Proceedings

2015 IEEE Blocks and Beyond Workshop (Blocks and Beyond)

B&B 2015

October 22, 2015
Atlanta, Georgia, USA

Edited by:
Franklyn Turbak
David Bau
Jeff Gray
Caitlin Kelleher
Josh Sheldon

ISBN 978-1-4673-8366-0
IEEE Catalog Number CFP15E28-USB

2015 IEEE Blocks and Beyond Workshop (Blocks and Beyond)

IEEE Catalog Number CFP15E28-USB
ISBN 978-1-4673-8366-0

Table of Contents

Blocks, Text, and Structured Editing

New Domains for Blocks Languages

New Features for Blocks Environments

Pedagogy of Blocks Languages

Foreword

Blocks and Beyond, 2015

Welcome to the 2015 IEEE Blocks and Beyond Workshop (B&B)!

Recently, there has been an explosion of interest in blocks programming languages and related visual programming environments, such as Scratch, Blockly, App Inventor, Snap!, Pencil Code, Alice/Looking Glass, Greenfoot, TouchDevelop, and AgentSheets/AgentCubes. Code.org's Hour of Code and blocks-based curricular activities have played a huge role in increasing the popularity of blocks languages. These environments have introduced programming and computational thinking to tens of millions, reaching people of all ages and backgrounds.

Despite their popularity, there has been remarkably little research on the usability, effectiveness, or generalizability of affordances from these environments. This workshop brings together educators, researchers, and developers with experience in blocks languages with the goal of beginning to distill testable hypotheses from the existing folk knowledge of blocks-based programming environments and to identify research questions and partnerships that can legitimize, or discount, pieces of this knowledge.

We are delighted that this workshop is being held in conjunction with VL/HCC 2015 in Atlanta. This raises the profile of blocks languages and encourages cross-fertilization between blocks language researchers and the broader VL/HCC community.

When we first started planning this workshop, we imagined an intimate gathering of about 15 to 20 people. So, we were astonished by the overwhelming response to our workshop. We received 52 submissions, underscoring a large untapped interest in blocks languages.

Submissions were invited in two categories: position statements (1 to 3 pages describing an idea, research question, or work related to the design, teaching, or study of blocks programming languages) or papers (up to 6 pages describing previously unpublished work in the blocks language area). Of the 19 papers submitted, 14 (74%) were accepted. Of the 32 position statements submitted, 22 (69%) were accepted. (These statistics do not include a paper that was withdrawn when the authors realized they could not attend the workshop.) We thank our Program Committee members for reviewing many more submissions than we expected, and for doing so on a very short timeline during prime summer vacation weeks.

Between submission authors, members of the program and organizing committees, and other interested parties, we are expecting about 60 attendees at B&B. This large number raises many challenges for the workshop schedule. From the beginning, we wanted the workshop to emphasize discussion, and not just be a mini-conference in which papers are presented. But having discussions with 60 participants is not very fruitful. So we decided on a model in which we would would have 6 presentation/discussion sessions scheduled in two tracks over three 1.5 hour time slots. Each such session will begin with about 30 minutes of presentations and will be followed by an hour of discussion. Having parallel tracks in a workshop is highly unusual, but we saw no other way to handle the large number of accepted submissions and preserve the discussion-oriented nature of the workshop. These sessions will be preceded by a 1.5 hour demo/poster session in which any attendee can present a poster or give a demo on a blocks language topic. At the end of the day, there will

be a wrap-up session summarizing the discussions of the 6 presentation/discussion sessions and planning for the future.

To further take advantage of having so many people interested in blocks languages together in one place, we have allocated the day after the workshop to informal meetings and discussions for interested attendees. Nearly two-thirds of the expected attendees have indicated that they want to participate in these post-workshop discussions.

We owe special thanks to the VL/HCC 2015 organizers for all their hard work in helping to make B&B a success. Scott Fleming gave us much help and encouragement in the planning and organization of this workshop. Eileen Kraemer did an extraordinary job finding additional space and hotel rooms when the number of expected attendees tripled. Jane Li provided valuable advice and flexibility in the preparation of the proceedings.

Most of all, we would like to thank everyone who submitted to and/or will be attending the B&B workshop in Atlanta. We are all looking forward to the conversations that will take place and the ideas that will emerge when so many blocks language enthusiasts are together in one place.

Sincerely,

The Blocks and Beyond Organizing Committee:
Franklyn Turbak (chair)
David Bau
Jeff Gray
Caitlin Kelleher
Josh Sheldon

Workshop Committees

Organizing Committee

Franklyn Turbak (chair), *Wellesley College, USA*
David Bau, *MIT and Google, USA*
Jeff Gray, *Unversity of Alabama, USA*
Caitlin Kelleher, *Washington University in St. Louis, USA*
Josh Sheldon, *MIT, USA*

Program Committee

Neil Brown, *University of Kent, UK*
Dave Culyba, *Carnegie Mellon University, USA*
Sayamindu Dasgupta, *MIT, USA*
Deborah Fields, *Utah State University, USA*
Neil Fraser, *Google, USA*
Mark Friedman, *Google, USA*
Dan Garcia, *University of California, Berkeley, USA*
Benjamin Mako Hill, *University of Washington, USA*
Fred Martin, *University of Massachusetts Lowell, USA*
Paul Medlock-Walton, *MIT, USA*
Yoshiaki Matsuzawa, *Aoyama Gakuin University, Japan*
Amon Millner, *Olin College, USA*
Ralph Morelli, *Trinity College, USA*
Brook Osborne, *Code.org, USA*
Jonathan Protzenko, *Microsoft Research, USA*
Alexander Repenning, *University of Colorado, Boulder, USA*
Ben Shapiro, *University of Colorado, Boulder, USA*
Wolfgang Slany, *Graz University of Technology, Austria*
Daniel Wendel, *MIT, USA*

Position Paper: Assessing Knowledge in Blocks-Based and Text-Based Programming Languages

Briana B. Morrison
School of Interactive Computing
Georgia Institute of Technology
Atlanta, GA USA
bmorrison@gatech.edu

Assessing student knowledge of programming concepts is a long studied, but many would say unsolved, problem. From the initial Empirical Studies of Programmers series *[1]* to McCracken's study *[2]* to Elliott-Tew's validated assessment of fundamental CS knowledge *[3]* many have attempted to precisely measure the knowledge and skills of students learning to program. All of the previous efforts have been centered on text based languages. Now we have another element to add to the mix – block-based programming languages. Can any of the existing assessment instruments work for both text and block based programming languages? Do we believe they will measure the same knowledge? Can existing assessments be modified and tailored for block-based languages? Are the two approaches equivalent, and would we even want to assess the same knowledge outcomes for both? These are but a few of the questions I will address in this position paper.

Keywords—assessment, outcomes, blocks, text

I. Background

Assessing programming knowledge of students is an often hotly debated topic among computing education researchers. In the original studies of programmer knowledge [1] both novices and expert programmers were studied to determine both their knowledge and skills and how novices could be encouraged to think and behave more like the expert programmers. However, no common assessment instrument was ever created to measure either knowledge or skills. In 2001, McCracken lead an ITiCSE working group to compare student knowledge and programming skills between institutions and across countries. The arguably simple task of programming a RPN calculator was attempted by over 200 students with much lower than expected results. The less than stellar results began a re-emphasis on assessment strategies – how were all these students able to pass a programming class and yet unable to complete the task?

Many science, technology, engineering and mathematics (STEM) disciplines have standard validated assessment tools that allow educators and researchers to accurately measure student learning and evaluate curricular innovations (e.g., [4], [5], [6]). However, computer science does not have a similar set of validated assessment tools, and practitioners and researchers must often devise their own instruments when they want to investigate student learning. The closest possible instrument may be the AP Computer Science A test, which is not readily available without cost and answers to all the free response questions are easily available on the internet.

In 2010, Elliott-Tew presented the FCS1, a language independent validated assessment instrument for measuring foundational CS1 knowledge in a language-independent manner. While much lauded at the time, the assessment has only been used one other time [7] and remains generally inaccessible today.

All of these assessments are based on text based programming languages. With the somewhat recent push toward block-based languages such as Scratch, AppInventor, and SNAP! for young novice programmers, there have been a few attempts at creating assessments of programming knowledge or computational thinking skills (e.g., [8], [9], [10]). In addition, an assessment for AP CS Principles, where instructors often uses block-based languages for the programming portion of the curriculum, is under development.

Yet, the assessment paths have never met. Is it possible to adapt the FCS1 for use with block-based languages? The test is implemented in pseudo-code which has been validated against student performance with procedural languages (Java, Python, MATLAB). Is it possible to adapt an existing blocks-based assessment for a text-based language?

II. Comparing Assessments

There are several issues to look at when examining the current assessments available for both text-based and block-based programming languages which include whether we should measure the same knowledge, the format of the test questions, and issues of scalability.

A. Knowledge Outcomes

There is a general consensus of the important concepts that should be covered in a CS1 course – sequence, conditionals, loops, data, and operators ([3], [9]. However, most existing assessments include these topics as well as others. Before creating an assessment which can be used with both block and text based languages, we must decide upon the knowledge outcomes to be tested. For example, [9] also includes testing knowledge of parallelism and events. Most educators would consider this inappropriate for a text-based language implementation of CS1. In most text-based language assessments, some form of procedure / parameter passing is

978-1-4673-8367-7/15/$31.00 ©2015 IEEE

considered important, yet this does not often occur in block-based language assessments.

As researchers looking at the equivalency of both block and text based programming languages, we must decide what outcomes the assessments aim to measure and whether or not they should be identical or different for the separate environments. Perhaps there should be a common subset of topics for both, with separate concept areas added for each unique environment.

B. Format

While the question "What format should the assessments take?" is seemingly a simple one, it is anything but when it comes to block-based languages. In general most existing validated assessment instruments consist of multiple choice questions with perhaps some open-ended or free response questions. Currently none of the block-based language assessments use multiple choice questions. The development effort for the AP CS Principles assessment is currently attempting to convert its open-ended questions into multiple choice questions which can then be validated.

In text based programming assessments, common question forms include tracing (What is the output of this code segment?) or fill in the blank (What line(s) of code belong in the indicated space in order for the code to do X?). Neither of these forms of questions is easily adaptable for blocks-based languages. In [10], they used open-ended questions like "After this script executes which way will the sprite be facing?" and "What instructions must be added to the script to allow X to happen?" While these appear to be similar question types, both the creation and the grading process for them are very different. In a text-based language the questions and answers can be easily typed into a document. Not so with blocks-based languages – the image of the script and blocks must be captured and inserted into the document; this is usually a much more time consuming effort. In addition, each block-based language has a different set of blocks. Is there a general consensus of the subset of blocks which could be used for test questions? Is there an equivalent pseudo-blocks for these environments?

One possible question format which may work for both environments is a Parsons problem [11]. In a Parsons problem, lines of code (either text-based or blocks) are mixed-up and students are asked to put them into the correct order to solve the problem. Unfortunately, no validation of Parsons problems has yet occurred.

C. Scalability

The majority of all existing block-based language assessments involve having the student create new scripts to evaluate their knowledge. While this is completely acceptable for a teacher with a small or average sized-class, it is unfeasible for someone with a large class or researchers attempting to measure knowledge across thousands of students. While much headway has been made in the automatic grading of text-based program solutions, little is available for block-based program solutions. There is a pressing need for scalable, repeatable, validated assessments for block-based languages. Is it possible to create questions which measure similar knowledge and skills of the students without requiring them to create scripts / programs?

It is true that multiple choice questions have limitations and cannot accurately assess all aspects of a student's knowledge or their skills. We need the ability to assess a student's ability to design and correctly implement an algorithm and this is not readily testable using only multiple choice questions. Yet we cannot also score hundreds (or thousands) of independently constructed programs without resources similar to ETS (who pays for hundreds of program graders every summer to score the AP CS A free response questions). We need an intermediate solution which is easily automatically assessed and can still distinguish the knowledge level of the students.

III. Differences

There are proven educational learning differences between text based and blocks-based programming languages. In 2012, Margulieux et al. [12] replicated existing studies in other disciplines using subgoal labels with a block-based language (AppInventor). The participants were able to retain more information and perform the task more accurately when receiving subgoals than those who did not receive subgoal labels. However when the experiment was replicated with a text-based language [13] results were not as clear. There is an underlying question of why subgoals work as expected in a blocks-based environment, but not a text-based environment. Does this difference carry-over into assessments? Are there types of questions suitable for one and not the other? We must carefully examine all known learning differences between the environments while designing assessments.

IV. Future Research Questions

When it comes to assessment of student programming knowledge in text based and block based languages, there are more questions than answers at this point in time. While text-based assessments exist, none are ideal and none have been adapted or "translated" to block-based languages. None of the existing block-based assessments have been validated or are scalable. Most involve the creation of new scripts to assess student knowledge which is generally labor intensive to grade with many different possible correct solutions. When looking at block programming environments, an important topic to be considered is assessment of student knowledge. If we hope for educators to adopt and use block-based environments for teaching CS1 we must also provide them with assessment instruments to measure student knowledge and skills.

References

[1] E. Soloway and K. Ehrlich, "Empirical studies of programming knowledge," *Softw. Eng. IEEE Trans.*, no. 5, pp. 595–609, 1984.

[2] M. McCracken, V. Almstrum, D. Diaz, M. Guzdial, D. Hagan, Y. B.-D. Kolikant, C. Laxer, L. Thomas, I. Utting, and T. Wilusz, "A multi-national, multi-institutional study of assessment of programming skills of first-year CS students," in *Working group reports from ITiCSE on Innovation and technology in computer science education*, Canterbury, UK, 2001, pp. 125–180.

[3] A. E. Tew, "Assessing fundamental introductory computing concept knowledge in a language independent manner," 2010.

[4] D. Hestenes, M. Wells, G. Swackhamer, and others, "Force concept inventory," *Phys. Teach.*, vol. 30, no. 3, pp. 141–158, 1992.
[5] C. D'Avanzo, "Biology concept inventories: overview, status, and next steps," *BioScience*, vol. 58, no. 11, pp. 1079–1085, 2008.
[6] J. C. Libarkin and S. W. Anderson, "Assessment of learning in entry-level geoscience courses: Results from the Geoscience Concept Inventory," *J. Geosci. Educ.*, vol. 53, no. 4, p. 394, 2005.
[7] I. Utting, A. E. Tew, M. McCracken, L. Thomas, D. Bouvier, R. Frye, J. Paterson, M. Caspersen, Y. B.-D. Kolikant, J. Sorva, and T. Wilusz, "A Fresh Look at Novice Programmers' Performance and Their Teachers' Expectations," in *Proceedings of the ITiCSE Working Group Reports Conference on Innovation and Technology in Computer Science Education-working Group Reports*, New York, NY, USA, 2013, pp. 15–32.
[8] S. Grover, S. Cooper, and R. Pea, "Assessing Computational Learning in K-12," in *Proceedings of the 2014 Conference on Innovation & Technology in Computer Science Education*, New York, NY, USA, 2014, pp. 57–62.
[9] K. Brennan and M. Resnick, "New frameworks for studying and assessing the development of computational thinking," in *Proceedings of the 2012 annual meeting of the American Educational Research Association, Vancouver, Canada*, 2012.
[10] O. Meerbaum-Salant, M. Armoni, and M. (Moti) Ben-Ari, "Learning computer science concepts with Scratch," *Comput. Sci. Educ.*, vol. 23, no. 3, pp. 239–264, Sep. 2013.
[11] D. Parsons and P. Haden, "Parson's Programming Puzzles: A Fun and Effective Learning Tool for First Programming Courses," in *Proceedings of the 8th Australasian Conference on Computing Education - Volume 52*, Darlinghurst, Australia, Australia, 2006, pp. 157–163.
[12] L. E. Margulieux, M. Guzdial, and R. Catrambone, "Subgoal-labeled instructional material improves performance and transfer in learning to develop mobile applications," in *Proceedings of the ninth annual international conference on International computing education research*, 2012, pp. 71–78.
[13] Morrison, Briana B., Margulieux, Lauren E., and Guzdial, Mark, "Subgoals, Context, and Worked Examples in Learning Computing Problem Solving," in *ICER 2015*, 2015.

The Challenges of Studying Blocks-based Programming Environments

David Weintrop
Learning Sciences
Northwestern University
Evanston, IL
dweintrop@u.northwestern.edu

Uri Wilensky
Learning Sciences & Computer Science
Northwestern University
Evanston, IL
uri@northwestern.edu

***Abstract*—In this paper we discuss some of the central challenges to rigorously studying blocks-based programming. We categorize these challenges into four groups: design, context, evaluation, and trajectory. For each category, we breakdown specific issues we confront as researchers in trying to isolate the blocks-based modality and outline our own strategies for addressing them. Throughout, we draw on our own experience as designers, educators, and researchers, along with past and current work, to provide guidance on ways to address these challenges and share insights our approach has yielded.**

Keywords—blocks-based programming; computer science education; design; methods

I. Introduction

Blocks-based programming is increasingly becoming the way that learners are being introduced to programming. Led by the popularity of tools like Scratch, Alice, and the suite of environments offered as part of Code.org's Hour of Code, blocks-based programming has become a central approach used in the design of introductory programming environments. Following this trend, a growing number of curricula are utilizing blocks-based programming tools, including the CS Principles project, the Exploring Computer Science program, and the materials being developed and disseminated by Code.org. Given the growing prominence of this approach, it is critical that we fully understand the consequences of the decision to rely on this modality in formal educational settings. Despite its growing prevalence, many open questions remain surrounding the effects of blocks-based programming on learning. This is due, in part, to the challenges of studying blocks-based programming independent of other factors that often accompany the activity of learning to program. In this paper, we discuss various challenges we face in trying to study blocks-based programming, grouping these challenges into four categories: Design, Context, Evaluation, and Trajectory. Drawing on our own experience conducting classroom-based research on blocks-based programming, we discuss specific challenges to studying programming modality and then present our own strategies to overcoming some of these challenges.

II. Challenges of Studying Blocks-based Programming

We break the challenges associated with studying blocks-based programming down into four general categories. The first category, Design, focuses on features of blocks-based programming tools and discusses challenges inherent to the representation, and various ways the blocks-based modality can be conflated with other features of introductory environments. The second category is Evaluation, which looks at challenges associated with evaluating learning in blocks-based environments. The next category, Context, outlines challenges associated with different factors external to the programming environment that affect learning and engagement. Finally, we explore challenges faced in studying learner trajectories that blocks-based programming tools support, specifically focusing on the question of studying transfer from blocks-based to more conventional text-based programming languages.

A. Design

The first set of challenges we discuss are those related to the goal of trying to study the blocks-based modality and features of its visual display and interaction dynamics. We use the term modality here to specifically refer to the visual, representational characteristics of blocks-based programming, which is distinct from the language (i.e. set of primitives provided), and the larger environment in which the modality is situated (i.e. the other elements of the interface like that palette that holds sets of blocks and the stage, where the results of programs are displayed). In highlighting the distinction between modality and language, we encounter the first challenge: the potential of conflating the two. While it is not possible to completely disentangle these two characteristics of blocks-based programming, it is important to recognize where language and modality are bound together, and where they can be separated, as well as the consequences of doing so.

An example of the challenge in conflating language and modality can be seen in Lewis' [1] comparison of Scratch and Logo. As part of their post test, students were asked to answer a question on conditional expressions, Figure 1 shows the two ways an if statement was presented in the question.

```
if greater? Mousex 0
  [setpencolor 2]
```

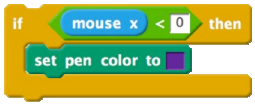


Fig. 1. A portion of a conditional logic question from [1].

