Filling the Gap in Programming Instruction: A Text-enhanced Graphical Programming Environment for Junior High Students

Joey C. Y. Cheung, Grace Ngai, Stephen C. F. Chan and Winnie W.Y. Lau
Department of Computing,
The Hong Kong Polytechnic University
Hung Hom, Hong Kong
{cscycheung, csgngai, csschan, cswylau}@comp.polyu.edu.hk

ABSTRACT
To address the unique demands and challenges of educational computing, various kinds of environments, including graphics-rich and textual environments, have been proposed for use in introductory courses to provide students with a rich and interesting learning environment. In our experience, students in Grade 7 and younger respond best to the graphics environments while senior high school students prefer a conventional textual programming environment. Clearly, this leaves a gap at Grade 11-13, with students often on the one hand finding the graphics-based environments too limited and on the other finding the textual environments too difficult. In this paper, we propose a text-enhanced graphical programming environment which is innovative and interactive, and designed for junior high students with no programming experience. This environment allows students to design their own creative stories or programs. They build their programs using drag-and-drop iconic blocks, but unlike other, similar icon-based programming languages, they are also presented with the syntax of the actual program they are constructing in real-time. Once a particular icon block has been dropped in the programming area, the syntax statements corresponding to that block is immediately generated and presented to the user. The environment also allows them to modify the code without any limitations. Our results show that our textual-graphical hybrid environment has a positive impact on the learning experience of the students.

Categories and Subject Descriptors
K.3.2 [**Computers and Education**]: Computer and Information

Science Education—*computer science education*

General Terms
Design, Human Factors, Languages

Keywords
Education, programming environments

SIGCSE'09, March 3–7, 2009, Chattanooga, Tennessee, USA.

1. INTRODUCTION
There has been much development in graphical-based educational programming software. However, we observe that there exists a lack of programming instructional materials that bridges the gap between the purely graphical programming environments appropriate for elementary school and the more complex text-based programming languages used in high school. Over the last few years, we have launched numerous programming workshops targeted at programming novices aged from 8 to 18 years old, using a number of different programming environments and software. Some of our workshops aim to teach children programming through using the Scratch software [8]. These workshops have been offered to both elementary school children as well as junior high school students. We observe that while Scratch is a suitable and enjoyable environment for the elementary school children, the junior high students complained that they felt bored as the environment was too simple and limited. They even stated that they would prefer learning syntax directly. However, in another workshop designed for senior high school students (Grades 11-13), in which the C programming language was taught and used, we realized that learning programming syntax was a very difficult task even for these more advanced students. Many of them could not complete the assigned tasks both because of the syntactic and logical errors.

We therefore see that there is a gap between the purely-graphical programming environments appropriate for elementary school students and the conventional, textual programming languages appropriate for more advanced high school and university students. Our target audience are the junior high school students (around Grades 8-10), who find the graphical based languages not challenging enough, but are still not yet advanced to go on to textual programming.

We propose BrickLayer, a text-enhanced graphical programming environment as a bridge over the junior high school level gap. BrickLayer is designed to arouse the learning interests of students as well as to guide them to learn both programming logic and syntax based on their ability. Within this programming environment, specific syntax statements are generated immediately in the range of the students' visibility once they dragged a block to a particular place. In this case, students can instantly know what happens in the coding when they are writing their stories or animations using an interactive way. We surveyed students throughout the workshop that deployed the proposed environment about their thoughts on this environment and the imposed impact on their further studies. Most of the junior high

school students felt this interesting environment could build up their confidence to modify the code without any human instructions on syntax rules. Others felt that the graphics with limited script variations obstructs the progress in designing their outcomes since they have prior programming experience. As every child owns different programming backgrounds, we should offer them a suitable environment that can be adapted to their needs and preferences.

The remainder of this paper is organized as follows: In Section 2 we describe background and motivation of the text-enhanced graphical programming environment in detail. In Section 3 we discuss the deployment of this environment. Section 4 provides the results of our trial. Section 5 offers our conclusion.

2. BACKGROUND & MOTIVATION

Numerous of graphics programming environments such as Scratch and Alice [3] are designed to teach programming to students ranging from 8 years old or above. These environments provide iconic elements and allow children to create their own stories and animations in using an interactive way by dragging and dropping different screen objects like boxes or bubbles onto a construction area. The iconic element acts like a puzzle that allows a beginner to design his programs by using a mouse to fit pieces together. Behind the graphical front-end, the source code for the program is generated. However, these source codes are not visible to the student; as far as he/she is concerned, the program is the graphical display in the construction area.

