each one. Each section introduced a new goal, which often built off of previous goals. The tutorial text was the same for both versions of Tiled Grace, except where the differing interfaces necessitated changes (referring to "blocks" instead of "code"). A finished project required the use of various programming concepts, including events, loops, variables and conditionals.

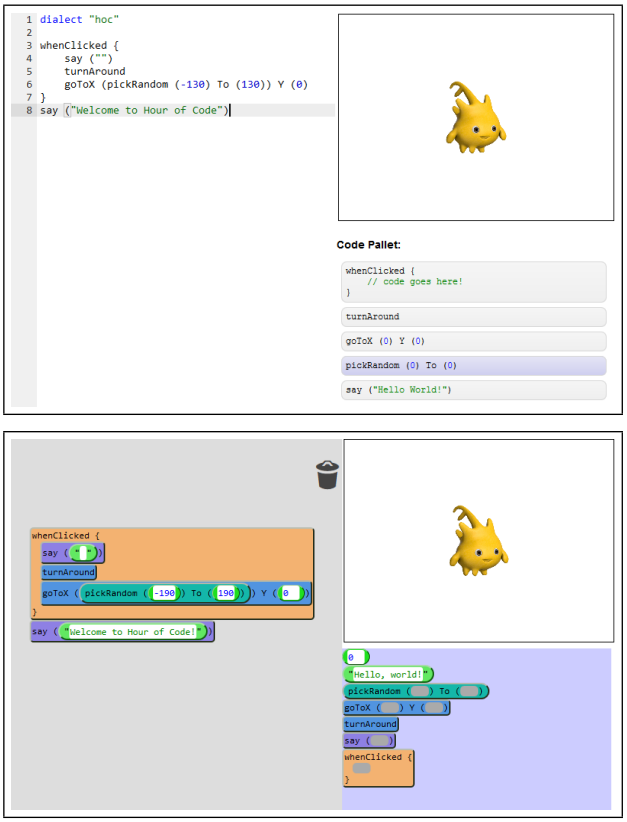

Figure 2: A comparison of the textual (top) and block (bottom) interfaces. The coding interface for both consists of a work area (left), an output canvas where the game can be played (top-right) and a palette of usable blocks or code (bottom-right).

3.2 Procedure

This study took place as part of a middle school STEM outreach program called SPARCS [6]. The program, which meets for half-day sessions approximately once a month during the school year, consists of lessons designed and taught by undergraduate and graduate students to promote technical literacy. We worked with a group of sixth graders, who were randomly assigned to use the block interface, and a group of 7th graders, who were assigned to use the textual interface. We chose to assign the conditions by classroom, rather than by student, to avoid confusion from students within a classroom seeing different interfaces. The Block group consisted of 17 6th grade students (12 male, 5 female). The Text group consisted of 14 7th grade students, (11 male, 3 female). We took measures to test for population differences between the classrooms, which are explained in Section 5. The Block group participated in the study in the late morning, while the Text group held an unrelated lesson on security, with no content that addressed programming. The text group participated in the study approximately 3 hours later, in the early afternoon. The students in both groups had participated in previous SPARCS sessions that semester, but none had participated in previous years.

In each classroom, the first author led the study. Before starting, the students were directed to complete a brief pre-survey (covered in Section 4.1). The programming exercise was then explained to the students. The students were allowed to go through the exercise at their own pace. If they had questions, the students were allowed to ask for help from the student volunteers, and the volunteers made a note of the type and duration of assistance offered. After finishing the exercise, the students were allowed to continue to work on their game. After 45 minutes, the students were directed to take a post-survey (covered in Section 4.1). The programming environment was instrumented to log most student interactions, such as button presses, block drags and compilations. Snapshots of each student's code were saved at regular intervals and each time it was run.