So, when Lewis found that students did better on the Scratch version of the question, how do we explain the result? Is it because students find the infix `<` easier to interpret than the prefix `greater?` operator? Or possibly because the colors, shapes, and nesting of the blocks make that form easier to parse? These two potential explanations highlight the conflation of language and modality, in one explanation the difference is a feature of the language (`<` vs. `greater?`) while the other attributes the difference to the modality (text vs. blocks display). In our own work, we have encountered this language/modality conflation in a number of places, the most clear-cut example being in our analysis of student understanding of definite loops. In questions that had students compare blocks-based `repeat` commands with text-based `for` loops, students did significantly better on the blocks-based questions. However, when we compared blocks-based `for` loops to text-based `for` loops, we saw that performance gap disappear [2]. This suggests that the gains we found in the `repeat` vs. `for` questions were more likely due to language than modality, a finding that is consistent with Lewis' [1].

One way to address this issue is to use environments in which the blocks-based and text-based modalities share language features. One nice example of this is Pencil Code [3], where the blocks-based interface is essentially a visual overlay rendered on top of the text-based form of a program. Thus, the actual text of the program is identical across the two modalities. Figure 2 shows Pencil Code text and blocks versions of the question from Figure 1 using JavaScript as the base language.

```
if (mouse.x > 0) {
  pen(red);
}
```

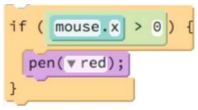


Fig. 2. Pencil Code's text and block-based rendering of the code in Figure 1.

Part of the reason it is not possible to completely disentangle modality from language is due to the blocks-based visualization's ability to clearly depict the constituent elements of a single command. Having a clearly delimited boundary to a command makes it visually apparent how to parse a program and thus makes it possible for a single command to be made up of compound phrases, like: `say Hello! for 2 seconds`. This enables a programming language to have a more readable, natural language feel; a feature that learners identify as contributing to the easy-of-use of blocks-based tools [4].

A second conflation the emerges in studying blocks-based programming tools is linking the blocks-based modality with the other features of the programming environment that surround and give meaning to the programming activity. For example, Scratch includes not just the blocks-based programming interface, but also the Logo-like ability to introduce sprites and have them visually enact programs on the screen. By embedding programming in an environment that makes it easy to create games and interactive stories that can easily be personalized and shared, Scratch has been very successful at getting learners engaged and excited about programming [5]. As we are interested in understanding the affordances of blocks-based programming, it is important to try and understand the contribution of the blocks-based modality to creating this sucessful and engaging programming environment. Our solution to this issue has been to build on the successes of rich, engaging programming contexts by bringing text-based programming into these interactive, graphical programming contexts. In Snappier!, a modification of Snap! [6], we made it possible for learners to create custom blocks that are controlled by short text-based programs. In this way, text-based programs could interact with all the elements of the blocks-based world and be incorporated into larger blocks-based programs [7].

B. Evaluation

A second type of challenge we face when trying to understand the strengths and drawbacks of blocks-based programming stems from a dearth of good, validated assessments that will give us comparative insights between blocks-based and text-based modalities. While there are a number of assessments that use blocks-based programming (e.g., [8], [9]), and others built around programming in blocks-based environments (e.g., [10]), there are few assessments well suited to the research pursuit of understanding the affordances and drawbacks of blocks-based programming relative to isomorphic text-based alternatives. To fill this gap, we created the Commutative Assessment [2]. Each multiple-choice question on the assessment includes a short program that can be displayed in either a blocks-based or text-based form. The set of potential answers for each question includes the correct answer along with choices informed by prior research on novice programming misconceptions. We used the Commutative Assessment in a 10 week study that had students take the assessment three times, alternating the modality shown for each question across the different administrations, thus giving us with-in student performance, as well as time-series data, to try and link conceptual understanding with modality. For a longer description of the assessment, as well as findings from its use, see [2].

The previously discussed assessments focus more on program comprehension than program generation. To date, most of the work looking at programming authoring has relied on qualitative analysis of program authoring, which is very useful for understanding how learners rely on features of the modality (e.g. [11], [12]). Another methodology that is growing in popularity that can be useful for assessing blocks-based programming tools is the use of educational data mining and learning analytics techniques. The collection and analysis of log data from students as they develop, run, and revise their programs can provide insights into the practice of program composition. While there is some innovative work happening in this space (e.g., [13], [14]), we have yet to see this approach applied as a methodology for evaluating programming modality. This is an analysis we intend on doing in the future and think it could be very insightful for understanding the effect that modality has on practices that can complement the findings from the quantitative content evaluations.

C. Context

There are many differences between formal classroom settings and the informal contexts in which much of the blocks-based learning research has occurred. The differences

include the layout of the physical space, the peer culture, the autonomy learner have, the time dedicated to each activity, and the presence (or absence) of experts in the form of teachers, councilors, or other experienced programmers. Given all these differences, it is not safe to assume that what is true in an informal, open-ended context would necessarily hold in formal classrooms. Trying to comparatively study the role of modality in supporting (or hindering) a student's emerging programming understanding needs to account for these larger contextual differences. This challenge is best addressed through study design. In a recent study, we used a quasi-experimental design to hold constant many of these external factors. We followed three introductory classrooms in a single school; all three classes worked through the same curriculum and were taught by the same teacher in the same classroom. Each class used a different version of the same programming tool, with the only difference between them being the programming modality, either blocks-based, text-based, or a hybrid blocks/text tool. This design allowed us to focus specifically on questions of representation and modality that is at the heart of our research agenda.

D. Trajectory

A major open question facing the computer science education community is what comes after blocks-based programming. As more curricula incorporate blocks-based tools, especially at the high school and undergraduate levels, educators face the question of how best to transition learners from the blocks-based modality to more conventional text-based programming languages. Part of the challenge of answering this question convincingly is its longitudinal nature, as this transition often happens between courses. In such cases, claims can be made about successful transitions, but it is hard to attribute the successes to specific features of the blocks-based modality due to the amount of time that has passed, leaving the researchers to make only general claims (e.g., [15]). Additionally, such studies lack a control condition, making it difficult to attribute improvements to the modality (or features of the modality), and instead have to take the environment as a whole, making such studies susceptible to the issues of conflation previously discussed. We are studying the blocks-to-text transition by having shift modalities during a single course, holding constant classroom, peers, and teachers. We follow students for the first five weeks of an introductory programming class while they use a blocks-based environment, then continue to observe, interview, and collect program log-data on those students for an additional ten weeks as they transition to Java. In following students across the transition and having students initially work in differing modalities, we have designed a study that will be able to give us data to more closely attribute successes and failures of the transition to features of the introductory programming tools.

III. Conclusion

Programming is now recognized as a core 21st century skill. School districts, responding to this trend, are rapidly introducing new computer science courses and integrating computational thinking into traditionally non-computational coursework to accommodate this shifting computational landscape. Given this trend, it is critical that we as computer science education researchers and designers understand the tools and curricula that we are advocating for schools to use, as the environments and courses that are adopted today will lay the foundation upon which computer science, and computational thinking more broadly, will be taught for years to come. In this paper we have laid out what we see as the central challenges to studying the blocks-based programming approach. Our hope with this paper is to raise awareness of the challenges we face and demonstrate possible ways to address these research challenges and pave the way for rigorous, careful research that will yield insights into the tools students are using today and guide the way for the tools that students will use tomorrow.

References

[1] C. M. Lewis, "How programming environment shapes perception, learning and goals: Logo vs. Scratch," in *Proc. of the 41st ACM Technical Symposium on CS Ed*, New York, NY, 2010, pp. 346–350.

[2] D. Weintrop and U. Wilensky, "Using Commutative Assessments to Compare Conceptual Understanding in Blocks-based and Text-based Programs," in *Proc. of the 12th Annual Int. Conf. on International Computing Education Research*, Omaha, NE, 2015.

[3] D. Bau, D. A. Bau, M. Dawson, and C. S. Pickens, "Pencil Code: Block Code for a Text World," in *Proc. of the 14th Int. Conf. on Interaction Design and Children*, New York, NY, USA, 2015, pp. 445–448.

[4] D. Weintrop and U. Wilensky, "To Block or Not to Block, That is the Question: Students' Perceptions of Blocks-based Programming," in *Proc. of the 14th Int. Conf. on Interaction Design and Children*, New York, NY, USA, 2015, pp. 199–208.

[5] M. Resnick et al., "Scratch: Programming for all," *Commun. ACM*, vol. 52, no. 11, p. 60, Nov. 2009.

[6] B. Harvey and J. Mönig, "Bringing 'no ceiling' to Scratch: Can one language serve kids and computer scientists?," in *Proc. of Constructionism 2010*, Paris, France, 2010, pp. 1–10.

[7] D. Weintrop, U. Wilensky, J. Roscoe, and D. Law, "Teaching Text-based Programming in a Blocks-based World," in *Proc. of the 46th ACM Technical Symposium on CS Ed*, New York, NY, USA, 2015, p. 678.

[8] C. M. Lewis, "Is pair programming more effective than other forms of collaboration for young students?," *Comput. Sci. Educ.*, vol. 21, no. 2, pp. 105–134, Jun. 2011.

[9] S. Grover, S. Cooper, and R. Pea, "Assessing computational learning in K-12," 2014, pp. 57–62.

[10] L. Werner, J. Denner, S. Campe, and D. C. Kawamoto, "The fairy performance assessment: measuring computational thinking in middle school," in *Proc. of the 43rd ACM technical symposium on CS Ed*, 2012, pp. 215–220.

[11] O. Meerbaum-Salant, M. Armoni, and M. Ben-Ari, "Habits of programming in Scratch," in *Proc. of the 16th Annual Joint Conference on ITiCSE,* Darmstadt, Germany, 2011, pp. 168–172.

[12] D. Weintrop and U. Wilensky, "Supporting computational expression: How novices use programming primitives in achieving a computational goal," presented at AERA 2013, San Francisco, CA, USA, 2013.

[13] M. Berland, T. Martin, T. Benton, C. Petrick Smith, and D. Davis, "Using Learning Analytics to Understand the Learning Pathways of Novice Programmers," *J. Learn. Sci.*, vol. 22, no. 4, pp. 564–599, 2013.

[14] P. Blikstein, M. Worsley, C. Piech, M. Sahami, S. Cooper, and D. Koller, "Programming Pluralism: Using Learning Analytics to Detect Patterns in the Learning of Computer Programming," *J. Learn. Sci.*, vol. 23, no. 4, pp. 561–599, 2014.

[15] M. Armoni, O. Meerbaum-Salant, and M. Ben-Ari, "From Scratch to 'Real' Programming," *ACM TOCE*, vol. 14, no. 4, pp. 25:1–15, 2015.

The Impact of Distractors in Programming Completion Puzzles on Novice Programmers Position Statement

Kyle J. Harms
Department of Computer Science & Engineering
Washington University in St. Louis
St. Louis, Missouri, United States
kyle.harms@wustl.edu

I. Position Statement

Our previous work has demonstrated that programming completion puzzles enable novice programmers to acquire new programming skills [1]. As shown in Fig. 1, programming completion puzzles ask users to reassemble a block-based program's statements into the correct order. Users use the available blocks in the puzzle statement bin (Fig. 1-A) and place them into the correct order in the puzzle workspace (Fig. 1-B). In our previous work we only included blocks that were part of the actual puzzle's solution [1]. However other puzzle-like programming systems often include distractor statements as part of the user's experience [2]. In the context of programming puzzles, distractors are extra blocks or statements that are not part of a puzzle's solution. We wondered what impact distractor statements might have in programming completion puzzles on novice programmers? Do distractors also help facilitate learning programming skills when used in programming completion puzzles?

Our programming completion puzzles are heavily inspired by Cognitive Load Theory's completion problems [3]. Completion problems are partially completed worked examples where a user completes the remainder of the example. By limiting extraneous work, completion problems focus a learner's mental resources towards processing new material. Our completion puzzles limit the blocks users need to use to complete the problem. This is intended to focus and direct their mental resources towards learning new programming concepts; a learner does not need to expend mental resources navigating the interface to the find the correct block. However, we were unsure how including distractors into completion puzzles might affect learning outcomes. Since the intent of these problems is to carefully manage learning resources, it is possible that distractors may overwhelm learners' limited working memory resources, thereby limiting their ability to learn new programming concepts.

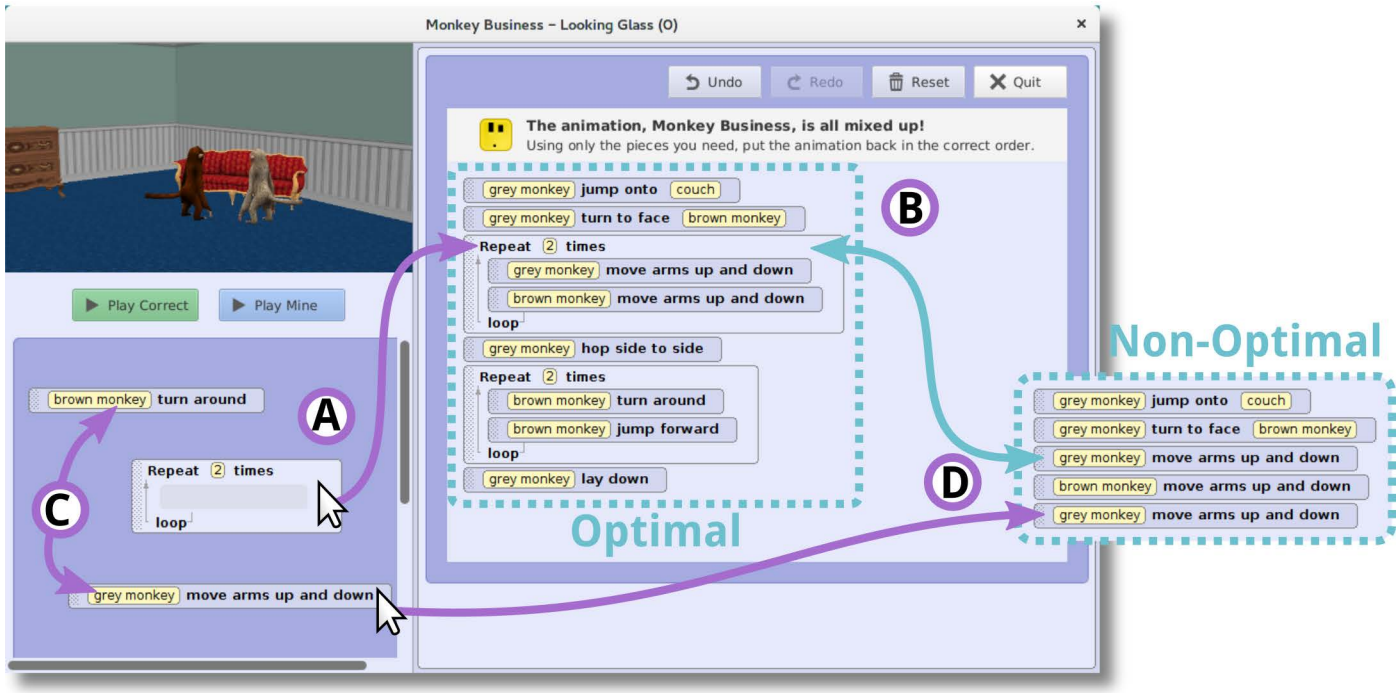


Fig. 1. A typical programming completion puzzle with distractors. Users complete these puzzles by dragging blocks from the puzzle statement bin (A) and dropping the blocks into the puzzle workspace (B). This puzzle includes several distractor statements (C) and an example of their use in a non-optimal solution (D).

978-1-4673-8367-7/15/$31.00 ©2015 IEEE

Recently, we have been conducting a formative evaluation into the impact of distractors within programming completion puzzles. Fig. 1-C shows an example of several distractor statements. Distractor statements are not part of the solution to the puzzle, instead they are extra unnecessary statements that user's must rule out in order to solve a puzzle.

From our early observations in this evaluation, we have noticed that distractors seem to make the puzzles more challenging for users. We have observed a greater frequency of comments related to the distractor puzzles being more challenging and more fun. Further, users typically need more time to complete a puzzle with distractors than one without. If distractors do increase the difficulty of the puzzle and the motivation to remain engaged with the puzzles, then they might provide a way to encourage users to continue practicing non-mastered programming concepts.

So far in our formative evaluation we have noticed two ways to use distractors to increase the difficulty of the puzzles: 1) use distractors to encourage users to follow a sub-optimal path, and 2) mix up the type of distractors between puzzles. Both of these methods appear to increase the challenge of the puzzle while also encouraging users to pay closer attention to the material.

We use sub-optimal path distractors to encourage users to begin to construct an alternative and incorrect solution to a puzzle. When constructing an alternative solution, the user will fail to fully complete the alternative solution because the puzzle is missing several critical pieces necessary for that solution. Once a user realizes that she cannot complete the puzzle using the alternative solution, she is forced to reevaluate her solution strategy to come up with the proper and correct solution.

See Fig. 1-D for an example of a distractor which encourages users to take a sub-optimal path. In this puzzle, we noticed that novices frequently want to repeat statements by duplicating the statements instead of using a loop. When we provide the distractor, "grey monkey move arms up and down," they usually start down the sub-optimal *duplicate statements* path. However, they eventually realize they cannot complete this puzzle by duplicating the statements because they lack the final statement necessary for this solution: "brown monkey move arms up and down." Rather, they have to reevaluate their solution to realize that they must use a loop instead. It appears that leading users down a sub-optimal path has the effect of making the puzzle more challenging while possibly encouraging more practice with programming concepts.

We have also observed that providing an unpredictable experience by mixing up the type of distractors between puzzles keeps users alert. For example, a puzzle curriculum might begin with a puzzle that includes sub-optimal distractors and then follow up with a puzzle that has no distractors. While completing the second puzzle, we have observed that participants carefully consider whether or not each statement is necessary for a solution. With a predictable experience, the users may begin to use their expectations about distractors to simplify the problem solving process. An unpredictable completion problem experience may encourage users to pay closer attention to the elements needed to solve each puzzle.

Our early work suggests that distractors may provide an additional approach to encourage novices to learn and practice programming skills. We also think that the additional challenge introduced by the distractors may increase the longevity of programming puzzles as a tool novices may use to develop their programming skills.

REFERENCES

[1] K. J. Harms, N. Rowlett, and C. Kelleher, "Enabling Independent Learning of Programming Concepts through Programming Completion Puzzles," in *2015 IEEE Symposium on Visual Languages and Human-Centric Computing (VL/HCC)*, 2015.

[2] M. J. Lee, A. J. Ko, and I. Kwan, "In-game Assessments Increase Novice Programmers' Engagement and Level Completion Speed," in *Proceedings of the Ninth Annual International ACM Conference on International Computing Education Research*, New York, NY, USA, 2013, pp. 153–160.

[3] J. J. G. Van Merriënboer, "Strategies for Programming Instruction in High School: Program Completion vs. Program Generation," *J. Educ. Comput. Res.*, vol. 6, no. 3, pp. 265–285, Jan. 1990.