There has been much previous work [1, 4, 5, 10, 11] that verifies that students enjoy creating games and animations in these environments, as they can get a tangible and immediate reward. Under these kinds of environments, students do not only have fun, but they also gain programming knowledge in many of the concepts that are taught in computer science foundation courses. Therefore, these environments have often been recommended to first-time programmers who do not possess any computer programming background.

In our experience, however the limitations of graphical programming means that it fails to provide an adequate challenge for junior high school students. In general, these kinds of environments tend to over-protect the students, only providing restricted functions and limited freedom for creation. As a result, students are restricted to simple animations and cannot design complex creations. In addition, since the programming environment is simplified, they do not face any challenges in learning to program and get bored, which lowers their motivation and intention to learn. We noted that students aged about 12 to 14 usually do own the ability to recognize some textual commands, so it is not surprising that they would prefer to learn some syntax statements.

At the opposite extreme are the textual programming environments, which offers a text editing area in which students type their programming statements. This is the mode of learning conventional programming languages, such as Java, Visual Basic and C++, which is taught in many high schools. In these environments, the logical thinking behind programming is introduced together with the syntactic constraints of the language. Much previous work has shown that one major problem with these environments is that the programming language and its syntax becomes the focus, rather than the problem solving skills [7], and lectures often end up concentrating on syntactic issues rather than the logic problems. The strength of the students' problem solving skills therefore depends on their personal development instead of the computing training that they receive in course.

Despite these problems, past experience has shown that senior high and university students can handle textual programming. However, in our experience, junior high school students tend to find logical problems difficult enough to deal with, without adding in syntactic concerns as well. It is therefore very difficult for younger novices to become a syntax master as well as a problem solver.

Based on the above concerns, our goal is to design a text-enhanced graphical programming environment that allows junior high school students to easily build their own programs and learn the syntax simultaneously. In order to lower the barriers to programming, our environment allows the students to drag-and-drop the given blocks or icons to create their programs. Once the items are dropped onto the construction area, the corresponding code is immediately generated and can be viewed. The advantage of this environment is that students can learn programmatic constructs and focus on logic issues without worrying about the syntax. At the same time, presenting them with the corresponding statements to their icons allows them to get a better idea of what programming entails, and also lets them learn some elements of syntax if they are interested.

3. The BrickLayer Environment

To fulfill our requirements, we designed and implemented a text-enhanced graphical programming environment for junior high school students called BrickLayer. Similar to a real bricklaying exercise, students drag blocks (or bricks) which represent each programming construct and drop them onto an "icon-laying" area to create a "wall". The corresponding source code is instantly generated and displayed in an onscreen code area.

BrickLayer is written in Javascript, which allows it to be run over a web browser. It was also designed to support the programming needs of a wearable computing workshop, which aims at teaching children basic electric and electronic concepts with programming. Students create programs to process signals from sensors such as light sensors and accelerometers, and to control output to actuators such as LEDs and motors. The generated source code is in C and is written to be executed on the open-source Arduino microcontroller [2].

One problem with graphics-based programming environments is that it is often difficult for users to determine the scope of programming constructs. For example, consider the case of a nested `if` statement that consists of an `if` statement inside the scope of another `if-else` statement. If only graphical icons are used, it is often not easy for users to trace the scope of the conditional statement from the dropped blocks. In order to aid the user, BrickLayer assigns a unique color to each conditional or looping block, and all icons within that block are rendered with that color. That allows the user to more easily trace and debug their program (see Figure 1).