Occasional technical issues did occur in both groups. One student in each group had severe technical issues, and these students were excluded from analysis (and are not reflected

Efficacy: Please say how confident you are that you can do each of the following tasks. I can:
- Use a computer to solve a problem
- Write a computer program
- Create something interesting using a computer
- Explain how a computer works

Choices: Strongly Disagree, Somewhat Disagree, Neutral, Somewhat Agree, Strongly Agree

Interest: Please say how likely you think you are to do each of the following in the future:
- Take a programming or Computer Science class
- Create a computer program, app or game for fun
- Learn more about programming on your own

Choices: Very Unlikely, Somewhat Unlikely, Undecided, Somewhat Likely, Very Likely

Table 1: Efficacy and Interest survey questions.

```
var x := 7
x := x + 2
print(x)
```

```
var y := 8
if (y > 4) then {
  y := y - 5
}
print(y)
```

```
var z:= 0
forever {
  if (z < 5) then {
    print z
  }
  z := z + 1
}
```

Figure 3: Students were asked to determine the output of these programs in the pre- and post-surveys.

Type (Group)	Pre	Post	Change
Efficacy (B)	3.51 (0.90)	3.88 (0.73)	0.250 (0.50)
Efficacy (T)	3.59 (0.70)	3.75 (0.93)	0.167 (0.57)
Interest (B)	4.24 (0.70)	4.26 (0.53)	0.128 (0.42)
Interest (T)	3.76 (0.71)	3.93 (0.81)	-0.037 (0.42)
Knowledge (B)	1.53 (0.94)	1.30 (0.95)	0.100 (0.99)
Knowledge (T)	1.14 (0.86)	1.13 (0.99)	0.125 (0.64)

Table 2: Results (and standard deviations) from the pre- and post-survey. Efficacy and Interest scales were from 1-5. Knowledge scores range from 0-3. The Post and Change columns are computed only for the students who took the post-survey. B and T indicate Block and Text groups.

in the counts above). Further, one student in the Text group had a parent present (this is not typical), who offered significant help, and this student's data was also excluded from analysis. Some students in the Block group arrived late, and the group was given more time to compensate; however, only the first 45 minutes of any given student's work is analyzed (not including time spent taking the pre-survey).

4. RESULTS

4.1 Survey Results

One set of survey questions was presented to students in both the pre- and post-survey. This was done to account for initial differences in attitudes towards computing between the two groups. This set of questions consisted of three sections. The first section assessed students' self-efficacy with regards to computing, and the second assessed students interest in computing in the future. These sections consisted of 3-4 Likert items, which are presented in Table 1. The last section consisted of three knowledge-based questions, in which students were asked to evaluate the output of a code routine. The code was presented as blocks or text to match the interface the student was using. The code used in these questions is presented in Figure 3. The questions were identical in the pre- and post-surveys, except that the Knowledge questions had numeric values changed on the post-survey. We calculated averages from the Likert questions to produce a numeric value for each student (1-5) for Efficacy and Interest questions in both the pre- and post-survey. We also calculated the number of correct answers in the Knowledge section (0-3) for both surveys. A summary of the results is presented in Table 2.

Since SPARCS is a voluntary program, we could not force students to take the surveys. In order to start the activity, the students did have to complete the pre-survey, but despite strong encouragement, some chose not to complete part or all of the post-survey. In the Block group, 15 of 17 students completed part or all of the post-survey, and in the Text group, 9 of 14 students did so. Students who failed to complete the post-survey were excluded from our analysis of survey results but were included in log data analysis.

A second set of questions was asked of both groups only in the post-survey. These questions assessed the user's experience when performing the activity, and asked them to rate their difficulty performing certain tasks within the activity. Both groups of questions received very similar ratings across conditions. The results of the difficulty questions can be seen in Figure 4. Finally, demographic data was collected, along with questions about the students' access to technology.

4.2 Logged Interactions

Each time the student performed an action within the environment, it was timestamped and logged to a database. Some actions were specific to one interface, such as drag and drop actions. Other actions, such as advancing to the next section of the tutorial, were used in both interfaces. From these logs, the time spent on each section of the tutorial was calculated for each student. Students were able to skip ahead and revisit sections, so the time spent on a given section may be divided among multiple visits. Idle time, defined as going more than 60 seconds without modifying code, was also calculated for each student. Though students were strongly encouraged to use all available time, some students also chose to end the activity early. The duration each student spent in the activity was also calculated.

The students' programs were also saved to the database after each edit. An ideal finished program (not from the collected data) is shown in Figure 5. Programs were analyzed for goal completion, as explained in Section 5.2.