Java as a second language:

Thoughts on a linguistically informed transition to typing languages

Eileen King
IB Design
Lakes International Language Academy
Forest Lake, MN
eking@lakesinternational.org

***Abstract*—As use of blocks-based languages as an introduction to programming has become popular, a need has become increasingly apparent for research in how to facilitate the transition from these environments to industry-standard languages. Mediated transfer strategies have been used that are not subject specific; parallels between learning a second human language and a second programming language may provide a more specific framework from which to facilitate the shift. In particular, the noticing hypothesis suggests merit in deliberately pointing out differences between the two, and an experiment is suggested to test how best to implement this idea.**

Keywords—blocks-based; typing; teaching; transfer

I. Introduction

Scratch Day celebrations are on the rise; coding puzzle apps abound; and the Hour of Code has introduced more girls to computer programming in the last two years than in several decades before it combined. Scratch, AppInventor, Blockly, Tynker, and Hopscotch have all arisen to provide an introduction to algorithmic and computational thinking for students who haven't mastered typing or would be put off by the unforgiving syntax of an industry-standard language. The question students and teachers are increasingly asking, then, is what comes next. How best can teachers facilitate the transition to a typing language?

II. Experiment Proposal

A. Motivation

Research into this problem has been informed primarily by what is known generally about the process of transfer and has focused on mediating transfer of programming languages either by implementing the same algorithm in two languages or creating an environment in which students can switch back and forth between two languages, or some combination of these approaches [1][2][3]. However, programming languages are just that -- languages. While they have been carefully and artificially constructed rather than having evolved naturally, they certainly still have grammars and other features shared with human languages. Considering language transfer not just generally but from a linguistic standpoint might provide new insights: we may not be able to raise children as "native" speakers of programming languages, but now that a growing pool exists of students who have a blocks-based first programming language, possibilities exist for borrowing from what we know about second language acquisition (SLA) of human languages. Linguist Richard Schmidt's "noticing hypothesis," for example, proposes that cognitive comparison between one's first and second language (L1 and L2) is a powerful catalyst in language acquisition. Debate is ongoing, however, about how explicitly those comparisons should be made; what I propose here is an investigation of the question with regard to programming languages.

B. Design

I suggest a controlled experiment with three groups: one taught the terminology for comparing their blocks-based L1 to typing L2 while learning L1 (e.g. "These blocks that tell something to repeat over and over are called loops"), one taught L1 with no attempt to foreshadow an L2 but having them explicitly connected in the teaching of L2 (e.g. "This is a while loop; it works the same way that the repeat until block worked in Scratch"), and one taught both languages in succession but left to make the connections on their own. The latter seems illogical, particularly given the work of Dann et al. on mediated transfer [1], but resembles the immersion approach to human language learning (in which use of L1 is effective forbidden), and might serve as a useful control.

Students' success in making the transition could be measured in any number of ways: self-evaluation of proficiency and/or comfort; dropout rates for typing-language classes or programs; and/or teacher analysis of students' output and process.[1] It may also be interesting to borrow again from human language acquisition and think about students' ability to adhere to style conventions of the typing language: that is, do students develop an "accent" from their blocks-based language, and which teaching method, if any, is the most useful in minimizing it? To provide an example, blocks-based Scratch currently includes the ability to declare

[1] Ideally, one of these dimensions should be selected rather than attempting all of them at once; it may be that one approach leads to better student comprehension but another correlates with higher student confidence, for example. Which is most interesting, and what constitutes a successful transition, likely depends on educational philosophy.

variables but no way to specify their type. To determine whether variable *levelComplete* is true, then*, if levelComplete = true* is not only neither redundant nor poor style, as it would be in Java, but the only way to accomplish the check. This sort of construction is common among beginning Java students in general, but would having Scratch background make it harder to learn to write *if levelComplete* instead?

III. Other Considerations

A. Student Interest

This proposal ignores altogether the question of *who* needs to transition from blocks-based to typing languages in the first place. As the answer is likely to rest more in philosophy of computer science education – that is, *why* it's important that students learn to code, and which non-programming aspects of CS are useful for all learners – rather than experimental data, this was deliberate. However, it is important to bear in mind; whether students have self-selected to make the transition or are having it imposed on them is likely to have a dramatic effect on their motivation, which is well known to affect learning.

B. Student Age

Human language teaching points at the fact that students' age is likely to matter enormously in which approach is most beneficial - one of the most concrete differences between child and adult learners of a second language is that young learners maintain their ability to absorb and internalize grammar, whereas adult learners are much more capable of metacognition and gain more from being taught rules explicitly. In keeping with this, it seems reasonable to hypothesize that the first of the three above approaches would have much more to offer older learners than young ones. If that is indeed the case, when in the developmental process does that shift occur? And furthermore, do programming languages, like human languages, have a "critical period" in which a child should be exposed to one in order to become fully proficient later in life? Much research remains to be done.

References

[1] W. Dann, D. Cosgrove, D. Slater, D. Culyba, S. Cooper, "Mediated transfer: Alice 3 to Java." Proceedings of the 43rd ACM technical symposium on Computer Science Education (SIGCSE '12). New York: ACM, 2012, pp. 141-146.

[2] Y. Matsuzawa, T. Ohata, M. Sugiura, and S. Sakai. "Language Migration in non-CS Introductory Programming through Mutual Language Translation Environment." Proceedings of the 46th ACM Technical Symposium on Computer Science Education (SIGCSE '15). New York: ACM, 2015, pp. 185-190.

[3] M. Armoni, O. Meerbaum-Salant, and Mo. Ben-Ari. "From Scratch to "Real" Programming." Trans. Comput. Educ. 14, 4, Article 25 (February 2015).

Learning Analytics for the Assessment of Interaction with App Inventor

Mark Sherman
Department of Computer Science
University of Massachusetts Lowell
Lowell, Massachusetts
msherman@cs.uml.edu

Fred Martin
Department of Computer Science
University of Massachusetts Lowell
Lowell, Massachusetts
fredm@cs.uml.edu

***Abstract*—Position statement on the ongoing development of an instrument to capture snapshots of App Inventor projects as students are developing, allowing examination of the students' processes. Data mining techniques will be used to process the code snapshots, identifying patterns of behavior. Future tools may include real-time assessment, and improved taxonomy of programming process.**

Keywords—programming, learning analytics, assessment, blocks, process-oriented

I. Introduction

Learning analytics offers new insights into the learning processes of students in computing. In our prior work, we developed an assessment rubric that was based upon the completed state of a student's work product [6]. Here, we present a plan for a process-oriented assessment of students work.

II. Research Concept

Add instrumentation to app inventor that will capture the state of the project, including code, the design, and the live-running app interaction, live as the user builds their app. Evaluate this data to discover user interactions with App Inventor that are not captured in wor

III. Research Questions

Can this form of data be used to indicate anything about student learning? How does a block-based environment affect this analysis? How does a live-running environment effect this analysis? Comparison against textual languages is addressed below, in Future Work.

Orthogonal to being a blocks language, App Inventor is also purely event-based. What insights can be found with these data about users interacting with event-based programming? What insights can be found about users interacting live with their app while programming?

Can real-time assessment instruments be developed from this technique? What improvements to the ex-post-facto instrument need to be made to allow for real-time analysis? Can relevant interventions for students be automatically triggered by their work?

IV. Background

The core idea is to apply big data analytics to student work, but not just the final product- instead, collect snapshots of the student program, live, as they work on it. This ability to see the process of programming in situ, not just the final product, was a huge advancement, with literature really picking up with and around the time of Piech 2012 [5].

Another data-mining paper, Berland 2013, focused on tinkering [1]. They used data mining techniques, with human-selected factors and machine-generated code state clustering. They analyzed one problem in a specific-domain educational language in a non-formal summer camp environment. They isolated certain factors, such as number of unique primitives in a program, as important for clustering the program snapshots into related "states," and how students transitioned through states, over time. This method of time-based assessment was a finding. Their analysis leaves open the opportunity for additional analysis, as it only looks at the entire cohort in aggregate, and treats that averaged data as a single, generalized student. This approach did not answer how any specific students moved through their program development.

Piech used unsupervised machine learning to perform a more robust analysis. The test problem was in Java, and in a large, university CS1 class. The problem studied was a variant of Karel the Robot, which provided a problem-constrained set of tools and API calls. They built a pattern of states of code, and mapped when and how often the students transitioned among these states. They were able to group the students into three groups based on their state transition patterns. They also looked at the students' mid-term grades, and found significant and consistent differences among those three groups. Students of a particular group performed similarly on the exam as others in that group. This was encouraging, that this method provides results that can be used immediately for course improvement, such is identifying flailing students, and was validated against a traditional assessment- the exam.

Building from Piech, another work, Blikstein 2014, examined that data further, and performed an additional study [2]. Results include that time spent tinkering did not relate to course performance. However, the ability to shift mental modes, such as from tinkering to planning, did relate.

978-1-4673-8367-7/15/$31.00 ©2015 IEEE

Many of the techniques outlined by these papers are already in use. Lipman built a system that shares similar goals with the one we propose here, which collected fine-grain, atomic interaction data from the development environment in a university CS1 course [4]. Such studies are excellent templates to leverage for the work discussed here.

V. Research Design

Most of the studies in related literature implemented a highly-constrained problem for the students to solve under observation. Piech used Karel the Robot [5], and Berland used another limited-input virtual robot [1]. Our first study will implement a similarly constrained problem, where the we will have a reduced world in which the students will act. This study will be conducted with a small cohort, both with the instrumented programming environment, and under researcher observation of their behavior. This small test, with both observation and data history, will be used to calibrate our understanding of what the data from the instrument may mean. With this knowledge we will be able to finalize the design for the second-stage experiment.

The second phase experiment will take place within a CS Principles MOOC. One particular assignment will be selected for study, just as with other. This environment for the students will be less constrained, but still have a specific goal and well-defined expected path towards that goal. This experiment will have a large number of participants, likely hundreds, and will not involve in-person observation, relying entirely on the instrumentation data.

We are considering a third phase, where we would invite any student working on App Inventor in any class in the world to participate. They would declare to us, by a short entry survey, what they are trying to accomplish, and opt in to data collection while they work towards their goal. Knowing their (self-reported) goals, we will expand our analysis techniques to work on larger, less specific data sets. These students will not all have the same goals, but much of their projects and their interactions with their projects may have similar properties that we can extract and compare against their coded goals, in aggregate. This study could have upwards of 10,000 students involved.

We are expecting to discover a certain class of work trajectories, such as: repeatedly undoing and redoing a change, exhaustively looking through block menus without recruiting a block, and removing code from execution path, putting blocks off to the side.

The trajectories in this list may or may not appear as strong signals in testing, but serve as examples of the types of patterns we believe we will identify with this method. This class of patterns may be suitable for a future real-time analysis tool, which has the potential to assess when specific students require particular interventions to improve their understanding.

VI. Future Work

Studies do exist that compare graphical and text languages with students, such as Lewis 2010, but the intersection of comparable features and comparable learning goals is often small, limiting the experimental design, and therefore offer limited generalizability. All studies that try to directly compare any two systems will have to overcome that different systems do better at certain things, usually by design. In designing the experiment, it is all too easy to set up one language or the other to fail. David Weintrop is working on overcoming this bias, and is developing assessments that work equally well in Snap!, JavaScript, and hybrid environments [7].

There are two kinds of analysis in this space: real-time and ex-post-facto. The methods described here are the latter-depending on snapshots captured from an entire and complete programming experience. The former is also of interest, as such tools could monitor students while they are working, detecting patterns that signal gaps in comprehension, and triggering a need-based intervention to aid that student. The intervention could be a pop-up referring to a specifically informative help page, a notification to an instructor to attend to the student, or something else. Follow-up work on tools like these will ensue promptly.

This paper describes a method for assessing student work progression in App Inventor (a blocks language), along with data-driven conclusions about student development behavior with this language. The methods described here may be developed further to create a meta-descriptor of programming process, which can provide a future research vector towards a neutral method of comparing learning in dissimilar languages. This meta-descriptor of language interaction will require identification of analytic measures (suitable for data analytics and/or machine learning) that are common to all (or many) languages, and a taxonomy of how those measures are represented in individual systems. This method will be informed by the work described here, and we hope to carry on this path of research for many years, towards that lofty goal.

Acknowledgment

This material is based upon work supported by the National Science Foundation under Grant No. 1433592. Any opinions, findings, conclusions, or recommendations expressed are those of the authors and do not necessarily reflect the views of the National Science Foundation.

References

[1] M. Berland, Martin, Benton, Smith, and Davis. Using learning analytics to understand the learning pathways of novice programmers. Journal of the Learning Sciences (2013), 22(4):564–599.

[2] P. Blikstein, Worsley, Piech, Sahami, Cooper, and Koller. Programming pluralism: Using learning analytics to detect patterns in the learning of computer programming. J. Learning Sciences (2014), 23(4):561–599.

[3] C. M. Lewis. How programming environment shapes perception, learning and goals: Logo vs. scratch. SIGCSE (2010), 346–350.

[4] D. Lipman. LearnCS!: a new, browser-based C programming environment for CS1. J. Comput. Sci. Coll. 29, 6 (June 2014), 144-150.

[5] C. Piech, Sahami, Koller, Cooper, and Blikstein. Modeling how students learn to program. SIGCSE (2012), 153–160.

[6] M. Sherman, Martin. The assessment of mobile computational thinking. 2015. J. Comput. Sci. Coll. 30, 6 (2015), 53-59.

[7] D. Weintrop. Minding the gap between blocks-based and text-based programming: Evaluating introductory programming tools. SIGCSE (2015). ACM.

Measuring Learning in an Open-Ended, Constructionist-Based Progamming Camp:

Developing a Set of Quantitative Measures from Qualitative Analysis

Deborah A. Fields
Instructional Tech. & Learning Sci.
Utah State University
Logan, USA
deborah.fields@usu.edu

Lisa C. Quirke
Faculty of Information
University of Toronto
Toronto, Canada
lisa.quirke@mail.utoronto.ca

Janell Amely
Instructional Tech. & Learning Sci
Utah State University
Logan, USA
jamely@aggiemail.usu.edu

***Abstract*— *In this paper we raise the issue of how to assess novice youths' learning of programming in an open-ended, project-based learning environment. One approach could be a way to apply quantitative measures to the analysis of programming education across frequent saves in a variety of open-ended projects. This paper focuses on the first stage of this endeavor: the development of exploratory quantitative measures of youths' learning of computer science concepts through deep qualitative analysis.*

Keywords—computer science education; big data; Scratch; assessment; novice programmers; constructionism

I. INTRODUCTION

Since Seymour Papert introduced the idea of constructionism [1], many computer science educators have sought to engage kids in learning through interest-driven project-based programming. This playful approach to learning-by-designing emphasizes the social and contextual factors that affect learning [2]. This approach is open and free-form, allowing users to participate and collaborate in the creation of products that are meaningful to them: no project is the same [3]. Yet while this can be motivating for learners, who can bring in outside interests and representations of themselves, their families, and communities into their projects [3], it brings challenges for educators and researchers who are tasked with measuring and assessing their learning.

Though constructionist programming environments allow for diverse programming styles such as "structured" or "bricoleur" [4], there is no guarantee that all students will learn the basics of key programming concepts. As Maloney, Peppler, Kafai, Resnick and Rusk [5] showed in their two-year study of a Computer Clubhouse, few students ever began using programming concepts such as variables, conditionals, randomization, or Booleans on their own. Similarly, in their study of a random sample of 5000 active users on the Scratch online community (scratch.mit.edu), Fields, Giang, and Kafai [6] noted that only 16% of users used key programming concepts such as variables, conditionals, and Booleans in their programs. This raises an important question of how educators can deepen and broaden students' grasp of programming in ways that allow for the creativity and ownership of constructionist learning environments.

This work was supported by a collaborative grant from the National Science Foundation (NSF#1319938) to Deborah A. Fields, Taylor Martin and Sarah Brasiel. The views expressed are those of the authors and do not necessarily represent the views of the National Science Foundation, Utah State University, or the University of Toronto.

II. BACKGROUND

In recent years, many programming environments have been created to support students' learning of computer science concepts without typical challenges of spelling, syntax, and punctuation [7]. Building on the work of Logo [1], visual block-based languages such as Scratch [8] and Alice [9] enable novice programmers to focus on learning programming concepts without worrying about these other issues [10]. Visual programming environments make coding simpler by making it impossible for users to make syntax errors and providing immediate feedback. Yet despite the availability of languages that support novices' entry into computer programming, there is a dearth of research on how novice programmers, and children in particular, learn computer science concepts in these environments, especially when they are used in a constructionist manner [e.g. 11]). The studies that do exist highlight the importance of key computing concepts as well as the challenges that novices may face in mastering these.

One difficulty constructionist educators have faced is how to evaluate students' programming or more broadly computational thinking [12], especially in interest-driven environments where projects are open-ended without a "correct" solution [13]. One approach is to look for the application of specific programming concepts. Brennan and Resnick [10] argue for several computer science concepts they see as central, including sequences, loops, parallelism, events, conditionals, operators and data. They separately consider computational practices such as debugging and remixing (see also [14]) and computational perspectives including expressing, connecting, and questioning.

Many researchers have applied the idea of programming concepts in evaluating students' programs, but often these are applied in very basic ways (e.g. frequency counts), especially on a larger scale where programs cannot be individually analyzed. For instance, Maloney et al [5] and Fields et al [6] differentiate students by the mere presence of particular blocks of code. The Scrape tool developed by Wolz, Hallberg, and Taylor [15] shows somewhat more breadth in students' code by identifying every block used in a single or series of Scratch projects amongst sets of particular blocks. Still, applications of this tool generally presume that the presence of select blocks denotes students' learning of a concept. While these

978-1-4673-8367-7/15/$31.00 ©2015 IEEE

approaches can differentiate between the programming patterns of hundreds or thousands) of students, they do not necessarily show that students have developed competence or mastery with a particular programming concept. Brennan and Resnick [10] highlight this point, noting the major differences found between students' projects using Scrape and their actual understanding of how their code worked.

In this position paper we discuss the issue of bringing rigor to constructionist programming environments and assessments of learning in these spaces. We share our "studio model of pedagogy" and our efforts to develop measures with which to assess student learning at a larger scale. As part of our analysis we have developed a set of measures of programming concepts including initialization, events, parallelism, conditionals, variables, randomization and Booleans. We developed these measures through deep qualitative analysis of changes in students' projects and have begun to apply them analytically across more than 600 project saves per student (roughly every 2 minutes). Below we briefly discuss the pedagogy we applied to three Scratch Camps followed by a brief introduction to some of the measures we are creating. We hope to engage in conversation at the workshop about authentically assessing students' learning while maintaining the richness of constructionist environments for creating with programming.

III. Scratch Camp: A Creative, Project-based Pedagogy

The questions examined in this position paper emerge from a study of children's learning trajectories with Scratch. The project includes data collected at three, week-long Scratch Camps held in summer 2014 at an intermountain university with the local 4-H club. Across five days and 25 hours of programming, campers aged 10-13 made a series of creative projects. Campers were mentored and taught by one professor and three graduate students.

A key inspiration for the design of our camps was a "studio model" of pedagogy [16], a prominent approach in arts instruction. This model involves three elements: (1) complex, authentic, real world projects, (2) guided problem solving with creative constraints, and (3) externalization and reflection with public feedback. In his study of art studies, Sawyer [16] found that art teachers created a series of carefully structured projects that necessitated students' learning of particular techniques. Although projects are open-ended and allow for creative expression, key constraints help to structure the project, focusing students' creative efforts on the targeted problems. Finally, open discussions and critiques of student work allowed the process to be transparent, as all students could learn from instructors' critiques and see changes in each other's projects.

It was with this theory of pedagogy in mind that we created a series of open-ended, genre-specific project challenges with constraints intended to impel students to learn particular programming techniques. In an earlier study, Fields, Vasudevan, and Kafai [17] successfully targeted the programming concepts of initialization and synchronization through a music video design challenge. We included that as one of the following projects in this series of camps: Scribble time & Name project (Day 1), Story project (Day 2), Music Video (Day 3), Video Game (Day 4), and Free Choice Project (Day 5). Scratch Camp concluded with a special gallery walk where interested parents could browse campers' completed work and attend the graduation ceremony.