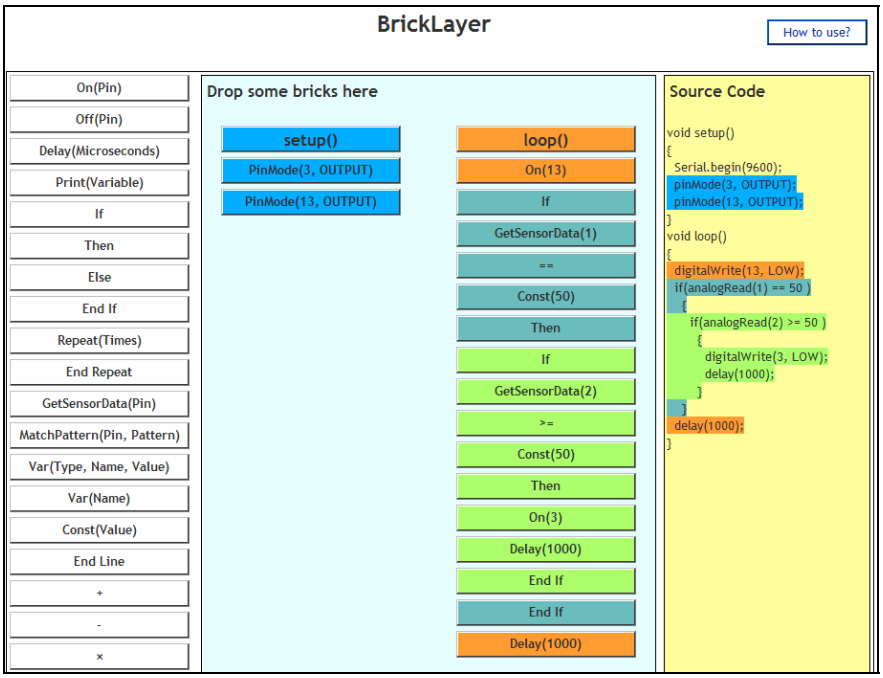

Figure 1: BrickLayer's interface consists of a block area (left), construction area (middle) and source code (right). The block area is composed of constructs. The construction area is provided for dropping blocks. The source code area shows the programming statements that correspond to the dropped icons.

As BrickLayer is designed to be a teaching tool, it had to accommodate students who have no programming background. On the one hand, we wished to help them avoid too much early discouragement or frustration, and on the other, give them feedback about errors that they would need in order to learn. To this end, the BrickLayer environment was developed with three types of constructs: intelligent, fundamental and user-defined. Intelligent constructs are those that provide checks and balances that protect the student from making careless mistakes. For example, one of the most basic tasks involves controlling the output pins on the microcontroller to perform simple tasks, such as turning LEDs on and off. The way that the microcontroller is set up, a particular pin needs to be "declared" as input or output before it can be used. Since we noticed in our early tests that students often forgot to perform this particular step, BrickLayer automatically inserts statements for setting the behavior and the voltage of a pin into the particular areas of the source code once the required boxes have been dragged and dropped into the construction area. For example, suppose that the student wants to turn on the light at pin 13 in the loop function. He has to drag and drop an `On(Pin)` block and place it under a `loop()` block. From figure 2, we can see that '`PinMode(13, OUTPUT);`' and '`digitalWrite(13, LOW);`' are generated in `setup()` and `loop()` respectively after the `On(13)` box is dropped. This is because pin 13 needs to be configured as an output before one can set it into a LOW state, which corresponds to the LED being turned on.

Continuing with the above example, suppose that the student tries to drop another `On(13)` block after the first `On(13)` block. In this case, only one '`PinMode(13, OUTPUT);`' statement will be inserted into the setup section as it does not make sense to configure the same pin again once it is already set up. This not only reduces some logical errors, but also illustrates the concept of a pin or a variable being initialized or set to a particular state.

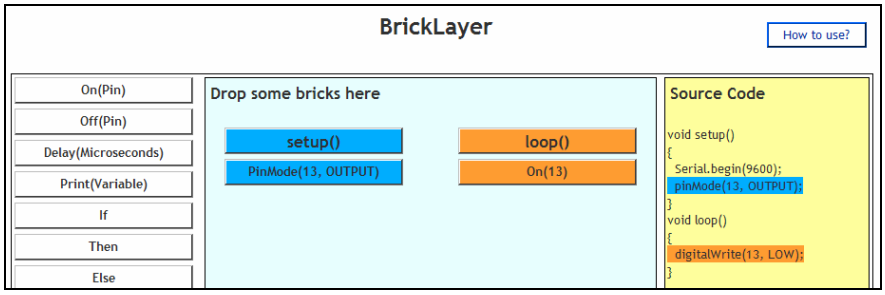

Figure 2: In the source code part of BrickLayer, dropping the `On(13)` block under the `loop()` block in the construction area automatically adds '`PinMode(13, OUTPUT)`' block under `setup()`. It also automatically generates the two corresponding code segments in the source code area.

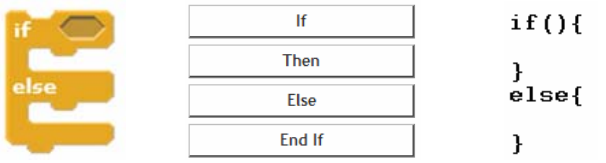

Figure 3: Fundamental constructs in Scratch (left), BrickLayer (middle) and C-programming (right).