5. ANALYSIS

We compared pre-survey and demographic data from both conditions to determine if there were significant differences between groups. In this analysis we did include students who did not finish the post-survey. We investigated results from each of the questions presented in Table 1, and the data did not appear normally distributed; therefore, a

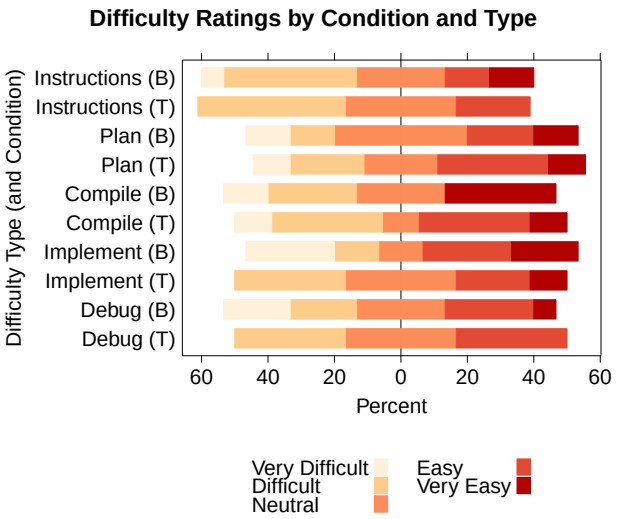

Figure 4: The distribution of difficulty ratings given by students in each condition. The questions, in order, asked users to rate their difficulty understanding instructions, deciding what to do, getting the program to run (compile), implementing a solution and figuring out what went wrong (debugging).

Mann-Whitney U Test was performed[2] to determine if one group had significantly higher average ratings on any of the three questions. The Block group did have a significantly higher Interest rating ($W = 163$, $p = 0.040$). No other differences between the groups were significant.

Demographic information included questions about access to computers, frequency of computer use and past computing experiences. These questions had similar results between groups, and no differences were significant. Lastly, both groups received an almost identical amount of volunteer assistance during the experiment. Since there were few differences between the two groups, we determined that they could be compared directly, but the difference in Interest ratings should be noted. We performed three primary analyses, covered in the following sections.

5.1 Survey Analysis

The survey questions shown in Figure 1 were repeated in both the pre- and post-survey. We first wanted to determine if the activity had an effect on participants' answers to these questions. Only the Efficacy scale shows a meaningful improvement. The data did not appear normally distributed, so we performed a Wilcoxon Signed-Rank Test and found the improvement to be significant ($V = 102$, $p < 0.040$, Cohen's $d = 0.289$). Note that this effect is present when considering both conditions together, but not strong enough to be significant in either condition alone. The effect is more pronounced in the Block group, and we compared the improvements of both groups using a Mann-Whitney U Test, but the difference was not significant ($W = 62$, $p = 0.412$).

We investigated the Efficacy Likert items individually to determine which contributed to the improvement. We tested each item using a Wlxocon Signed-Rank Test and used the

[2]All statistical analysis was performed using the R statistical software package.

```
1  dialect "hoc"
2                                  Goals:  1 2 3 4 5 6 7 8 9
   var delay := 2                  Colors:
   var score := 0
   var maxScore := 0
   whenClicked {
       say ""
       delay := delay - 0.1
       score := score + 10
       if (maxScore < score) then {
           maxScore := score
           clear
       }
       forever {
           goToX( pickRandom (-190) To (190) )
               Y ( pickRandom (-130) To (130) )
           turnAround
           if (score > 0) then {
               score := score - 1
           }
           wait (delay)
       }
   }
24
25 goToX (-190) Y (130)
   say ("Welcome to the hour of code")
   penDown
```

Figure 5: The target finished program. Colors indicate the goals to which each line of code corresponds.

Benjamini-Hochberg procedure [4] to control the False Discovery Rate (FDR) at 0.05. The items which showed a significant improvement were Item 2, "I can write a computer program" ($V = 16.5$, $p = 0.006$), and Item 4, "I can explain how a computer works" ($V = 4.5$, $p = 0.005$). Surprisingly Item 3, "I can create something interesting using a computer," showed a significant decline ($V = 24.5$, $p = 0.035$). This may be in part the result of very high pre-suvey ratings. The distributions of each question can be seen in Figure 6.