IV. Example of Measures

Due to space limitations, we share just one example of some measures we have developed to see students' progress with events and parallelism over time. The graph below shows measures focused on broadcasts and use of green flags that illustrate students' use of events and parallelism in programming (see Figure 1). They show how these measures of programming stand out over the accumulated progress Virginia, age 10, made throughout the camp.

In Figure 1, the red line represents the use of working broadcasts. In essence these are "events" in Scratch—issuing a broadcast and linking it with a receive triggers a new set of commands. (Note: this measure only includes broadcasts that have a matching receive and are connected to a working script.) Looking at the graph, it is possible to see how Virginia began to use broadcasts with corresponding receives over time. The linear dotted grey lines delineate the five days of the Scratch Camp. She began to use broadcasts during the "Story" project which began at the very end of Day 1 and continued into the majority of Day 2. One of the goals of this project was to authentically introduce a reason to use broadcasts, and creating a narrative with multiple scenes successfully provided a context for that learning. Though some campers introduced broadcasts during the first hour of their story program, Virginia caught on later (we verified this with qualitative data analysis), despite multiple introductions to the concept on Days 1 and 2. The graph also shows the growing size of the projects as more broadcasts are introduced in the Story and then immediately plateau at the start of the Music Video project.

Two other measures show uses of parallelism in students' programming: the green line shows the number of broadcasts with more than one "receive" in a given project, while the blue line illustrates the number of sprites that had multiple green flags with scripts in a given project.

We found that using multiple receives with a single broadcast or using multiple green flags in a single sprite introduced a interesting level of abstraction, a form of parallelism, into students' programming. It generally took some time before students caught on to the utility of this idea of triggering multiple things to start at the same time, especially within a single sprite or with a single broadcast. Virginia began to use these in the development of her story projects almost as soon as she began to use broadcasts. Then Virginia applied this strategy in a different way in in her music video (Day 3) using a large number of green flags in many sprites, returning to this strategy with fewer sprites in her video game on Day 4. Looking at these measures as a whole shows students' progressions across using a concept in multiple projects and in different ways.

This example shows the beginnings of how we are trying to evaluate the quality of programming and trajectories of

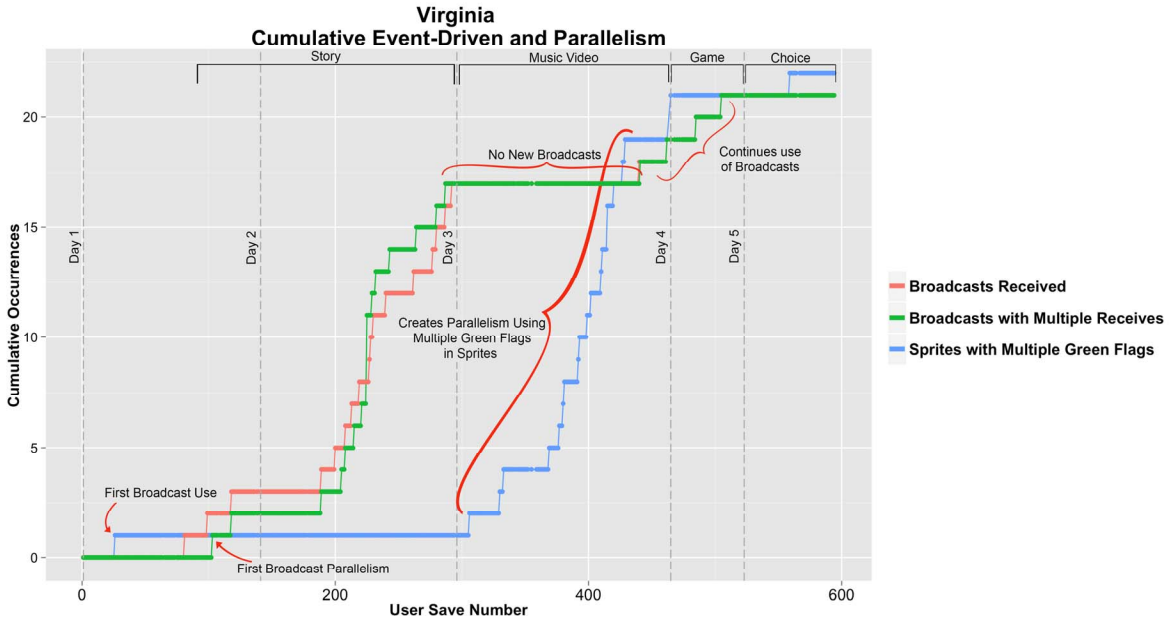


Fig. 1. Virginia's use of events and parallelism across project saved throughout the Scratch Camp.

learning in a relatively open-ended environment. Our other measures include categories of initialization, events, parallelism, conditionals, variables, randomization and Booleans. Throughout this process we have gone back and forth between studying students' learning through hand-analysis of projects (with videos and field notes providing additional context for their learning) to the development of quantitative measures. We are in the process of evaluating the quantitative measures to see which ones are most valuable, at least in the context of the projects at the Scratch Camps. This is but one approach to the challenge of finding ways to authentically trace students' learning in open-ended environments. We look forward to discussions on this issue of supporting students' interest and creativity while challenging them to deepen their skills and thinking.

ACKNOWLEDGMENT

Special thanks to the Scratch Team, Jason Maughan, Xavier Velasquez, Tori Horton, and Katarina Pantic.

REFERENCES

[1] S. Papert, Mindstorms: Children, computers, and powerful ideas. New York: Basic Books, 1980.

[2] Y. B. Kafai, "Playing and making games for learning: Instructionist and constructionist perspectives for game studies," Games and Culture, vol. 1(1), 2006, pp. 36-40.

[3] K. Peppler and Y. Kafai. "Creative coding: Programming for personal expression." In The Proceedings of the 8th International Conference on Computer Supported Collaborative Learning (CSCL). Rhodes, Greece, 2009.

[4] S. Turkle and S. Papert. "Epistemological pluralism: Styles and voices within the computer culture." Signs, 1990, pp. 128–57.

[5] J. H. Maloney, K. Peppler, Y. Kafai, M. Resnick, and N. Rusk. "Programming by choice: Urban youth learning programming with Scratch." ACM SIGCSE Bulletin, vol. 40(1), 2008, pp. 367–371.

[6] D. A. Fields, M. Giang, and Y. Kafai. "Programming in the wild: Trends in youth computational participation in the online Scratch community." In Proceedings of the 9th Workshop in Primary and Secondary Computing Education, 2–11. ACM, 2014.

[7] C. Duncan, T. Bell, and S. Tanimoto. "Should your 8-year-old learn coding?" In Proceedings of the 9th Workshop in Primary and Secondary Computing Education, 60–69. ACM, 2014.

[8] M. Resnick, J. Maloney, A. Monroy-Hernández, N. Rusk, E. Eastmond, K. Brennan, A. Millner, et al. "Scratch: Programming for all." Comm. ACM. 52, no. 11, 2009, 60–67.

[9] C. Kelleher and R. Pausch. "Lowering the barriers to programming: A taxonomy of programming environments and languages for novice programmers." ACM. Comput. Surv. vol. 37(2), 2005, pp. 83–137.

[10] K. Brennan and M. Resnick. "New frameworks for studying and assessing the development of computational thinking." Paper presented at annual American Educational Research Association meeting, Vancouver, BC, Canada, April 2012.

[11] S. Grover, R. Pea, and S. Cooper. "Designing for deeper learning in a blended computer science course for middle school students." Computer Science Education, vol. 25(2), 2015, pp. 199-237.

[12] J. Wing, J. "Computational Thinking." Comm. ACM. 49(3), 2006, pp. 33-35.

[13] P. Blikstein, M. Worsley, C. Piech, M. Sahami, S. Cooper, and D. Koller. "Programming pluralism: Using learning analytics to detect patterns in the learning of computer programming." J. Learn. Sci. 23(4), 2014, pp. 561-599.

[14] L. Murphy, G. Lewandowski, R. McCauley, B. Simon, L. Thomas, and C. Zander. "Debugging: The good, the bad, and the quirky – a qualitative analysis of novices' strategies." In ACM SIGCSE Bulletin, 40: 163–67. ACM, 2008.

[15] U. Wolz, B. Taylor, and C. Hallberg. "Scrape: A Tool for Visualizing the Code of Scratch Programs." Presented at the ACM SIGCSE, Dallas, Texas, 2010.

[16] K. Sawyer. "Learning how to create: Toward a learning sciences of art and design." In van Aalst, J., Thompson, K., Jacobson, M.J., & Reimann, P. (Eds.), The Future of Learning: Proceedings of the 10th International Conference of the Learning Sciences (ICLS 2012), Volume 1, Full Papers. International Society of the Learning Sciences: Sydney, NSW, Australia, 2012, pp. 33-39.

[17] D. Fields, V. Vasudevan, and Y. Kafai. "The programmers' collective: connecting collaboration and computation in a high school Scratch mashup coding workshop." Interact. Learn. Envir. in press.

Profiling Styles of Use in Alice

Identifying patterns of use by observing participants in workshops with Alice

Leonel Morales Díaz
Universidad Francisco Marroquín
6ta Calle Final, Zona 10
Guatemala, Guatemala 01010
litomd@ufm.edu

Laura S. Gaytán-Lugo
Universidad de Colima
Km. 9 Carretera Colima - Coquimatlán
Coquimatlán, Colima, México 28300
laura@ucol.mx

Lissette Fleck
Universidad Francisco Marroquín
6ta Calle Final, Zona 10
Guatemala, Guatemala 01010
afleck@ufm.edu

***Abstract*— During workshops on computer animation using Alice, a free platform for three dimensional computer animation created by Carnegie Mellon University, we detected and observed a series of different patterns of use of the platform by attendants. Most participants would start following instructions as precisely as possible. Within a short time some would divert to create visually attractive scenes worrying little about movement and action, others would put two or more characters in the scene and make them talk telling a story, and a fourth group would explore by their own advanced features and functions of Alice well beyond the content of the workshop creating complex and action-rich animations. As these styles kept appearing in every event, a more systematic observation was attempted by designing a form for observers and a survey for participants. Although we have had only one opportunity to test our instruments, we believe that the study is worth continuing and the results may turn revealing not only for Alice but for other block programming environments. In this article we describe the styles observed and the instruments designed for observation. The preliminary results are also discussed.**

Keywords*— *Alice; Computer Animation; Styles of Use; First Programming Environments.

I. Introduction

First programming environments like Scratch [2], Alice [1], Kodu Game Lab [8], Stencyl [9] and several others, are designed to deliver a pleasant experience to kids, youngsters and even adults wishing to learn computer programming. It is important then to analyze the extent to which each one succeeds in achieving that goal.

Equally important is to discover how their users use the tools, what tasks they attempt, the actions they prefer to perform, the outcomes they enjoy obtaining, and the way they relate to others while using the environments. Such discovery can be made on purpose through experiments and controlled settings or by a sort of intuition after noticing patterns of use in different groups and events.

In this paper we will present four profiles of what we call "styles of use" that we detected over time in participants in computer animation workshops with Alice, a free environment for learning to program. After conducting several of such workshops we started noticing that participants engaged with the platform in one of four different styles: instruction follower, scene designer, dialogue storyteller and action animator. In an attempt to validate our observations we designed a survey for participants and a form for an observer to fill for each participant and tested both instruments in one workshop, the results were highly revealing although the number of participants and the setting of the experiment were less than optimal.

We plan to continue with the research in order to find if the identified styles appear consistently across time, diversity of participants, and events, if the descriptions of the styles are clear enough to allow reproducibility of our observations and if there is some tendency in the frequency of observation of each style.

Finally we will argue on the importance of identifying styles of use not only in Alice but in every first programming environment in order to allow the adaptation of teaching and evaluation methods and even make changes in models and designs of these environments.

II. Alice and the Observed Styles of Use

A. Learning to Program with Alice

Alice is an interactive graphical environment for programming three dimensional computer animations that was created and is maintained by Carnegie Mellon University [4]. In the beginning it was thought as a tool for rapid prototyping of 3D animations but soon their creators realized its potential as a first programming environment [3]. It can be freely

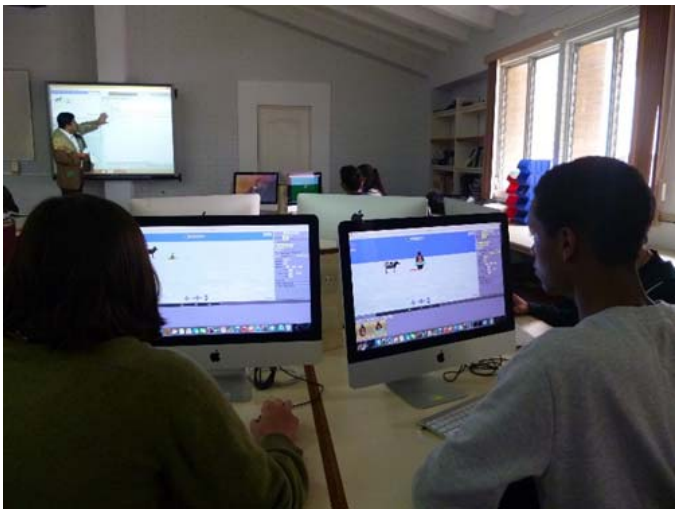


Fig. 1. A workshop on computer animation with Alice.

downloaded and installed in multiple operating systems including Linux, Microsoft Windows and Apple Mac OS. It uses block programming mechanics to produce animations in a 3D space.

In Alice programmers create scenes choosing from predefined stages and adding 3D models from a well populated gallery. Then they script the movements, conversations, property changes and other animation actions by dragging and dropping instructions, methods and functions into the scripting area. Although it is possible to use complex programming structures and algorithms it only requires a few lines dropped to produce interesting and fun animations. As in other block programming environments syntax errors are not possible. All of this makes Alice an attractive and enjoyable platform for learning to program.

However, in announcing our workshops we have avoided the word "programming" and instead used the phrase "Computer Animation with Alice" and the workshops have actually offered instruction on the different forms of computer animations that can be made using Alice. Participants do program, and are introduced to computer programming only within the context of computer animation.

Alice allows learners to test and explore its features in a robust environment of immediate visual feedback. Its friendliness with users and its flexibility to create both simple and complex animations is key for the apparition of different styles of use. In the workshops we often invited participants to create and explore on their own and included free time for that purpose. The combination of a supportive platform and freedom to work on personal ideas can be connected to the "support many paths, many styles" principle, identified by Resnick and Silverman in [10].

B. Instruction Followers

Participants in computer animation workshops with Alice usually start the workshop following the instructions and explanations given. After a while some will divert to exploring the environment on their own, trying and testing the features as they discover them.

Fig. 2. Participant in a workshop adding elements to a scene.

Fig. 3 Time to work freely and a supportive platform enable the apparition of styles of use.

Those that persevere following instructions and paying attention to explanations seem to have been captured by the process and need to continue on it until the end. They are the instruction followers.

Instruction followers try enthusiastically to obtain an animation that is as exact a copy as possible of that the instructor is exposing. They place elements in the scene and in the script in the same position as the instructor, use the same scene settings, and imitate details like the size of the display windows.

Their questions are usually related to how to achieve the same results the instructor obtained: "why is this not working like yours?", "what did you do previously?" or "how did you get that?" They seem to have a sense of accomplishment by producing an animation that looks and behaves like that shown in the workshop. Exploring and discovering by their own is usually delayed until they are sure their work is an acceptable copy and even then they are willing go back to follow instructions when the workshop resumes.

C. Scene Designers

In Alice there is a scene setup mode that lets users add all types of characters and things to the scene making it rich and colorful. It is also possible to use predesigned scenes inspired in popular culture themes and change properties of the ground to make it look like grass, ice, jungle, desert, water, and others.

Some people find this fascinating and start changing and experimenting with different designs of the scene. Alice allows repositioning objects, resize, change color, rotate and other actions, and they seem to have fun doing all that worrying little about scripting actions. In fact they spend most of their time in scene setup mode and go back to edit code mode infrequently.

They may show some interest in learning to script camera movements, a very interesting feature of Alice, because with it they can use camera effects to explore different views of their creation. Alice is for them a sort of artistic composition tool in which they have a digital canvas to work. When they have the

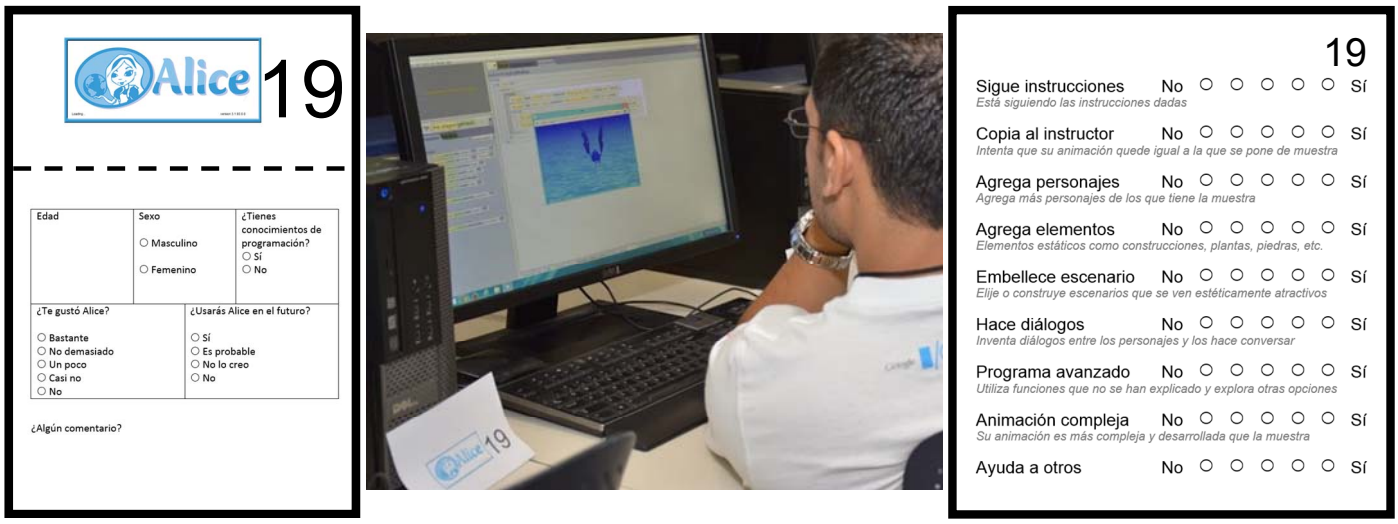


Fig. 4 The survey filled by participants (left) and the observer form (right) both in Spanish. Each participant worked in a station marked with a number (center).

opportunity they ask questions related to how to resize, reposition or change the appearance of objects.

D. Dialogue Storytellers

In Alice it is possible to script objects to say or think something with bubble or cloud callouts that display text messages for a few seconds. The programming instructions to make this are easy to find and some persons start building simple and complex dialogues between two or more characters. They are the dialogue storytellers.

They enjoy, sometimes with visible delight, building these dialogues and watching the resulting animation, adding movement and camera effects only to support the main story. Their code usually contains lengthy sequences of consecutive "say" and "think" instructions.

Working in pairs or in groups goes well with them. They like to receive suggestions for their story and also like letting others watch what they did.