Apart from the intelligent constructs, BrickLayer also includes fundamental constructs such as conditions, loops and variables. In many graphical programming environments such as Scratch and NXT-G [6], students are over-protected by the constraints of the programming environment. For example, the `if` statement icon in Scratch encompasses the entire conditional block, which guides and protects the user in placing statements that are under the scope of the conditional statement. This protects the user from his or her own mistakes, but it is possible that students will become over-reliant upon the interface and not remember the structures of the programmatic construct when they move to more advanced programming environments. In contrast, BrickLayer requires the student to drag and drop the separate '`If`', '`Then`' and '`End If`' components of a conditional statement individually into the construction area (see Figure 3). This gives them a more realistic view of programming concepts and constructs, and may make it easier for them to make the switch to conventional textual programming later on.

The final type of programming constructs is the user-defined function that we do not expect beginner students to be able to construct. For instance, in order to recognize time-series signal patterns, we constructed a function called `MatchPattern(Pin, Pattern)` that is based on the dynamic time warping algorithm. It takes in signal readings from specific sensors, such as the accelerometer, sees whether the current series of signals match with the specified pattern and returns a Boolean value to the user. This allows students to use more powerful constructs in their program, but frees them from having to work with overly complex syntax or algorithms.

More interestingly, as the program is constructed block by block, the source code is generated instantaneously next to the "icon-laying" layer. This gives students an idea of what the code actually looks like. In addition, this code can be cut-and-pasted into a normal text editor for further modification before compiling and downloading to the microcontroller. In our workshop, we noticed that many students would take this step on their own initiative and explore the syntax of the program code themselves.

4. RESULTS

We ran a series of summer workshops to gauge students' responses to various educational programming environments. Two of our workshops were based on the Scratch programming environment: one for elementary school students (Grades 5 and 6), and another for junior high students (Grades 7-9). To give a tangible dimension to the Scratch animations, our camp made use of the ScratchBoard [9] input devices.

Another one of our workshops was targeted at senior high school students, which focused on using the C programming language to control the same Arduino microcontroller that BrickLayer was designed for. To help with the learning process, we used the Arduino integrated development environment, which provides some support in programming syntax and logic. All four workshops ran for five full consecutive days.

The last of our workshops used the BrickLayer programming environment. We had 25 students enroll in the course: they were of ages from 11-16 (Grades 8-11).

Our experience over the three workshops was as follows: while the elementary school students found the Scratch workshop interesting and enjoyable, the junior high students found it less so. The C programming workshop was found to be challenging by the senior high students, who required a lot of help to finish their tasks.

The BrickLayer workshop confirmed our intuitions about using a text-enhanced graphical programming environment. At the beginning of the workshop, we surveyed students about their prior experiences in programming. We found that about 80% of the students had only minimal or even no programming experience. However, during the course, even these students were able to construct impressive and creative outcomes using BrickLayer within one day. A post-course survey revealed that nearly 70% of students felt that their interest in computational subjects had increased after the course.

In addition, we found that nearly 60% of the students were motivated to actually edit the textual source code by themselves, such as by adding in variables or constant values. We believe that this is a byproduct of the BrickLayer environment, as it exposes them to the textual source code as they construct their program, and allows them to gradually build up their confidence in programming. Since the textual statements are usually more easily recognized rather than recalled, we believe that this mode of learning is both effective and efficient.

Additionally, the majority of the students (over 80%) said that they would choose computational subjects in the future after the workshop that deployed the text-enhanced graphical programming environment. They enjoyed working as a programmer and felt that the course was stimulating and interesting.

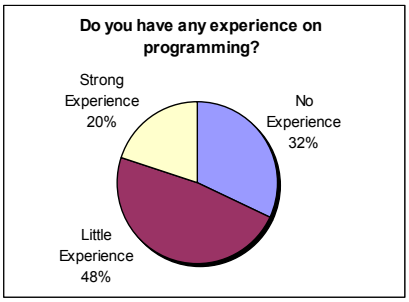

Figure 4: In the beginning of the workshop, we surveyed students about their previous experiences in programming. There were 25 respondents, 32% had no background in programming. 48% of them had little experience (i.e., exposure to but limited experience with one language). 20% had strong experience (i.e., at least one year's experience with more than one language).

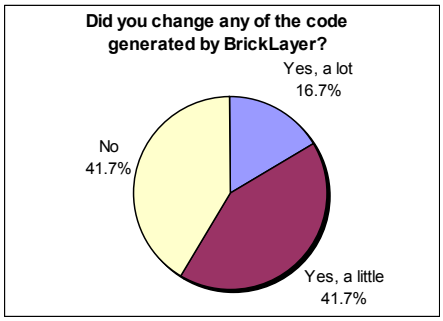

Figure 5: At the end of the workshop, we surveyed students to find out whether they modified the source code that was automatically generated by BrickLayer. Of 24 respondents, 16.7% modified a lot, 41.7% modified a little and 41.7% did not modify.