The remaining questions were only present in the post-survey, as they were about the completed activity. This included questions about the user's experience using the interface, and their difficulty completing the activity. Responses appeared to have similar distributions in both conditions, and Mann-Whitney U Tests confirmed that there were no significant differences between conditions.

These analyses suggest that the activity did have some positive impact on students' self-efficacy regarding computing, though for such a short activity it is not surprising that no other effects were observed. Still, the results offer little evidence to support a claim that the interface affected students' attitudes towards computing. Most surprising is that it seems to have had no impact on students' perceived difficulty, despite pronounced differences in student behaviors in the system, as explored in the following sections.

5.2 Time Analysis

As discussed in Section 4.2, the time each student spent on the activity was calculated, including how much of that time was spent active or idle. Results are shown in Table 3. While the interface did not appear to affect the duration

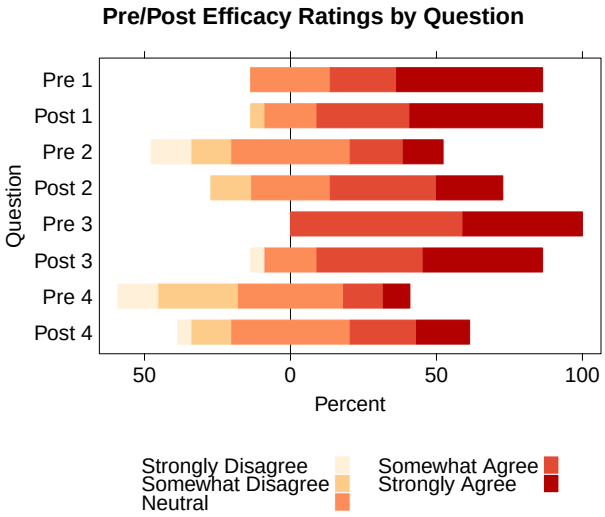

Figure 6: The distribution of ratings given by students in the pre- and post-survey for each Efficacy item (see Figure 1 for the items' text).

Value	Block	Text	p	*d*
Total	2273.9 (596.4)	2208.0 (427.1)	0.851	–
Idle	407.2 (238.9)	793.5 (368.3)	0.002	1.27
Active	1866.8 (617.4)	1414.5 (463.1)	0.014	0.82

Table 3: Average total, idle and active time in seconds for both groups (with standard deviations). The differences in idle and active time are significant, and Cohen's *d* is given.

spent on the activity, it did have a significant effect on both idle and active time.

5.3 Completion Analysis

Each section of the activity had a goal, stated in the instructions, which could be uniquely accomplished with the blocks and concepts introduced in that section. Snapshots of each participant's program were analyzed to determine if and when each of the sections' goals were met. Goal specifications were designed to be independent of the rest of the program; thus, a student could accomplish a section's goals even after skipping previous sections of the activity, which did occur. Since programs with syntax errors are inherently ambiguous and could not be tested by students, only compilable snapshots of programs were analyzed. While we believe this requirement to be reasonable, it likely had a disproportionate effect on textual programs. The analysis was automated, but we checked it for correctness manually on 1/8 of the students. While at least one student completed each goal, no students completed the 8th or 9th goal within the first 45 minutes analyzed here.

Figure 7 shows the percentage of students who completed each goal, as well as the percentage who viewed that goal for at least 10 seconds. The Block group outperformed the Text group in all respects by these measures. Of particular interest is Goal 4, which introduced loops. A nearly equivalent percent of participants from both groups viewed the problem, but many more from the Block group completed the

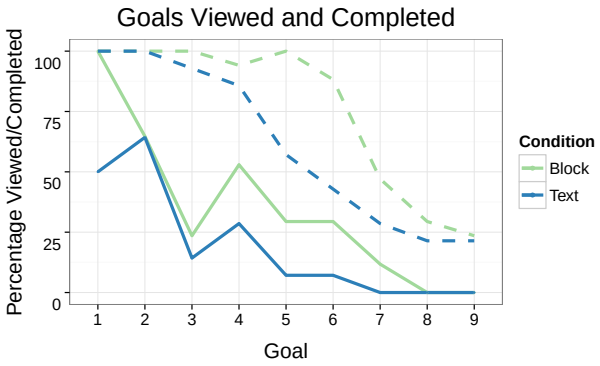

Figure 7: For each condition, the solid line shows the percentage of students who completed each goal. The dashed line shows the percentage of students who viewed each goal for at least 10 seconds.

goal to specification. Goal 5 then shows a marked dropoff in viewers for the Text group, but an increase for the Block group. This implies a logical relationship between the completion of a goal and viewing the next goal.