E. Action Animators

In Alice it is possible to employ complex structures of code to produce action-rich animations involving multiple characters and objects. Some discover these features and seem to have a faculty to master them quickly and by their own. They are the action animators.

Action animators enjoy producing elaborated scenes that employ advanced functions and elements not explained in the workshop. For example, certain models in Alice have joints and parts that move independently but need to be coordinated to produce a realistic action as in walk that requires legs steps and arms swings, or wheels that rotate and bump when vehicles move, and so on. Action animators are capable of achieving these effects quickly and create scenes that combine and display them sometimes impressively.

III. Verifiying Observations

As sound and reasonable the detected styles may seem a verification process is needed in order to establish if they can be consistently observed and studied or if otherwise they are an artifact of our workshop methodologies.

In a first attempt to perform the verification we designed a survey to be answered by workshop participants at the end of the event, with basic questions regarding age, sex, previous knowledge of computer programming, the extent to which they liked Alice, how likely they believed it is that they will use Alice in the future and a space for free comments (Fig. 4 left).

In conjunction with the survey a person not enrolled in the workshop would be appointed as observer to collect data in a form with nine qualities to score in a Likert-type scale for every participant: follows instructions, copies instructor, adds characters, adds elements, embellishes scenario, creates dialogues, uses advanced programming, produces a complex animation, and helps others (Fig. 4 right).

Fig. 5. The workshop in which the instruments were tested.

Both tools, the survey and the observation form, were tested for the first time in a workshop in May 2015, and with the collected data we created a model of analysis that produced interesting results. The scores of relevant qualities for each style were used to calculate an index that graded the affinity of the person with the style. After calibrating parameters in formulas and making adjustments we found that classifying a person in a style required an algorithm and not simply selecting the highest index.

With the necessary caution on the preliminary nature of our results, knowing that they come from instruments that need refinement and even other complementary instruments that are not yet designed, we processed the data and present and discuss what we obtained.

A. Preliminary Results

In the following tables we present the results from applying the designed instruments in a workshop on computer animation with Alice with 24 participants. Although all of them answered the survey, 2 left blank the age question and the data for one was discarded because of lack of observation information.

Table I summarizes participants information. Table II shows the raw frequencies observed for each style.

TABLE I. WORKSHOP PARTICIPANTS INFORMATION SUMMARY

Description	Total	Discarded [a]
Answered the survey	24	
Provided information on age	22	
Women	9	
Men	15	1
Had previous knowledge of computer programming	13	1
No previous knowledge of computer programming	11	
Under 20 years old – teenagers	11	
20 years old and above – adults	11	1
Gave Alice highest mark in "how much did you like Alice?" question	15	1
Gave Alice less than highest mark in "how much did you like Alice?" question	9	

[a] Data from one participant had to be discarded because the observation form lacked key information.

TABLE II. RAW FREQUENCIES FOR EACH STYLE

Style	Cases	Percentage
Instruction Follower	11	48%
Scene Designer	3	13%
Dialogue Storyteller	5	22%
Action Animator	4	17%
Total	23	100%

Women exhibited only two of the four styles, instruction follower and dialogue storyteller, while among men the four were observed. Even though the group of women was small, only 9 participants, it accounted for 3 of the 5 dialogue storytellers (60%) as shown in Table III.

TABLE III. FREQUENCIES OF EACH STYLE AMONG WOMEN AND MEN

Style	Women		Men	
	Cases	*Percentage*	*Cases*	*Percentage*
Instruction Follower	6	67%	5	36%
Scene Designer	0	0%	3	21%
Dialogue Storyteller	3	33%	2	14%
Action Animator	0	0%	4	29%
Total	9	100%	14	100%

The previous knowledge of computer programming appears to have little influence in the adoption of a style according to the frequencies observed in both groups and shown in Table IV. The percentage for each style is similar in these two groups.

TABLE IV. FREQUENCIES OF STYLES ACCORDING TO PREVIOUS KNOWLEDGE OF COMPUTER PROGRAMMING

Style	Had Knowledge		No Knowledge	
	Cases	*Percentage*	*Cases*	*Percentage*
Instruction Follower	6	49%	5	46%
Scene Designer	2	17%	1	9%
Dialogue Storyteller	2	17%	3	27%
Action Animator	2	17%	2	18%
Total	12	100%	11	100%

On the other hand, age seems to be important for style selection as can be derived from data in Table V. Younger participants tended much less to adhere to the instruction follower profile while for adults it was the preferred one although the other three were also observed.

TABLE V. FREQUENCIES OF EACH STYLE AMONG TEENAGERS AND ADULTS

Style	Teenagers		Adults	
	Cases	*Percentage*	*Cases*	*Percentage*
Instruction Follower	4	37%	7	70%
Scene Designer	1	9%	1	10%
Dialogue Storyteller	3	27%	1	10%
Action Animator	3	27%	1	10%
Total	11	100%	10	100%

Finally, when grouped according to the mark given to Alice when rating how much they liked it ("how much did you like Alice?" question), we found that the action animator style only

Fig. 6. An observer walked behind participants filling a form with scores in various attributes.

appeared among those that gave the highest grade as shown in Table VI.

TABLE VI. FREQUENCIES OF EACH STYLE ACCORDING TO HOW MUCH PARTICIPANT LIKED ALICE

Style	Liked Alice Most		Liked Alice Less	
	Cases	*Percentage*	*Cases*	*Percentage*
Instruction Follower	6	43%	5	56%
Scene Designer	1	7%	2	22%
Dialogue Storyteller	3	21%	2	22%
Action Animator	4	29%	0	0%
Total	14	100%	9	100%

Again, the preliminary character of these results makes impossible to derive further conclusion although they are important as a baseline and as a comparison set. With new iterations of the application of the instruments after making the appropriate corrections, we hope to identify a tendency and publish findings with better support.

IV. RELEVANCE OF THE PROJECT

First programming environments are powerful tools that support teaching and learning complex computer science concepts. It is thus important to study not only what are the most effective design principles for their construction and evolution but also how their target audiences uses them. Understanding the different styles in which users prefer to engage with these platforms may lead to incorporate features that make it easier to discover the possibilities that match personal interests.

There is a reasonable amount of studies concerning the design, construction and utilization of first programming environments for education or for entertainment, see [5], [6] and [7] for examples. However, the way actual users engage with the environments after initial instruction on their operation has been provided, which features and functions they prefer, or what type of outcomes they attempt is little understood.

When styles of use are fully identified for a platform the instruction methods for prospective users can be adapted to increase efficiency in the education process. It is impossible to tell if the proposed styles for Alice are going to be validated as such but because their identification is the result of observation and attempts of validation that have encouraging results until now, there are good reasons to continue.

There are research questions that could be addressed after the validation is accomplished: do different styles of use require different methods of instruction? Is it possible to identify styles of use for other environments in a similar fashion? Would the styles for other environments be the same as for Alice? And several others.

V. CONCLUSIONS

In this paper we have introduced the description of four styles of use for the Alice graphical environment: instruction follower, scene designer, dialogue storyteller and action animator. The styles were identified after several iterations of computer animation workshops the authors conducted using Alice and patterns of interaction and use were observed in the participants. Grouping the patterns according to similarity led us to profile and describe the styles.

In order to validate that the styles were not an artifact of the methodology used during the workshops and that they can be consistently observed in different settings and with different groups, a pair of instruments were created and tested: a survey for participants that collects basic information and a form to be filled by an observer that registers scores of participants in nine attributes that are later used to classify the person in one of the styles.

Preliminary results obtained from the application of the instruments in one workshop with 24 participants were presented and discussed. The rate of adoption of styles seems to be particularly different between men and women and between teenagers and adults.

There are several limitations that prevent the generalization of the outcomes, but these limitations will hopefully be resolved incorporating corrections and improvements as they are identified and proposed. New instruments may also be created.

We argued on the importance of attending to how users interact with first programming environments on the basis of the foreseeable improvements in instruction methods when they adapt and harmonize with styles of use. As pointed in the article studying users and finding how to they engage with the environment is an area of opportunity for research.

The extension of the search for styles of use to other platforms and programming environments even those that are more advanced and complex was presented as a plausible possibility. The supportiveness of “many paths, many styles” found in Alice can be reasonably expected in several other platforms.

With the iteration of the observation experiment using improved and new instruments in collaboration with other academic institutions we are confident that more reliable data will be obtained and better supported findings will be reported.

REFERENCES

[1] Carnegie Mellon University, "Alice.org," 2015. [Online]. Available: http://www.alice.org/. [Accessed 28 June 2015].

[2] Lifelong Kindergarten Group at the MIT Media Lab, "Scratch - Imagine, Program, Share," 2015. [Online]. Available: https://scratch.mit.edu/. [Accessed 28 June 2015].

[3] M. Conway, S. Audia, T. Burnette, D. Cosgrove and K. Christiansen, "Alice: Lessons Learned from Building a 3D System," in *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems*, The Hague, The Netherlands, 2000.

[4] W. Dann and S. Cooper, "Education: Alice 3: concrete to abstract," *Communications of the ACM,* vol. 52, no. 8, pp. 27-29, August 2009.

[5] S. Flanagan, "Introduce Programming in a Fun, Creative Way," *Tech Directions,* vol. 74, no. 6, pp. 18-20, 2015.

[6] M. Resnick, J. Maloney, A. Monroy-Hernández, 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,* vol. 52, no. 11, pp. 60-67, November 2009.

[7] S. H. Rodger, J. Hayes, G. Lezin, H. Qin, D. Nelson, R. Tucker, M. López, S. Cooper, W. Dann and D. Slater, "Engaging middle school teachers and students with alice in a diverse set of subjects," in *Proceedings of the 40th ACM Technical Symposium on Computer Science Education*, Chattanooga, 2009.

[8] Microsoft Research, "Kodu Home," [Online]. Available: http://www.kodugamelab.com/. [Accessed 28 June 2015].

[9] Stencyl, LLC, "Stencyl: Make iPhone, iPad, Android & Flash Games without code," [Online]. Available: http://stencyl.com/. [Accessed 28 June 2015].

[10] M. Resnick and B. Silverman, "Some Reflections on Designing Construction Kits for Kids," in *Proceedings of the 2005 Conference on Interaction Design and Children*, Boulder, Colorado, 2005.

Quizly: A Live Coding Assessment Platform for App Inventor

Francesco Maiorana[1,2]
[1]Department of Electrical, Electronic and Computer Engineering, University of Catania, Viale A. Doria, 6, Catania, Italy
[2]IISS G.B.Vaccarini, Catania
francesco.maiorana@dieei.unict.it

Daniela Giordano
Department of Electrical, Electronic and Computer Engineering, University of Catania, Viale A. Doria, 6, Catania, Italy
daniela.giordano@dieei.unict.it

Ralph Morelli
Computer Science Department
Trinity College
Hartford, CT, USA
ralph.morelli@trincoll.edu

***Abstract*— There is a strong worldwide movement which is pushing for the teaching of serious computer science principles besides reading, writing and basic numeracy starting from first grade and reaching all students across all grades. This is being done through both formal initiatives carried out by international organizations and at the national level by putting forward curricula, some of which are mandatory. In order to accomplish this goal, and based on the consensus that computer science is not programming and that programming languages are a tool, visual languages have become the preferred method for teaching introductory courses in computer science. The absence of rigid syntactic rules makes them the ideal tool for focusing on problem solving and computational thinking activities. Recent reports have pointed out the need for supporting the international community of teachers by providing assessment methods, an internationally validated question repository as well as tools and assessment platforms. In this context our work presents an assessment platform for formative, summative and informal assessment of computer science competencies by using a visual language, namely App Inventor, which allows for the rapid development of a mobile app and has a strong appeal to the younger generation of students. The capability to log user activity allows the teacher to monitor the progression in the student's learning path as well as her/his solution-building approach.**

Keywords*— *Computer Science curricula, Visual languages; assessment; assessment tools and platforms

I. INTRODUCTION

There is a strong worldwide movement advocating for the teaching of serious computer science concepts, besides reading, writing and numeracy, as early as possible in the student's learning path. This has brought to international attention the need to reform computer science teaching practices in order to support its introduction in all stages of school and for all students. A remarkable example of this trend can be found in the United Kingdom when since 2014 computer science has been considered mandatory from the first year of school [1], and in the United States where a strong informal movement (https://hourofcode.com, https://code.org/learn) [2] is reforming the attitude towards computer science worldwide by engaging millions of people by way of games and activities with the aim of pushing a legislative reform to introduce computer science in all schools. The movement has flourished in many similar initiatives across the globe such as http://codeweek.eu/, http://uk.code.org/, http://codeweek.it/, and http://www.programmailfuturo.it/. These curricula are aligned with modern 21st century competency frameworks such as [3- 6] and the Framework for 21st Century Learning developed by the P21 Partnership for 21st Century Learning (http://www.p21.org/our-work/p21-framework). One of the key elements of these efforts is the importance given to visual programming languages. Even if it is well recognized that computer science is not programming, visual languages allow for a programming approach based on snapping together blocks that differ in shape and color, thus avoiding much of the syntactic burden common to textual languages. This in turn allows students to focus on the problem solving process and sharpen their computational thinking skills [7-10].

Besides this flourishing of curricula there is an urgent need for an assessment in computer science (CS) education, which is aligned with the modern curricula and competency frameworks as stated recently in [11] which recommends that the "CS education community leads the development of a curated assessment library for teachers" and launches a "call to action for the computer science education community to develop valid and reliable assessments." The importance of assessment is recognized by other international networks such as the assessment and accountability roadmap (http://www.roadmap21.org/assessment.html). The assessment has been the focus of many working groups, such as the Conference on Innovation and Technology in Computer Science Education (ITICSE), which has a long tradition in computer science assessment at the university level [12], with a proposal to also cover computer science assessment in schools [13]. This work will present two tools: 1) Quiz Maker for the creation of quizzes, and 2) Quizly for assessing and automatically grading exercises done through the visual language App Inventor [14], with its extension into an assessment platform able to manage students and classes, administer and propose formative, summative and informal tests and to be able to track user progress as well as the question solving process. The work is organized as follows: Section 2 reviews the state of the art and is organized into three main subsections: Computer Science Curricula, Visual Languages, and Assessment and Assessment Tools; Section 3

presents the assessment tools Quiz Maker and Quizly; Section 4 describes the extensions and functionalities of the entire assessment platform; Section 5 describes the logging mechanism developed in the platform, which allows for tracking user activities and the early detection of gaps in the student's learning path. The logging activities are extended to the student's solution development process by allowing them to record the entire process of assembling stacks of blocks in order to monitor the solution; finally, Section 6 draws some conclusions and presents further developments.

II. State of the art

A. Curriculum development

In 2013 the United Kingdom's Department of Education released a mandatory computer science curriculum, beginning in the 2014 scholastic year and starting from the first year of school at Key Stage 1 up to Key Stage 4. Since this regulation was passed, national associations such as Computing At School (CAS) has put forward a set of guidelines for primary [15] and secondary teachers [16]. The curriculum guidelines have been organized into two progression pathways: the first concerns topics, namely: Algorithms, Programming & Development, Data & Data Representation, Hardware & Processing, Communications & Networks, and finally Information Technology with the topics arranged by difficulty level. The second progression pathway is for strands: Computer Science, Information Technology and Digital Literacy. During the first three Key Stages, the focus of the teaching and learning activities should be on the computational thinking aspects and at Key Stage 4 students can choose between the Computer Science and the Information Technology strands. The CAS network has developed a QuickStart Computing Guide for primary and secondary teachers [17] [18], which can be found at http://www.quickstartcomputing.org/. This is a general curriculum suited for all students and does not consider any specialized program of study focusing on computer science.

In the USA the three main curricula are organized around the Computer Science Principles framework [19], the Computer Science Teacher Association (CSTA) curriculum, which is currently under review [20], and the Computer Science Equity Alliance curriculum [21] as well as other state level curricula. The AP Computer Science Principles Curriculum Framework is equivalent to a first semester introductory computer science course at the college level. It is organized around seven computational thinking practices: Connecting, Computing, Creating Computational Artifacts, Abstracting, Analyzing Problems and Artifacts, Communication and Collaboration; it represent the competences that students must have. The areas of the course are organized around seven big ideas: Creativity, Abstraction, Data and Information, Algorithms, Programming, The Internet, and Global Impact. Each big idea is associated with a set of essential questions useful for guiding students in finding connections to the content of the big ideas; it contains enduring understanding which specifies core concepts that have to be mastered by the students. Each enduring understanding is aligned with Learning Objectives (LO) that provide a more detailed articulation of what the students should be able to do. These objectives integrate computational thinking practice with specific content. Next to each LO there is a list of essential knowledge statements which specify facts or content which has to be mastered by the students. The assessment is done through two performance tasks: Explore – Implications of computing innovations; Create applications from ideas. An end of course assessment with a time restriction completes the assessment process. The performance task takes longer but allows students to demonstrate a diverse skill set, beginning with creativity and should engage them in real work and motivating tasks which may be socially relevant. Students can present their work in an online portfolio, such as a website. The general framework has been implemented into several courses, notably the Mobile Computer Science Principle course both for students as well as professional development for teachers [22-24], and the Beauty and Joy of Computing [25]. For an overview of the course approach the reader can refer to [26 – 28].

The CSTA curriculum is organized into three levels: the first for grades K1-K6, the second for grades K6-K9 and the third for grades K9-K12. The latter is divided into three courses: Computer Science in the Modern World (K9-K10), Computer Science Concepts and Practices (K10-K11), and Topics in Computer Science (K11-K12), with each course intensifying in depth and material. Each level is organized into five strands: Computational Thinking, Collaboration, Computing Practice and Programming, Computers and Communication Devices, and finally Community, Global and Ethical Impacts. The curriculum describes the specific computer science concepts and skills associated with each strand that has to be mastered by the student at each level.

The Computer Science Equity Alliance curriculum was developed by combining computer science content and computational practice and has been mapped to several standards (http://pact.sri.com/index.html). The curriculum is shaped around four main elements: Curricular Material, Professional Development, Assessment (forthcoming) and Local Policy Support, with the aim of teaching the creative, collaborative, interdisciplinary and problem solving nature of computing. It is organized into six units: Human Computer Interaction, Problem Solving, Web Design, Introduction to Programming, Computing and Data Analysis, and Robotics. Each unit has daily lesson plans with student activities and teaching strategies. The lessons are developed to reinforce the three main themes of the curriculum: the creative nature of computing, technology as a tool for solving problems, and the relevance of computer science and its impact on society. For a wider overview of the curricula, the reader can refer to two recent special issues [29-30] and a review about high school curricula [31] as well as an international effort on reforming higher education curricula in Computer Science and Information Technology [32] to grasp the worldwide push to introduce computer science as early as possible and along as many learning paths as possible..

B. Learning environments for initial programming

Visual languages have become a common choice for an initial programming environment suitable for an introductory course in computer science. Examples of these include: Scratch [33] which is suitable for an introductory approach to

computing and for primary and lower middle school students. App Inventor [34] which allows the student to build apps for mobile devices and enables him/her to manage a rich set of components that respond to events. It is suited for middle and high school students. BYOB/SNAP [35] which builds on top of Scratch and allows students to practice with an even richer set of computer science concepts such as "procedures as first class data, from the Scheme language". Enchanting (http://enchanting.robotclub.ab.ca/tiki-index.php) which is built on top of BYOB/SNAP. It offers a programming environment for LEGO Mindstorms robots. Robotics has been seen as a way to introduce and engage students in STEM education [36] with its rich set of initiatives, such as the collaborative robotics programming competition [37] performed either by a visual language automatically translated in C or directly in the C language. Flip [38] which combines a visual editor with a natural version of the scripts, thus allowing students to use natural language as a means of improving code comprehension as well as their computational communication skills.