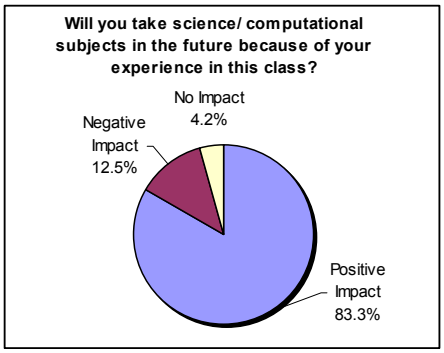

Figure 6: At the end of the workshop, we surveyed students about the impact of the workshop on their choice of future education. Among the 24 respondents, 83.3% agreed that the proposed environment brought a positive impact. 12.5% felt that this had a negative impact and 4.2% said that this had no impact.

It is not surprising that the students with more programming experience preferred to take the course that directly jumps to a real programming language. They felt that they had learned enough and were thus able to handle more challenging topics and tasks. They said that they came to the workshop because they want to learn the things that are new to them.

Given the evaluations and our own observations, we believe that this kind of text-enhanced graphical programming environment is effective and gives students a positive learning experience. This way of learning programming not only provides an environment in which fundamental computing concepts can be introduced and students can focus on logic issues without worrying about syntactic concerns, but are still exposed to the programming syntax, which can be picked up through absorption and observation. This mode of learning can also be applied to other programming domains, such as animations, games, interactive arts and robotics.

5. CONCLUSION

We have presented a text-enhanced graphical programming environment for teaching programming to junior high students. The age range and abilities of these students makes them less easily challenged and satisfied with traditional graphical programming tools such as Scratch, while they are not yet sufficiently advanced enough to tackle real textual programming such as C++ and Java. Our text-enhanced graphical programming environment is designed to fill in that gap. The proposed environment uses graphical-based programming to free students from syntactic issues, while exposing them to the programming syntax, which they pick up and learn through observation and experimentation.

To test our proposal, we organized four summer workshops targeting different age groups and using different programming environments. The feedback collected from these workshops validates the impact and effectiveness of our proposed environment. In the future, we will hold several more workshops and continue to assess this proposed environment for various conventional programming languages such as Java and C++.

6. ACKNOWLEDGMENTS

The presented work was supported by the Department of Computing and the eToy lab at the Hong Kong Polytechnic University.

7. REFERENCES

[1] Adams, J. Alice, Middle Schoolers & The Imaginary Worlds Camps. In SIGCSE '07: Proceedings of the 38th SIGCSE technical symposium on Computer science education, pages: 307 – 311, Covington, Kentucky, USA, 2007. ACM Press.

[2] Arduino. www.arduino.cc.

[3] Carnegie Mellon University. Alice v2.0. www.alice.org.

[4] Kelleher, C. & Pausch, R. Lowering the Barriers to Programming: A Taxonomy of Programming Environments and Language for Novice Programmers. ACM Computing Surveys, Vol. 37, No.2, pages 83-137. 2005.

[5] Malan, D. J. and Leitner, H. H. Scratch for budding computer scientists. In SIGCSE '07: Proceedings of the 38th SIGCSE technical symposium on Computer science education, pages 223-227, Covington, Kentucky, USA, 2007, NY, USA, 2007. ACM Press.

[6] NXT Tutorial-G. www.ortop.org/NXT_Tutorial/html/about.html.

[7] Schollmeyer, M. Computer programming in high school vs. college, 1996. In SIGCSE '96: Proceedings of the 27th SIGCSE technical symposium on Computer Science education', Philadelphia, Pennsylvania, USA, pages 378-382. ACM Press.

[8] Scratch. scratch.mit.edu.

[9] ScratchBoard. scratch.wik.is/Support/Sensor_Boards.

[10] Sivilotti, P.& Laugel, S. Scratching the Surface of Advanced Topics in Software Engineering: A Workshop Module for Middle School Students. In SIGCSE '08: Proceedings of the 39th SIGCSE technical symposium on Computer science education, pages 291-295, Portland, Oregon, USA, 2008. ACM Press.

[11] Yoder, M. & Black, B. Work in Progress: A Study of Graphical vs. Textual Programming for Teaching DSP. In ASEE/IEEE Frontiers in Education Conference, 36th Annual, page 17-18, San Diego, CA, 2006.