Figure 8 shows the average total time that had passed before students completed each goal. Each bar only includes the students who completed the given goal. On sections where at least 25% of students in both conditions completed the goal, a Mann-Whitney U Test was performed (data did not appear normally distributed) to determine if the difference in completion time was significant, with the FDR controlled at 0.05. For each of these goals it was significant: Goals 1 ($W = 13, p < 0.001$), 2 ($W = 21.5, p = 0.018$) and 4 ($W = 4, p = 0.017$). This seems sufficient evidence to assert that the interface significantly increased the rate at which students completed programming goals.

Pearson correlations were calculated between the number of goals completed and a variety of the survey questions, including pre- and post-survey Efficacy, Interest and Knowledge scores, and reported difficulty, and none appeared meaningful, or had a magnitude greater than 0.35.

6. DISCUSSION

We now revisit our original research questions. Compared with the textual interface, how did the block interface:

RQ1 *Affect students' attitudes towards computing?* This question was addressed primarily by the Efficacy and Interest questions in the pre- and post-surveys. While we did observe significantly improved responses to the Efficacy questions as a whole after the activity, two items primarily accounted for this shift. The most relevant item, "I can write a computer program," was one of these. It is unclear why one item, "I can create something interesting using a computer," showed a decline. It may be a reflection of the students' opinion of the programming activity, more than their self-efficacy with regards to computing. Regardless, there was no significant difference in improvement on these questions between conditions. It is quite possible that the effect was simply too small to be observed in the relatively small sample size of this study. It is also possible that a longer activity would have produced more pronounced changes, and these would have been more dependent on the programming in-

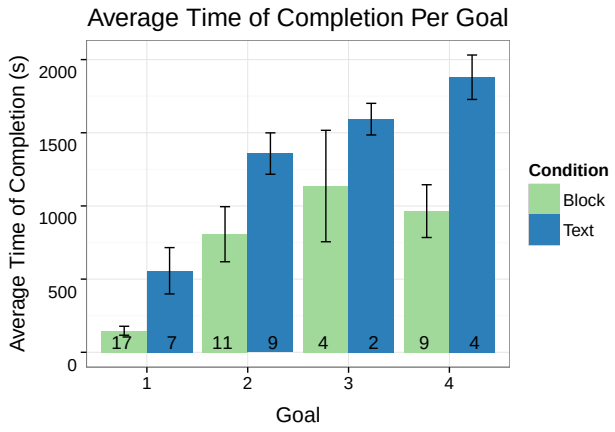

Figure 8: The average total time spent in the activity before completing each goal, with bars indicating standard error. The numbers at the bottom of each bar indicate the number of students who completed the goal. Values are not strictly increasing, in part because goals could be completed out of order. Goals 5-9 are not shown, as at most 1 student from the Text group completed them.

terface. Interpretation is further complicated by the higher pre-survey Interest ratings in the Block group. While we certainly cannot argue against the possibility that the interface has an impact on user attitudes towards computing, we can offer no evidence to support it.

RQ2 *Affect students' perceived difficulty of programming?* There is no survey evidence to support the claim that the Block group experienced less difficulty than the Text group, either in general or on specific aspects of programming. This matches similar findings in previous work [21]. Interestingly, there was little correlation between perceived difficulty and goal completion. One possible explanation for both of these observations is that the free-form nature of the exercise allowed students to progress until they hit an aspect of the activity that was challenging, and that is where they spent the majority of their time. Even if the block interface helped students overcome confusion about the language's syntax, perhaps they simply moved on to other, equally challenging, aspects of programming. In this case, one might still expect students to emphasize the difficulty of different aspects of programming, depending on their interface, and this was not observed. This may be due to novices' inability to distinguish sources of difficulty in programming.

RQ3 *Affect students' performance on a programming activity?* Whether measured by self-pacing, completion, efficiency or time on task, the Block group did demonstrate increased performance. Not all of these observations are easily quantified or statistically tested, but in combination they seem to conclusively show that the block interface did improve performance.