C. Assessment and assessment tools

In this rapidly changing environment of competency frameworks, which focus on key competencies that students have to acquire in order to succeed in modern working environments, and curricula that are issued worldwide there is an urgent need for good assessment practices aligned with the new frameworks and curricula. Assessments should be seen as powerful learning tools in order to teach students the skills needed to prepare their questions [39] and to guide them in asking the right questions during their lessons [40]. Beside this it is necessary to have assessment tools which allow for an accurate and quick verification of student performance with immediate feedback, thus fostering self-reflection. Several useful tools exist for visual block languages, such as:

- Scrape [41] and the related set of tools that allow for the analysis of single/multiple Scratch projects with statistics such as the number of blocks or the types of computational constructs used.
- Scratch Explorer [42], a tool that depicts relationships between different parts of the program.
- Hairball [43] which is a system that can be useful to students and teachers. For students it can point out potential errors or unsafe practices while allowing teachers to inspect Scratch programs.
- Dr. Scratch [44][45] which is a free/open source web tool that allows for the analysis of Scratch projects by automatically assigning a Computational Thinking score in terms of abstraction, logical thinking, synchronization, parallelism, flow control, user interactivity and data representation. It can be used by students for self-assessment as well as by teachers.
- REACT (Real-Time Evaluation and Assessment of Computational Thinking) [46] is a tool that provides a teacher with a sortable dashboard showing the characters used as well as the semantic meaning behind what students have programmed in AgentSheets [47].

The basic approach of these tools is an insightful static analysis of the type of blocks used, an analysis that leads to the detection of bad programming practice, potential problems, mastery of computational thinking concepts by analyzing the type and number of blocks used, something that offers teachers a dashboard for summarizing students' projects even in groups. Our work will describe a powerful platform that allows for assessment by means of live coding using the App Inventor visual language which also offers an automatic check of the solution, giving freedom for a rich problem space thus engaging the students in different types of problems, from ordering a set of blocks to producing a complete solution to a problem using all the blocks available on App Inventor and by posing questions in different types of formats..

III. QUIZ MAKER AND QUIZLY

Quiz Maker is a web-based tool that allows teachers and students to create questions based on the App Inventor language. Its companion tool, Quizly, allows teachers to assess their students through formative, summative and informal means while allowing students to play with challenging questions using a live coding platform which provides hints and real-time automatic feedback. The tool is built upon Blocky (https://developers.google.com/blockly/), a library for building visual programming editors. The two apps are available at https://github.com/ram8647/quizly. The interface of Quiz Maker is shown in Fig. 1. The tool allows for the creation of questions using the following workflow, as is clearly described in the Quiz Maker site:

- Select the type of quiz. Fill in the name of the quiz and a brief description. Compose the quiz question, including HTML and hints, then select the built-in and App Inventor component blocks, i.e., those needed to solve the problem.
- The Quiz tab sets up the starting blocks.
- The Solution tab constructs the solution to the problem.
- The Preview tab allows the participant to try the quiz.
- The JSON tab generates the quiz as a JSON string, which is saved in the back end database.

The types of questions that can be generated range from building a solution with App Inventor to writing the answer to the given question in a text box. By providing clear and careful instructions in the questions, the solution provided by the students can be automatically compared to the reference solution, thereby providing immediate feedback and offering a means both for self-learning and automated assessment. The Quizly interface is shown in Fig. 2. The quiz generated using Make Quiz can be executed with Quizly. The student has the option to choose a quiz. The available quizzes, which are within the assessment platform, are assigned by the teacher to the student, a group of students, or to the class. When the student chooses the quiz, its description appears. The question can be associated with a tutorial related to the problem at hand. The quiz generated using Make Quiz can be executed with Quizly. The student has the option to choose a quiz. The available quizzes, which are within the assessment platform,

are assigned by the teacher to the student, a group of students, or to the class. When the student chooses the quiz, its description appears. The question can be associated with a tutorial related to the problem at hand.

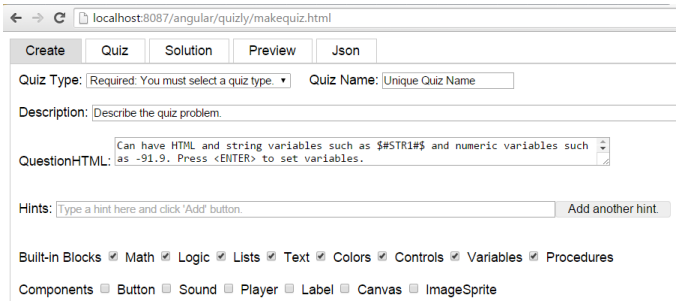


Fig. 1. Quiz Maker interface

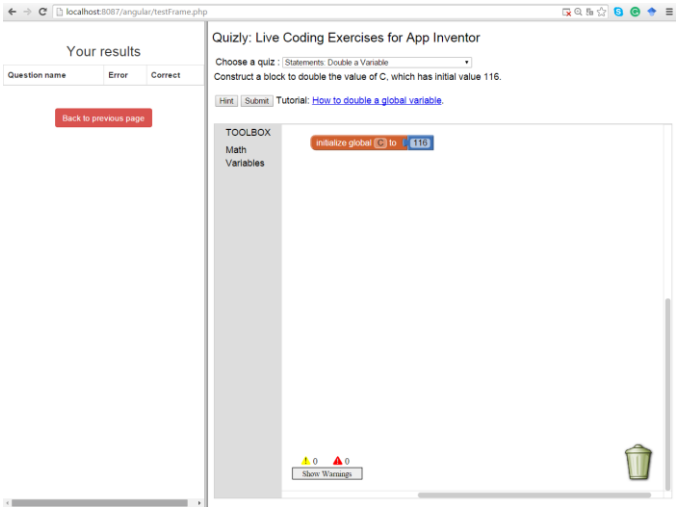
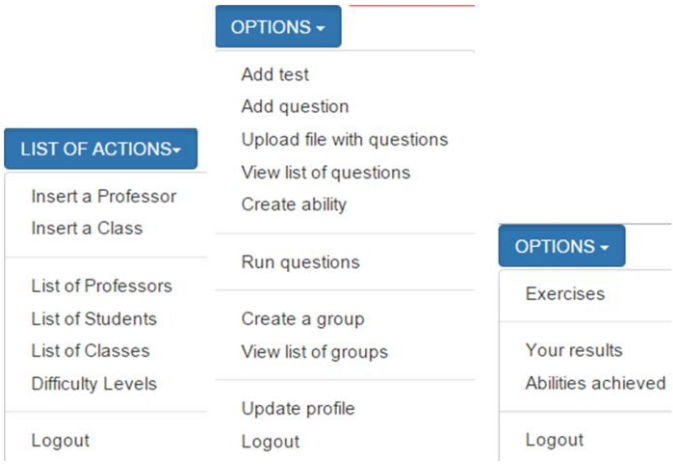


Fig. 2. The Quizly interface embedded in the Assessment platform.

The student has the option to request hints that should be as much as possible inquiring questions for scaffolding student's discovery of the solutions. When the student is satisfied with the answer, he/she can submit it and obtain an immediate response which informs him/her whether or not the answer is correct. This is done by comparing the student's solution with the solution prepared by the teacher when the question was first created. The Quizly interface shown on the left pane of Fig. 2 has been embedded into an assessment platform, which is described in the next section. The left pane includes other information, summarizing her learning journey by displaying the list of questions with an indication of the number of errors and correct solutions submitted for each question.

IV. The Assessment Platform

The two tools, Quiz Maker and Quizly, are embedded into an assessment platform which was developed with a three tier architecture using AngularJs (https://angularjs.org/) as a client scripting language, PHP as a server side scripting language and MYSQL as the underlying database. The platform allows teachers to manage the assessment activities of their classes and to monitor their students' progress. The platform allows for three types of users: administrators, teachers and students. The main functionalities available to them are shown in Fig. 3. The whole platform will be available on GitHub.



a) b) c)

Fig. 3. Main functionalities of the assessment platform available to administrators (a), teachers (b) and students (c).

The administrators have the ability to create a class and to create teachers that are associated with a class; they can also manage the level of difficulty of the questions. Since the assessment platform manages multiple questions and multiple classes, when the teacher assembles a test he/she can assign a certain level of difficulty to each question depending on the level of the class. Each level of difficulty has an associated weight given to the number of errors and the number of hints. These weights are used to automatically assign a grade to the final answer submitted by each student. The level of difficulty is assigned by the teacher to each question assembled in a test created for a class. The teacher has the ability to manage:

- new groups inside the class, e.g. reinforcement or advanced groups within the same class.
- new questions, even uploading their JSON description from a text file. The teacher can try the questions inside the Quizly application using "Run questions."
- tests composed of a set of questions. The test could be used as a learning tool, e.g. guiding the student in the construction of a game by designing and developing each stack of blocks inside each question. As a further avenue of research we envisage using the platform to create tests, seen as fun and engaging activities where the students, by solving each question, proceed in the development of a small application. The tutorial that is already linked to each question can offer scaffolding material. Since Quiz Maker and Quizly allow for the use of both the control structures and the components of App Inventor, the teacher has the complete freedom to build questions and tests that guide the student in developing a reach set of activities that could also be engaging. The test can be assigned to a class or to a group of students.
- view statistics either for each single student or for an entire class. Statistics for class results range from max,

min, average as well as standard deviation of the grades, including graphics for statistics related to the questions inside the test, e.g., the simpler, more difficult, or more time-consuming questions, with the max. number of suggestions and detailed statistical results for each question. Similar statistics are available for each student. The platform automatically alerts teachers when the activities of a student fall below her/his average, e.g., frequency of test completion.

- create abilities. The abilities can be created on the basis of an extendible taxonomy, such as [48 – 50] or 21st century skills, or skills associated with progression pathways. The abilities can be associated with each question within each test assigned to a given class and can be arranged in various levels (from beginner to master). Further development is necessary.

The student can answer questions and take tests assigned to her, view her results and statistics as well as check her abilities.

V. Logging user activities

The logging mechanism of the assessment platform has been extended in order to log the activities related to the block movements, which will allow for insight into the student's solution building process. In particular, the platform allows the following activities to be logged: inserting a block in the workspace, canceling a block from the workspace, connecting or disconnecting two blocks as well as the activities of requesting a hint and submitting an answer. All the activities are logged with a time indication so that inference on the student's solution building process can be made on the basis of objective parameters such as time, number of trials and so on. The number of clicks and block connects and disconnects with the related statistical metric can be the starting point of these analyses. An in-depth analysis of the different types of blocks used in developing the solution requires a careful design and will be considered in further research. All the collected information is permanently stored in the database.

VI. Conclusion and further developments

In this paper we have presented an assessment platform based on a live coding web-based tool for App Inventor with real-time feedback and automatic grading. The platform allows teachers to manage classes and groups of students, create questions and tests, assign them to students, track their performance, detect learning paths as well as the progress of the course and to log the activities related to the solution construction in order to gain insight into the solution building process, thereby detecting user difficulties. As future work we plan to extend the types of questions [51] that can be asked and used with the platform and their format, enlarge the set of blocks that can be used in the assessment platform by using the building block capabilities of Blocky, which allows for constructs from other languages such as SQL [52]. This can be further developed by designing blocks and environments which will allow for the assessment of design capabilities related to Flowchart [53], an Entity Relationship model or a Unified Modeling Language diagram such as Class Diagram. The basic graphics element of each design language can be used as a basic building block which can be snapped together in order to construct the correct design. The platform can also be extended, allowing digital ink [54] in the assessment process and in the annotation of both the questions and the grading/commenting process by the teacher. Furthermore the assessment platform is currently being validated by courses both at the school and at the university levels [55] as well as in professional development courses for teachers, such as the mobile computer science principles course (http://mobile-csp.org/). The questions have been used to assess the enduring understanding, learning objectives and essential knowledge described in [19]. Ways on how to integrate the questions and activities developed inside Quizly in the App Inventor environment will be explored.

Acknowledgment

This work was supported in part by National Science Foundation grants CNS-1240841 and TUES-1225976.

References

[1] Department of Education. National curriculum in England: computing programmes of study. September 2013 Retrieved on-line from https://www.gov.uk/government/publications/national-curriculum-in-england-computing-programmes-of-study

[2] F. Kalelioğlu, "A new way of teaching programming skills to K-12 students: Code. org". *Comp. in Human Behavior*, *52*, 200-210, 2015.

[3] M. Binkley, O. Erstad, J. Herman, S. Raizen, M. Ripley, M. Miller-Ricci, and M. Rumble, "Defining twenty-first century skills", In *Assessment and teaching of 21st century skills, 2012* (pp. 17-66). Springer Netherlands.

[4] International Commission on Education for the Twenty-first Century, & Delors, J. (1996). Learning, the Treasure Within: Report to UNESCO of the International Commission on Education for the Twenty-First Century. UNESCO.

[5] J. Gordon, G. Halász, M. Krawczyk, T. Leney, A. Michel, D. Pepper, ... and J. Wiśniewski, "Key competences in Europe: opening doors for lifelong learners across the school curriculum and teacher education", *CASE network Reports*, (87) 2009.

[6] K. Ananiadou, and M. Claro, "21st century skills and competences for new millennium learners in OECD countries.

[7] P. Curzon, M. Dorling, T. Ng, C. Selby, J. Woollard. Developing computational thinking in the classroom: a framework (2014).

[8] S. Y. Lye, and J. H. L. Koh, Review on teaching and learning of computational thinking through programming: What is next for K-12?. *Computers in Human Behavior, 41*, pp. 51-61, 2014.

[9] S. Grover, R. Pea. Computational Thinking in K–12 A Review of the State of the Field. *Educational Researcher, 42*(1), pp. 38-43.

[10] J. M. Wing, Computational thinking. *Communications of the ACM, 49*(3), pp. 33-35, 2006.

[11] A. Yadav, D. Burkhart, D. Moix, E. Snow, P. Bandaru, and L. Clayborn, Sowing the Seeds: A Landscape Study on Assessment in Secondary Computer Science Education. CSTA annual conference 2015.

[12] K. Sanders, M. Ahmadzadeh, T. Clear, S. H. Edwards, M. Goldweber, C. Johnson, ... and J. Spacco, The Canterbury questionbank: Building a repository of multiple-choice CS1 and CS2 questions. In *Proceedings of the ITiCSE working group reports conference on Innovation and technology in computer science education-working group reports*, pp. 33-52. ACM, 2013

[13] D. Giordano, F. Maiorana, R. Morelli, "A repository for high school computer science questions, visual assessment tools and metadata annotations" Working group proposal submitted at the 20th Conference

on Innovation and Technology in Computer Science Education (ITICSE 2015).

[14] B. Magnuson, Building blocks for mobile games: a multiplayer framework for App inventor for Android (Doctoral dissertation, Massachusetts Institute of Technology), 2010.

[15] Computing at School: Computing in the national curriculum. A guide for primary teachers

[16] Computing at School: Computing in the national curriculum. A guide for secondary teachers

[17] Computing at School: QuickStart Computing: A CPD toolkit for primary teachers.

[18] Computing at School: QuickStart Computing: A CPD toolkit for secondary teachers.

[19] College Board. AP Computer Science Principles Curriculum Framework

[20] The CSTA Standards Task Force. K–12 Computer Science Standards Revised 2011

[21] Computer Science Equity Alliance, 2013. *Exploring Computer Science*

[22] R. Morelli, C., Uche, P. Lake, and L. Baldwin, Analyzing Year One of a CS Principles PD Project. In *Proceedings of the 46th ACM technical symposium on Computer science education* . ACM, 2015.

[23] R. Morelli, D. Wolber, J. Rosato, C. Uche, and P.Lake, Mobile computer science principles: a professional development sampler for teachers. In *Proceedings of the 45th ACM technical symposium on Computer science education* (pp. 750-750). ACM, 2014.

[24] R. Morelli, D. Wolber, S. Pokress, F. Turbak, F. Martin, Teaching the CS principles curriculum with App Inventor. In *Proceeding of the 44th ACM technical symposium on Computer science education* (pp. 762-762). ACM, 2013.

[25] D. Garcia, J. Campbell, R. Dovi, and C. Horstmann, Rediscovering the passion, beauty, joy, and awe: making computing fun again, part 7. In *Proceedings of the 45th ACM technical symposium on Computer science education* (pp. 273-274). ACM, 2014.

[26] D. Garcia, O. Astrachan, B. Brown, J. Gray, C. Lin, B. Beth, ... qnd N. Sridhar, Computer Science Principles Curricula: On-the-ground; adoptable; adaptable; approaches to teaching. In Proceedings of the 46th ACM Technical Symposium on Computer Science Education (pp. 176-177). ACM, 2015.

[27] P. T. Tymann, F. P. Trees, L. Wainwright, R. Kick, S. Czajka, A. Kuemmel, and L. Diaz, Achieving a shared goal with AP Computer Science A and AP Computer Science Principles. In Proceedings of the 46th ACM Technical Symposium on Computer Science Education (pp. 436-437). ACM, 2015.

[28] O. Astrachan, R. Morelli, G. Chapman, and J. Gray, Scaling High School Computer Science: Exploring Computer Science and Computer Science Principles. In *Proceedings of the 46th ACM Technical Symposium on Computer Science Education* (pp. 593-594). ACM, 2015.

[29] P. Hubwieser, M. Armoni, and M. N. Giannakos, How to Implement Rigorous Computer Science Education in K-12 Schools? Some Answers and Many Questions. ACM Transactions on Computing Education (TOCE), 15(2), 5, 2015.

[30] J. Tenenberg, and R. McCartney, Editorial: Computing Education in (K-12) Schools from a Cross-National Perspective. ACM Transactions on Computing Education (TOCE), 14(2), 6, 2014.

[31] V. Garneli, M. N. Giannakos, and K. Chorianopoulos, Computing Education in K-12 Schools: A Review of the Literature. EDUCON 2015. In Global Engineering Education Conference (EDUCON), 2014 IEEE (pp. 556-563). IEEE

[32] J. Impagliazzo, Curriculum Design for Computer Engineering and Information Technology. Workshop at the Global Engineering Education Conference (EDUCON), 2015.

[33] J. Maloney, M. Resnick, N. Rusk, B. Silverman, and E. Eastmond, E. The scratch programming language and environment. *ACM Transactions on Computing Education (TOCE)*, *10*(4), 16, 2010.

[34] F. Turbak, M. Sherman, F. Martin, D. Wolber, and S. C. Pokress, Events-first programming in APP inventor. *Journal of Computing Sciences in Colleges*, *29*(6), 81-89, 2014.

[35] B. Harvey, J. Mönig, Bringing “no ceiling” to scratch: Can one language serve kids and computer scientists. *Proc. Constructionism*, 2010.

[36] D. Catlin, A. P. Csizmadia, J. G. OMeara, J. G., and S. Younie,. Using Educational Robotics Research to Transform the Classroom.

[37] S. Nag, J. G. Katz, and A. Saenz-Otero, Collaborative gaming and competition for CS-STEM education using SPHERES Zero Robotics. *Acta astronautica*, *83*, 145-174, 2013.

[38] K. Howland, J. Good, Learning to communicate computationally with Flip: A bi-modal programming language for game creation. Computers & Education, 80, 224-240, 2015.

[39] P. Denny, Generating Practice Questions as a Preparation Strategy for Introductory Programming Exams. In *Proceedings of the 46th ACM Technical Symposium on Computer Science Education* (pp. 278-283). ACM, 2015.

[40] S. Mishra, S. Iyer, Question-Posing strategies used by students for exploring Data Structures. In *Proceedings of the 2015 ACM Conference on Innovation and Technology in Computer Science Education* (pp. 171-176). ACM, 2015.

[41] U. Wolz, C. Hallberg, and B. Taylor, Scrape: A tool for visualizing the code of Scratch programs. In *Poster presented at the 42nd ACM Tech. Symposium on Computer Science Education, Dallas, TX*, 2011.

[42] Y. Zhang, S. Surisetty, and C. Scaffidi, Assisting comprehension of animation programs through interactive code visualization. *Journal of Visual Languages & Computing*, *24*(5), 313-326, 2013.

[43] B. Boe, C. Hill, M. Len, G. Dreschler, P. Conrad, and D. Franklin, Hairball: Lint-inspired static analysis of scratch projects. In Proceeding of the 44th ACM technical symposium on Computer science education (pp. 215-220). ACM, 2013.

[44] J. Moreno, and G. Robles, Automatic detection of bad programming habits in scratch: A preliminary study. In Frontiers in Education Conference (FIE), 2014 IEEE (pp. 1-4). IEEE, 2014.

[45] J. Moreno, and G. Robles, Analyze your Scratch projects with Dr. Scratch and assess your Computational Thinking skills. Scratch Conference 2015, 2015.

[46] K. H. Koh, A. Basawapatna, H. Nickerson, and A. Repenning, Real time assessment of computational thinking. In *Visual Languages and Human-Centric Computing (VL/HCC), 2014 IEEE Symposium on* (pp. 49-52). IEEE, 2014..

[47] A. Repenning, Agentsheets: a tool for building domain-oriented visual programming environments. In Proceedings of the INTERACT'93 and CHI'93 conference on Human factors in computing systems (pp. 142-143). ACM, 1993.

[48] L. W. Anderson, D. R. Krathwohl, and B. S. Bloom, A taxonomy for learning, teaching, and assessing: A revision of Bloom's taxonomy of educational objectives. Allyn & Bacon, 2001.

[49] S. E. Dreyfus, and H. L. Dreyfus, A five-stage model of the mental activities involved in directed skill acquisition (No. ORC-80-2). California Univ Berkeley Operations Research Center, 1980.

[50] J. B. Biggs, and K. F. Collis, Evaluating the quality of learning: The SOLO taxonomy (Structure of the Observed Learning Outcome). Academic Press, 2014.

[51] O. Hazzan, T. Lapidot, and N. Ragonis, Types of Questions in Computer Science Education. In *Guide to Teaching Computer Science* (pp. 143-163). Springer London. Second Edition, 2014.

[52] Y. N. Silva, and J. Chon, Dbsnap: Learning database queries by snapping blocks. In Proceedings of the 46th ACM Technical Symposium on Computer Science Education (pp. 179-184). ACM, 2015.

[53] D. Giordano, and F. Maiorana, Teaching Algorithms: Visual Language vs Flowchart vs Textual Language. In Global Engineering Education Conference (EDUCON), 2015 IEEE (pp. 556-563). IEEE, 2015 .

[54] D. Giordano, F. Maiorana, and L. M. Vaccalluzzo, Integrating Digital Ink and Paper in a Learning Content Management System for Rapid Learner Assessemnt. In Pen-Based Learning Technologies, 2007. PLT 2007. First Int. Workshop on (pp. 1-6). IEEE, 2007.

[55] D. Giordano, F. Maiorana, Use of cutting edge educational tools for an initial programming course. In Global Engineering Education Conference (EDUCON), 2014 IEEE (pp. 556-563). IEEE, 2014.

Design of a Blocks-Based Environment for Introductory Programming in Python

Matthew Poole
School of Computing
University of Portsmouth, UK
Email: matthew.poole@port.ac.uk

***Abstract*—This paper details the design of a visual blocks-based tool for editing Python programs. Its purpose is to close the gap between programming using a simplified blocks-based language and textual programming in a mainstream language. As well as helping to guarantee the syntactic validity of programs, the tool aims to reduce the occurrence of run-time errors, a source of learner frustration with dynamic languages, by ensuring that constructed programs will remain well-typed during execution. The design promotes understanding of how data types are used in the language by representing them using colors: each expression block is colored according to its type, and each unfilled hole contains colors which indicate valid argument types. Connected blocks preserve conventional use of whitespace, demonstrating good practice for novice programmers.**

I. Introduction

Visual programming systems such as Alice [1], Blockly [2] and Scratch [3] are attractive environments in which to learn to program. They each provide a restricted programming language and an accessible editing environment of jigsaw piece program elements manipulated through drag-and-drop interactions.

Many learners will need to move onto traditional textual languages such as Java, Python or JavaScript. Other students' first taste of programming will be in such a textual language. Whatever their prior programming experience, these novice programmers need to cope with the combination of the complexity of a mainstream language's syntax and semantics and the absence of any guidance provided by blocks-based editing.

Python is an increasingly popular language in introductory programming courses due largely to its comparative simplicity. This paper describes a design for a blocks-based environment for Python 3, with the aims of providing the advantages of visual editing to support the learning of programming in a mainstream textual language.

This paper is structured as follows. In the next section we detail the aims and principles which have guided the design. Section III describes the design of the blocks themselves, and Section IV demonstrates how programs are constructed. Section V discusses future work and concludes the paper.

II. Design Principles

The target users of the proposed system are high school or higher education students who are beginning to learn programming in Python, either with no previous programming experience, or are transitioning from having used a blocks-based programming environment such as Scratch. The intention is for the tool to be used as a first environment for editing Python programs, before moving on to a text-only IDE such as IDLE. It could also be used as a fallback for students who are struggling with the structure of the Python language when faced with a text editor. The design doesn't aim to support the whole Python 3 language; instead it focusses on those features commonly taught in the first phase of an introductory course. The current paper describes support for constructing programs featuring expressions, assignments, input-output and control structures. A future paper will consider function definitions, module imports and user-defined classes, in addition to giving implementation details.

By their very nature, an aim shared by block environments is to reduce "syntax overload", and to guide the user in constructing syntactically legal programs. This is also a primary aim of the proposed software. Learners of mainstream dynamically typed languages also know that a major source of frustration is the frequent occurrence of run-time errors (one disadvantage of their use compared with statically-typed languages such as Java). Students are constantly faced with "TypeError" messages (such as when a program attempts to add a float and a string or to index an integer). Block languages such as Scratch tend to avoid run-time errors, but do so by means of simpler, more forgiving and less dynamic type systems than Python's.

The proposed software aims to minimize such type-related run-type errors, mainly by enforcing static typing during the construction of programs. This is achieved by requiring that (i) variables' types are fixed ("declared") when they are created within the block palette, and (ii) valid argument types of all operator and function blocks are enforced. Furthermore, the types of all operations, arguments and expressions should be made explicit to the user to help develop their understanding of how types work within their programs. This latter point tends to be de-emphasized in typical block languages.

Programs in textual languages should not only be correct, but should also be readable; readability necessitates adherence to accepted conventions regarding whitespace. The use of indentation in Python helps to enforce readability, but correct whitespace between neighboring lexical elements (e.g. around operators and after commas) is also important. A block editor which honors correct whitespace would not only demonstrate good code layout to learners, but would also help provide a more seamless transition to textual code editing. The use of connector shapes in languages such as Blockly and Scratch

978-1-4673-8367-7/15/$31.00 ©2015 IEEE

takes up space on the left/right sides of blocks and thus prevents conventional use of whitespace.

Some recent related work on editing textual languages using blocks includes Tiled Grace [4] and Pencil Code [5], [6]. Tiled Grace supports the pedagogical language Grace, whilst Pencil Code currently works for JavaScript and CoffeeScript. In both systems the user can edit a program using blocks and then switch to a well-formatted text view, and in Pencil Code whitespace is preserved in the block view. Colors are used in both systems, but somewhat arbitrarily and not for types. Expression types are generally not explicitly indicated and restrictions on what constitutes a legal block argument often only becomes apparent on attempting a drag-and-drop operation, if at all.

III. Block Design

The presented design supports those core Python types which might be used in a typical introductory programming course. These types include integers, floats, strings and Booleans, and lists of these basic types. Ranges are supported for their use in `for` loops. Tuples and dictionaries are not currently supported in this design. It is envisaged that all the most commonly used built-in operators, functions and methods would be available for the user in a block palette with categories for statements, variables and for each supported type. Assignments, conditionals and loop statements, as well as comment lines would be present in the statement palette. This initial design does not describe module imports or function definitions, but clearly inclusion of these features would be desirable in a subsequent version.

Each type is assigned a unique color, and blocks are colored according to type. Statement blocks are considered uncolored, and are chained together vertically using a single block connector shape.

A. Type colors and literals

Colors for the four basic types are illustrated in Fig. 1 using blocks for literal values. The numeric and string blocks are editable to allow the user change the literal value within the same type. The absence of space around the values, and the lack of rounded corners, is deliberate due to the desire for programs to adhere to conventions concerning whitespace.

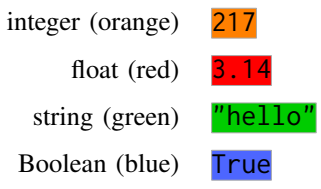


Fig. 1. Basic type colors and literal values

B. List types

Our design also allows for homogeneous lists of the basic types. We use blocks with three colored stripes to represent list types, hinting at the notion of multiple values. Example list variable blocks are illustrated in Fig. 2. The choice to restrict the types of lists simplifies the color representation required, aids static typing, and also helps to enforce good programming practice.

Fig. 2. Two list variable blocks: a list of strings and a list of floats

C. Function and operator blocks

All blocks except those for literal values and variables will include holes for argument blocks. Blocks for operators and functions which give result values are colored according to the result type. Furthermore, to indicate clearly the valid argument types for these blocks, all unfilled holes are marked with small "type indicators" showing all acceptable argument types.

Example function and operator blocks are shown in Fig. 3. The `round` block is colored orange since it gives an integer result. The argument hole includes a single red type indicator, denoting that the argument should be a float. Any attempt at dropping a non-red block into this hole will be rejected. The `float` block is colored red (calling `float` results in an float value); the green and orange type indicators state that the argument should be either a string or an integer. Finally, we can see that the division operator '`/`' takes two numeric values (integers or floats) and produces a float.

Notice that operator blocks cannot extend horizontally past their arguments if we want programs to adhere to whitespace conventions. We therefore extend blocks downwards in order to make clear the type, extent and structure of expressions.

Fig. 3. Blocks for the functions `round` and `float`, and the float division operator '`/`'

The result type of many built-in functions and operators depends upon the argument types. Examples include functions such as `abs` (absolute value) and overloaded operators such '`+`' and '`*`'. Whilst `abs` operates only on numeric data, the symbols '`+`' and '`*`' are also used for string and list operations. We choose to have three distinct pairs of blocks (i.e. numerical, string and list) for these operators since they differ in nature for the different types. Fig. 4 shows the blocks for `abs` and the overloaded numerical operator '`+`'. With as yet unfilled holes, the result types of these blocks are not fully determined—each could return either an integer or a float—and they are therefore colored both orange and red, using a chequered pattern.

Fig. 4. Blocks for the function `abs` and the overloaded operator '`+`'

As we will see in Section IV, the colors of blocks and the visibility of type indicators is dynamic: as argument blocks are added, a multicolored block may become monocolored, and some type indicators may disappear from argument holes.

Some operations (e.g. the equality operator '`==`') accept arguments of several types. Rather than include a large number of type indicators in such a case, we use a single color, gray, to represent any of the four basic types, and striped gray to stand for any list type. Fig. 5 shows the block for '`==`', and also

blocks for list construction and list indexing, which together serve to illustrate the use of gray.

We see that both arguments to '==' can be of any basic or list type, and the result type is Boolean. List construction is considered an operator; an argument of a basic type will result in a list of that type. (The small triangle indicates a menu allowing the addition of extra arguments.) Indexing a list with an integer gives a value of the type of data stored in the list.

Fig. 5. Blocks for the equality operator '==', for list construction and for list indexing

The result types of most of Python's commonly used functions and operators, including all those discussed above, are either fixed or determined from the types of their arguments. In some cases however, we need to know the *values* of arguments. The most obvious example is the `eval` function which evaluates the contents of a string argument (`eval("12")` is the integer 12 and `eval("1 + 2.1")` is the float 3.1). In introductory programming, `eval` is typically used to convert keyboard (string) inputs into numerical values. However, the use of the functions `int` and `float` is preferred since (i) their use encourages students to think more about their choice of types, and (ii) they are considered simpler and safer. We choose therefore not to include `eval`.

The power function `pow` (and operator '`**`'), which when empty is colored like the '+' operator in Fig. 4, cannot reasonably be omitted. The `pow` function returns a float if either of its arguments are floats. If both arguments are integers then the return type depends on the value of the second argument: a non-negative value results in an integer, and a negative value gives a float. We take a straightforward approach to typing `pow` blocks if the arguments are both integer blocks: the `pow` block is typed (colored) as an integer, and if the second argument is not an integer literal (i.e. not guaranteed to be non-negative) then the user is alerted that the type will be incorrect at runtime whenever the value of the second argument is negative. (This is unlikely to cause a problem in practice: the second argument will most often be a literal value such as 2.)

D. Statements

The proposed design includes blocks for assignment, conditional and loop statements. Blocks for functions and methods which do not return values (including `print` and `append`) are also considered as statements. Statement blocks do not have types, are all colored neutral gray-green, and are chained together vertically using a simple 'notch' connector.

When the user introduces a variable, which involves choosing a name and a type, an assignment block and appropriately colored variable block are added to the block palette; see Fig. 6. The new variable might be considered to be 'declared' with the given type (albeit externally from the program text), and the assignment block will only allow it to be assigned values of this type.

The `if` and `while` statement blocks take Boolean-valued conditions, as do blocks for the Boolean operators such as `or`.

Fig. 6. Assignment and variable blocks for an integer variable

This prevents the use of data of other types being interpreted as Boolean values which often leads to confusion for novice Python programmers (for example, the expression `x == 10 or 20` is legal Python and always evaluates to True since 20 is interpreted as True). These blocks are illustrated in Fig. 7. The block for the `if` statement includes a triangle menu indicator for adding `elif` and `else` clauses.

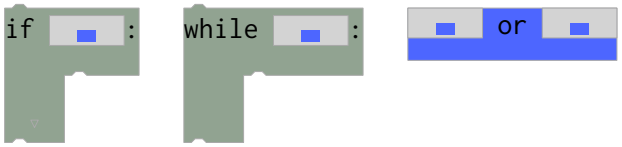


Fig. 7. Blocks for the `if` statement, `while` loop and Boolean `or` operator

The range data type and three `range` function blocks (for ranges with stop-values, and optional start- and step-values) are provided for use in `for` loops. The `for` loop block includes a selector to choose or create a loop variable of an appropriate type, and a hole with type indicators allowing a list, string or range argument block over which to iterate. Fig. 8 shows the `for` loop and the single-argument `range` block. Note the use of a new color, magenta, for the range type.

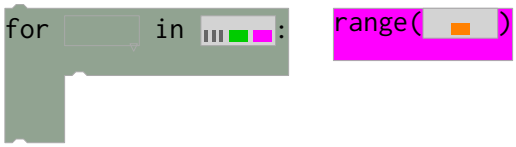


Fig. 8. Blocks for the `for` loop and `range` function

IV. Program Construction

Programs are constructed by dragging blocks from the block palette and dropping them into place within a workspace, as in Blockly. As a block is dragged, appropriately colored type indicators in close-by empty holes are highlighted to the user, and any attempted illegal drops are rejected.

Fig. 9 illustrates the interactions involved in the construction of a list indexing expression. Dropping the `rainfall` variable block (a list of floats) into the gray list indexing block causes the latter block to be recolored red (float), since indexing a list of floats will result in a float. Dropping the `day` integer variable block into the first argument hole of the '+' block causes no recoloring (the type of '+' will now depend on the second argument). The '+' block can be dropped into the list index hole (which has an integer type indicator), since the '+' block's colors include orange and so it can potentially be typed as an integer. When the drop is made, the '+' block is recolored to just orange, and the remaining red type indicator disappears to ensure that the second argument to '+' cannot be a float. Dropping the integer literal 1 block into this argument hole completes the expression. Note that, as subexpression blocks are added, the lower portions of operator and function

blocks are narrowed vertically. This ensures that the outermost expression block remains the same height and therefore that lines of code within a program are equally spaced. Blocks' heights would only need to increase if they were deeply nested.

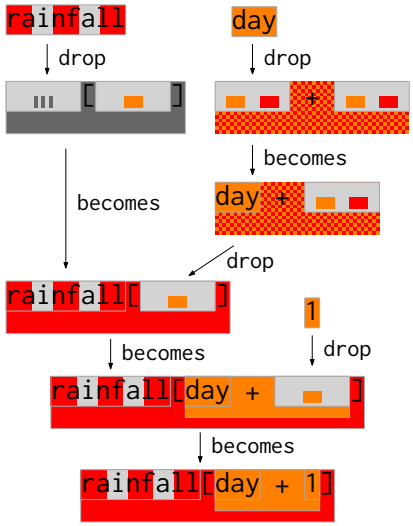


Fig. 9. Construction of an expression by drag-and-drop

One property of block languages is that parentheses are not needed to control the order of subexpression evaluation: the nesting of blocks suffices. For our purposes however, parentheses are required. Therefore, when an operator block is dropped into (i) an argument hole of a higher-precedence operator block, or (ii) the right-hand hole of an equal-precedence operator block, parentheses are automatically added around the dropped block; see Fig. 10.

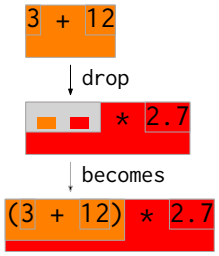


Fig. 10. Automatic insertion of parentheses

In the current design, a program is considered complete and valid by the editor if (i) it comprises a single set of connected blocks without empty holes, and (ii) each variable is guaranteed to have been initialised before its value is referenced. Fig. 11 gives an example program.

V. Discussion and Conclusion

We have presented a design for a blocks-based editor, inspired by visual editors such as Blockly and Scratch, and tailored towards introductory programming in Python. The presented design shares the goals of other blocks-based languages, allowing novice programming to focus on the fundamentals of programming rather than on the accidental complexities of the programming language used. In addition to ensuring syntactically correct programs, the presented design aims to

```
total = 0.0
count = 0
nextStr = input("Number (or end): ")
while nextStr != "end":
    next = float(nextStr)
    total = total + next
    count = count + 1
    nextStr = input("Number (or end): ")
average = total / count
print("The average is", average)
```

Fig. 11. A complete program

reduce the learner's frustration caused by frequent run-time errors inherent to dynamically typed languages. The design achieves this via declarations of variables' types (within the block palette) which enables types of expressions to be determined during program construction.

The uncomplicated use of color to represent types aims to promote comprehension of the language's type system. We have restricted the discussion to Python's built-in basic types and homogeneous lists. However, the use of color can be readily extended to encompass further types; homogeneous dictionaries can be represented using bi-colored blocks (for the key/value types), and simple class hierarchies might be colored using a single color for the root class, and the same color with various shading patterns added for the subclasses.

Future work will include implementation of the design, and its adaptation and extension to support program execution, function and class definitions, and imports of library modules.

References

[1] S. Cooper, W. Dann, and R. Pausch, "Teaching objects-first in introductory computer science," in *ACM SIGCSE Bulletin*, vol. 35, no. 1. ACM, 2003, pp. 191–195.