6.1 Limitations

It is worth stating that none of the findings in this paper should be casually generalized to other contexts. The findings with regard to RQ3 are likely the most robust, but some limitations should be considered.

Tiled Grace lacks some of the common features of other block-based languages, such as "jigsaw" pieces that only snap into legal locations. Conversely, while Grace was designed to teach CS, it might be considered more complex syntactically than other textual languages, due to its use of meaningful whitespace and method names which are split between arguments (e.g. `goToX( 10 )Y( 20 )`). However, these differences are likely no more extreme than those between most programming languages.

The activity studied here was adapted from one designed for a BBPE. It is possible this activity unfairly emphasizes aspects of programming which are advantageous for BBPEs. For instance, the first section of the activity used event handling, which was syntactically trivial in Tiled Grace, but involved nesting one command inside of another, which is much more complex in textual Grace. This is evidenced by the lack of completion of Goal 1 by the Text group, as shown in Figure 7. Even if the activity highlighted advantages in block interfaces, this means that such advantages do exist, which is still a meaningful finding.

The lack of responses to the post-survey make it difficult to draw strong conclusions from it. It is also likely that some students did not take the surveys seriously, but this is difficult to verify. Further, the survey questions, while typical of validated instruments, were not themselves validated. They were also kept short to avoid survey fatigue. This may explain why we observed contradictory responses to the Efficacy Likert items and why the surveys do not seem to reflect the differences observed in the log data.

Finally, it should be noted that the Text group was a full grade higher than the Block group. While this makes the success of the Block group more notable, it may have impacted the study in unforeseen ways. The Block group also had significantly higher Interest ratings in the pre-survey.

6.2 Future Work

Inconclusive results from the survey questions indicate the need for further study of the effect of a block interface on students' perceptions of programming, specifically with improved survey design and larger samples. The discrepancy between students' performance on the activity and their perceived difficulty with the activity merits further investigation, as well.

Though this study does support the claim that block interfaces improve programming performance in novices, it would be useful to investigate whether this remains true for first or second year programmers. Further, while this study does show that block interfaces reduce idle time and increase goal completion, it does not suggest a mechanism for these changes. While some answers may seem intuitive, an investigation of *how* block interfaces reduce idle time, or facilitate goal completion would be fruitful, as a better understanding of the underlying causal relationships could lead to the creation of better BBPEs in the future.

7. CONCLUSION

This study supports the claim that block programming interfaces can significantly improve novice performance on some programming activities, specifically through increased time on task and quicker, more frequent achievement of programming goals. The study suggests that the block interface is an important component of BBPEs, which is worthy of future study and development.

8. REFERENCES

[1] Google Blockly. A Visual Programming Editor. http://code.google.com/p/blockly/.

[2] LEGO Mindstorms. http://mindstorms.lego.com.

[3] A. Begel. LogoBlocks: A graphical programming language for interacting with the world. *Electrical Engineering and Computer Science Department, MIT, Boston, MA*, 1996.

[4] Y. Benjamini and Y. Hochberg. Controlling the false discovery rate: a practical and powerful approach to multiple testing. *Journal of the Royal Statistical Society. Series B (Methodological)*, pages 289–300, 1995.

[5] T. Booth and S. Stumpf. End-user experiences of visual and textual programming environments for Arduino. *End-User Development*, pages 25–39, 2013.

[6] V. Cateté, K. Wassell, and T. Barnes. Use and development of entertainment technologies in after school STEM program. In *Proceedings of the 45th ACM technical symposium on Computer science education*, pages 163–168, 2014.

[7] D. C. Cliburn. Experiences with the LEGO Mindstorms throughout the Undergraduate Computer Science Curriculum. In *Frontiers in Education Conference, 36th Annual*, pages 1–6, 2006.

[8] S. Cooper, W. Dann, and R. Pausch. Teaching objects-first in introductory computer science. *ACM SIGCSE Bulletin*, 2003.

[9] W. Dann, D. Cosgrove, and D. Slater. Mediated transfer: Alice 3 to java. In *Proceedings of the 43rd ACM technical symposium on Computer Science Education*, pages 141–146, 2012.

[10] D. Garcia, B. Harvey, L. Segars, and C. How. AP CS Principles Pilot at University of California, Berkeley. *ACM Inroads*, 3(2), 2012.

[11] T. Green and M. Petre. When visual programs are harder to read than textual programs. In *Tasks and Organisation, Proceedings of ECCE-6 (6th European Conference on Cognitive Ergonomics)*, 1992.

[12] T. Green and M. Petre. Usability analysis of visual programming environments: a 'cognitive dimensions' framework. *Journal of Visual Languages & Computing*, 1996.

[13] M. Guzdial. Programming environments for novices. In S. Fincher and M. Petre, editors, *Computer Science Education Research*, pages 127–154. Taylor & Francis, 2004.

[14] B. Harvey. The Beauty and Joy of Computing: Computer Science for Everyone. In *Proceedings of Constructionism*, pages 33–39, 2012.

[15] B. Harvey and J. Mönig. Bringing "No Ceiling" to Scratch: Can One Language Serve Kids and Computer Scientists? In *Proceedings of Constructionism*, pages 1–10, 2010.

[16] M. Homer and J. Noble. Combining Tiled and Textual Views of Code. In *Proceedings of 2nd IEEE Working Conference on Software Visualization*, 2014.

[17] T. Jenkins. On the difficulty of learning to program. In *Proceedings of the 3rd Annual Conference of the LTSN Centre for Information and Computer Sciences*, 2002.

[18] C. Kelleher and R. Pausch. Lowering the Barriers to Programming: A Taxonomy of Programming Environments and Languages for Novice Programmers. *ACM Computing Surveys*, 37(2):83–137, June 2005.

[19] C. Kelleher, R. Pausch, and S. Kiesler. Storytelling alice motivates middle school girls to learn computer programming. *Proceedings of the SIGCHI conference on Human Computer Interaction*, 2007.

[20] M. Kölling. Greenfoot - A Highly Graphical IDE for Learning Object-Oriented Programming. *Journal of Computer Science*, 13:60558, 2008.

[21] C. Lewis. How programming environment shapes perception, learning and goals: logo vs. scratch. In *Proceedings of the 41st ACM technical symposium on Computer science education*, pages 346–350, 2010.

[22] J. Maloney, K. Peppler, Y. Kafai, M. Resnick, and N. Rusk. Programming by choice: urban youth learning programming with scratch. *ACM SIGCSE Bulletin*, 40(1):367–371, 2008.

[23] F. McKay and M. Kölling. Predictive modelling for HCI problems in novice program editors. In *Proceedings of the 27th International BCS Human Computer Interaction Conference.*, pages 35–41, 2013.

[24] O. Meerbaum-Salant, M. Armoni, and M. Ben-Ari. Learning computer science concepts with scratch. In *International Computing Education Research Conference 2010 (ICER '10)*, pages 69–76, 2010.

[25] R. Morelli, T. de Lanerolle, and P. Lake. Can android app inventor bring computational thinking to k-12. In *ACM technical symposium on Computer science education (SIGCSE'11)*, 2011.

[26] B. Moskal, D. Lurie, and S. Cooper. Evaluating the effectiveness of a new instructional approach. *ACM SIGCSE Bulletin*, 36(1):75–79, 2004.

[27] S. Papert. *Logo Philosophy and Implementation*. Logo Computer Systems Inc., 1999.

[28] S. Pokress and J. Veiga. MIT App Inventor: Enabling personal mobile computing. In *Workshop on Programming for Mobile and Touch*, 2013.

[29] M. Resnick, J. Maloney, H. Andrés, N. Rusk, E. Eastmond, K. Brennan, A. Millner, E. Rosenbaum, J. Silver, B. Silverman, and Y. Kafai. Scratch: programming for all. *Communications of the ACM*, 52(11):60–67, 2009.

[30] I. Utting, S. Cooper, and M. Kölling. Alice, Greenfoot, and Scratch–a discussion. *ACM Transactions on Computing Education (TOCE)*, 10(4), 2010.

[31] A. Wagner, J. Gray, J. Corley, and D. Wolber. Using app inventor in a K-12 summer camp. In *Proceeding of the 44th ACM technical symposium on Computer science education*, 2013.

[32] K. Whitley. Visual programming languages and the empirical evidence for and against. *Journal of Visual Languages & Computing*, 8(1), 1997.