[2] "Blockly," http://code.google.com/p/blockly/, accessed 14 July 2015.

[3] M. Resnick, J. Maloney, A. Monroy-Hernández, N. Rusk, E. Eastmond, K. Brennan, A. Millner, E. Rosenbaum, J. Silver, B. Silverman *et al.*, "Scratch: programming for all," *Communications of the ACM*, vol. 52, no. 11, pp. 60–67, 2009.

[4] M. Homer and J. Noble, "Combining tiled and textual views of code," in *Software Visualization (VISSOFT), 2014 Second IEEE Working Conference on*. IEEE, 2014, pp. 1–10.

[5] D. Bau, D. A. Bau, M. Dawson, and C. Pickens, "Pencil code: block code for a text world," in *Proceedings of the 14th International Conference on Interaction Design and Children*. ACM, 2015, pp. 445–448.

[6] D. Bau, "Droplet, a blocks-based editor for text code," *Journal of Computing Sciences in Colleges*, vol. 30, no. 6, pp. 138–144, 2015.

Lambda in Blocks Languages: Lessons Learned

Brian Harvey
Computer Science Division
University of California
Berkeley, CA, USA
bh@cs.berkeley.edu

Jens Mönig
SAP SE
Göttingen, Germany
jens@moenig.org

***Abstract*—In designing BYOB and Snap*!*, we wanted to extend the Scratch idea of visual metaphors for control structures to include anonymous procedures and higher order functions. We describe the iterations in the design leading to the current "grey ring" notation.**

Keywords—blocks language, BYOB, Snap!, visual metaphor, control structure, lambda, higher order functions.

I. Introduction: Visual Metaphors

Scratch wasn't the first drag-and-drop programming language, but it pioneered the use of carefully designed visual metaphors for computer science ideas, such as snap-together Lego-brick-esque blocks (procedures) to form a script (thread), the C-shaped block for a control structure that wraps around a script, and the subtle upward arrow on those control blocks that implement looping (Figure 1).

In 2010 we collaborated on a project to extend the power of Scratch to more advanced computer science ideas, suitable for an undergraduate CS course. The first of these was the ability to create new blocks by, essentially, giving a script a name. This capability was important not only for the sake of modularity — Scratch kids would write scripts hundreds of lines long that no adult could read — but also because defining procedures opens the door to recursion, one of the central CS ideas. This first extension (done by Jens before Brian got involved) was therefore named Build Your Own Blocks, or BYOB.

In representing the script that defines a new block, we wanted to follow the Scratch principle of visual metaphors. Scratch has hat blocks that are used to specify an event that should trigger the running of a script (Figure 2).

In defining custom blocks, we wanted to convey the idea "this script should run when the new block is used," so we put the block's picture in a hat block above its script (Figure 3). (The BYOB block shapes were based on those of Scratch 1.4, so they look a little different from the new Scratch 2.0 blocks.)

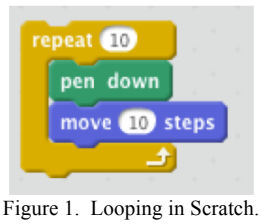


Figure 1. Looping in Scratch.

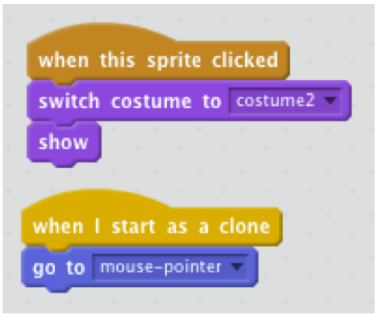


Figure 2. Event blocks in Scratch.

Scratch users already understand that a hat block means "when this, do that"; given that understanding, the meaning of the hat block in the script that defines a custom block is clear.

II. Anonymous Procedures

For the new introductory CS course then being developed at Berkeley, we weren't satisfied with recursion. We also wanted to teach another central CS idea: higher order functions — functions that take functions as arguments. To make this work, we needed to be able to distinguish between invoking a block (the usual thing in Scratch) and using the block itself as a value (e.g., as input to a higher order function). In CS terms, we needed to be able to encapsulate an expression as a procedure; that is, we needed anonymous procedures. Saying that another way, we needed the Lisp λ. For the introductory students, our emphasis is on making it easy to use higher order functions; at the same time, we want experts to be able to provide students with arbitrary control structures in libraries, and so we want to provide the full power of lexically scoped λ.

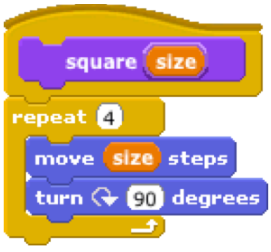


Figure 3. Custom blocks in BYOB.

978-1-4673-8367-7/15/$31.00 ©2015 IEEE

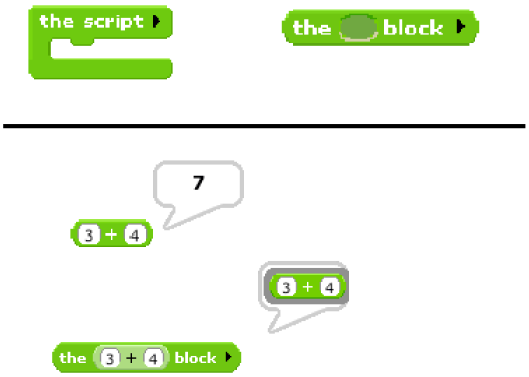


Figure 4. Initial λ representation in BYOB.

A. *Anonymous Procedures in BYOB*

In a blocks language, the obvious way to introduce a new mechanism is as a block. Two blocks, actually, because we needed one for commands/scripts and another for reporters/expressions (Figure 4.) The name `the block` dates from early in the design process, when we weren't sure whether a single command block should go in `the script`, as suggested by its shape, or in `the block`, as suggested by its name. The final decision was that `the block` accepts only reporters, so perhaps we should have renamed it "the reporter" or "the expression."

As shown in the example, even at this stage the visual metaphor of a block in a speech balloon is very powerful in conveying the idea of procedure as data. We drew a grey border around such blocks and scripts to further emphasize that the procedure was not being invoked in the current context.

Technically, in a lexically scoped language a procedure is not the same as the expression it encapsulates, because it also encapsulates an *environmant* that includes local variable bindings of the context in which this procedure was defined. In Scheme, the language we used as our model in this work, there is no standard output representation of a procedure, not even as text, let alone a visual metaphor. The grey border around an anonymous procedure is also a sort of fingers-crossed indication that there's more to a procedure than meets the eye. We try not to tell lies in the pictures we draw, but we don't always tell the entire story. This is true even in Scratch; consider the looping blocks in the two scripts in Figure 5. In the script on the left, the `repeat` block evaluates the variable `count` just once, before the loop begins, so the loop runs four

Figure 5. Special forms in Scratch.

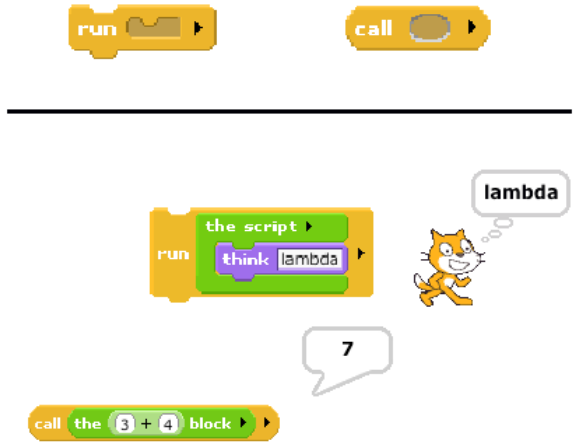


Figure 6. Applying an anonymous procedure.

times even though the value of `count` is 8 by the time the loop finishes. By contrast, the otherwise very similar `repeat until` block re-evaluates the `>` test each time through the loop. In Scheme terms, `repeat until` is a special form, whose Boolean argument is evaluated in normal order rather than the usual applicative order. (Actually, all C-shaped blocks are special forms, because their script argument isn't pre-evaluated.) Nothing in the appearance of the two blocks provides a metaphor for that difference

The point of encapsulating a procedure is that eventually it will be invoked. Since anonymous procedures may be inside another procedure, they don't have blocks in the global *palette* (the menu of blocks to the left of the scripting area). Instead we need generic blocks that take the desired procedure as an argument (Figure 6).

B. *Arguments to Anonymous Procedures*

So far, our examples have been of procedures with no arguments. But procedures are much more powerful when they embody a family of actions, controlled by argument values when they are invoked. Our lambda blocks can be expanded, as in Figure 7, to include formal parameters, which can be dragged into the expression being encapsulated. The `call` block can similarly be expanded to provide actual arguments.

At this point, the core intellectual work was done: We could create and invoke anonymous procedures with inputs, and use that capability to write higher order functions.

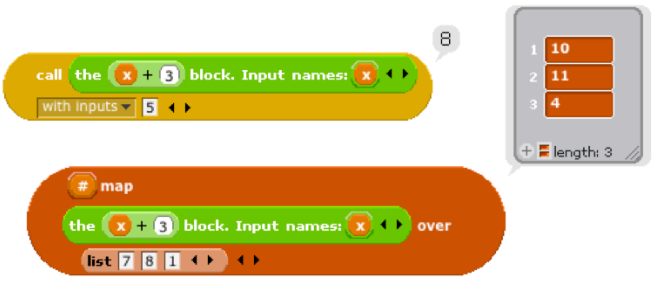


Figure 7, Explicit formal parameters.

Figure 8. `Call` block with grey ring input.

C. *Efforts to Improve the User Interface*

The lambda implementation as described so far had both pragmatic and pedagogic problems. Starting with the pragmatic, it was just too much trouble always to be searching for the lambda blocks at the bottom of the Operators palette. Also, especially for reporters, expressions using λ ended up being too wide to fit in the scripting area. Each of us proposed partial solutions. First, shift-clicking on any block or script displayed a one-item menu, namely "quote," which if clicked would surround the block or script with the appropriate λ block. This solution had the defect of not being something a user would automatically discover, but it was very convenient in avoiding the need to pull a block from Operators. But it didn't solve the width problem. The other solution was to take advantage of the fact that input (argument) slots in a block knew if they were expecting a procedure input. The idea was to reuse the grey rings around encapsulated procedures when displaying them as an *input* notation also. We changed the procedure-type input slots from being the same color as the block to being grey, and they could put a grey ring around any procedure dropped into them (Figure 8). This new notation was a dramatic improvement in the width problem. (You'll notice that the formal parameter is missing. This point is coming up in the next paragraph.) The trouble is that *sometimes* a user wanted to call a function that would provide the function input to the `call` block, and so wouldn't want that call encapsulated (Figure 9). We were therefore forced into what turned out to be a very difficult part of the interface for users to understand: An expression dropped into a procedure-type input slot would be ringed or not ringed depending on how close to the slot the expression was dropped (Figure 10). In retrospect this is the only truly embarrassingly bad design decision we made in BYOB

About formal parameters: The connection between formal parameters and actual arguments is always difficult for beginning programmers. Brian's mom was a math teacher during the post-Sputnik New Math days, and he remembered one of the lessons of those curricula: If you show an eight-year-old an equation such as $x+3=7$ and ask what x is, you're unlikely to get an answer. But if you show the same kid $\square+3=7$ and ask "what number goes in the box," s/he can solve it easily. We allow empty input slots as our equivalent of the box (Figure 11).

The empty-slot convention does have a pragmatic problem: Blocks that have default input values in their palette won't work in functional input slots unless the user explicitly erases the default input. For example, if you want to use the 

block to form the plural of a word, it's not good enough to replace "world" with an "s"; you must also

Figure 9. Computing the function input to `call`.

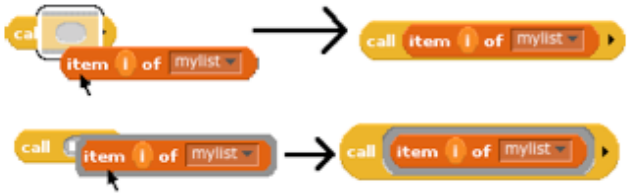


Figure 10. Using drop distance to control encapsulation.

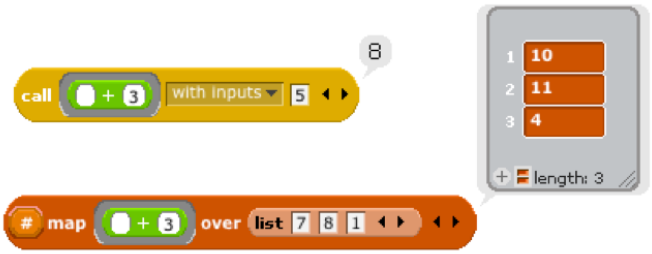


Figure 11. Empty box as implicit formal parameter.

delete the "hello." (Figure 12) We do also continue to support explicit formal parameters, but our teaching materials start with empty slots.

III. From BYOB to Snap*!*

In 2011 we started planning for the next major release, BYOB 4. Partly influenced by the Scratch Team's plans for Scratch 2.0, we decided BYOB 4 should run in a browser. Meanwhile, however, we heard vocal complaints from a few teachers about the name BYOB, and we reluctantly agreed to change it. Unfortunately, we couldn't find an equally clever alternative, so we named the new version after the act of snapping blocks together to make a script. (The new name has a backronym: Software for the New Advanced Placement course.) Snap*!* 4.0 was released in 2013.

A. *Taming the Grey Ring*

One of our biggest design concerns was to undo the distance-dependent ringing of expressions. We had two somewhat contradictory goals: to handle every possible case, and to allow users to use higher order functions with essentially no knowledge of encapsulated procedures, and with no arcane maneuvers.

The ultimate solution was to try to make the common use cases really easy, even at the cost of (1) an evaluation model that's hard to explain in its entirety and correctly, and (2) requiring more user sophistication for the less common cases.

As a simple example of the first point, we decided that blocks or scripts dropped into a procedure-type input slot are *always* ringed, *unless the block is a variable reference.* Variable blocks are not ringed. Although it's possible to imagine wanting to delay evaluation of a variable (e.g., because its value will be changed later), the most common case

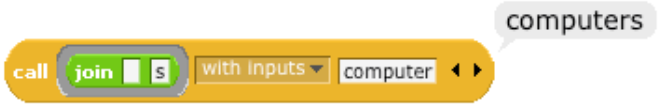


Figure 12. The user must explicitly empty the left box in `join`.

Figure 13. Using a reified `item` block as the function input to `map`.

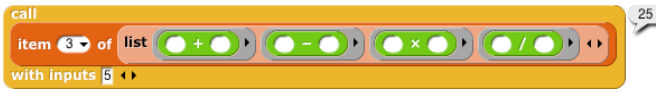


Figure 14. A non-reified `item` block computes the function input.

of putting a variable into a procedure-type input slot is that the user is writing a higher order function, and the variable is a formal parameter of that function, representing a functional input. In that case, the procedure to be called is the one that's the value of the variable, not the variable block itself.

The item of block is a difficult special case. Most of the time, it should be ringed, because the user's intent is to invoke the `item` block itself as the anonymous function (Figure 13). But if the list input is a *list of functions,* then it makes sense to evaluate the `item` block right away to determine which function to call (Figure 14). This is one of the use cases we decided would be uncommon enough that we could require extra sophistication on the part of the user, who must drop the `item` block into the `call` block (ringed, automatically) and then right-click it and select "unringify" from its context menu.

This last example also shows that there are still situations in which the user must explicitly ask to encapsulate expressions as functions: The `list` block takes inputs of any type, and so it would ordinarily take the arithmetic operator blocks as expressions (meaning, as in Scratch, 0+0, 0–0, etc.) to evaluate. To construct a list of functions, the user must either choose "ringify" from the context menu or drag a ring from the palette; they're now at the top of the Operators palette.

What if the user ringifies a block and *then* drags it into a procedure-type input slot? Does it get ringified again? We decided that in such a situation the user is almost certainly

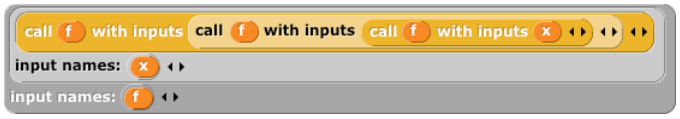


Figure 15. The Church numeral three.

Figure 16. Two nested `map` calls to process a matrix

confused about how Snap*!* handles functions as arguments, and does not actually want a double ring. So in this situation the two rings are "absorbed" into one. There *are* uses for doubly-ringed expressions, but they're rather advanced; an example is the definition of Church numerals in λ calculus, in which the number 3 is represented as

$$\lambda f\,.\,\lambda x\,.\,f\,(f\,(f\,x))$$

In Snap*!* this would be rendered as in Figure 15. More common, but unproblematic, is the case of rings that are nested but with blocks separating them, as in Figure 16. (Note in passing that both kinds of nested rings really call for explicit formal parameters to make the code readable, even though the matrix example would do what the user intends even with empty-slot implicit inputs.)

Church numerals aside, though, the main result of this redesign is that a naïve user can use higher order functions even without understanding what the rings mean! The ring for a functional input is now built into the input slot itself, as in Figure 17. The user drags an expression into the ring, and is unsurprised that the ring is still visible. Using the empty slot notation for implicit parameters, it's easy to construct Figure 18 and just read it as "map plus-three over numbers."

IV. Still to Come

We haven't yet come up with a satisfactory (to both of us) solution to the problem of blocks with default input values.

More excitingly, we are planning in the next version of Snap*!* to have self-reflection for programs, so that we can implement the equivalent of Lisp macros. This will complete the project of smuggling all the major ideas of Scheme into Scratch.

Figure 17. The map block provides a grey ring where a function is expected.

Figure 18. "Map box plus three over numbers."

A Module System for a General-Purpose Blocks Language

Yoshiki Ohshima
Yoshiki.Ohshima@acm.org
CDG

Jens Mönig
jens@moenig.org
Communications Design Group (CDG), SAP Labs

John Maloney
jmaloney@media.mit.edu
CDG

Abstract

Our team is developing GP, a new blocks language that aims to be beginner-friendly, like Scratch, yet capable of scaling up to support larger applications. We hope to allow a world-wide community of users to share projects, sprites, and code libraries and to create new ones using a mash-up development style.

To support such easy sharing and reuse, GP incorporates a strong notion of *modularity*. Modularity allows components created by different users at different times to interoperate without worrying about conflicts. However, we do not want modularity to add to the burden of the beginning programmer; we'd like GP's module features to stay out of the way until they are needed.

In this paper, we present the key ideas around the GP module system. A module in GP is a unit of encapsulated code and data. It exports a selected set of classes, functions, and variables, provides a namespace for the code inside, and may include private helper classes, functions, and variables. A module can also extend system classes with additional methods. Such extensions can be used freely within the module but are not visible outside it. Modules can be saved and re-loaded, allowing modules to store user-created projects, sprites, and code libraries.

I. Introduction

Our team is working on a new block-based language we are tentatively calling GP (Figure 1). In GP, as in Scratch, a beginner can start programming without having to learn non-essentials (i.e, it has a "low floor"). At the same time, as GP programmers learn and gain experience, they can create sophisticated applications or even extend the system itself ("high ceiling"). In other words, we hope to create a blocks programming language that is as beginner-friendly as Scratch while being as "general purpose" as languages like Python or Java.

There are three GP design goals that are especially relevant to the module system:

- **Blocks-Based:** We created a new language specifically designed for visual blocks. The user can write code in either blocks and text and can view, edit, and debug GP code as blocks, including GP library code and the GP programming environment itself.

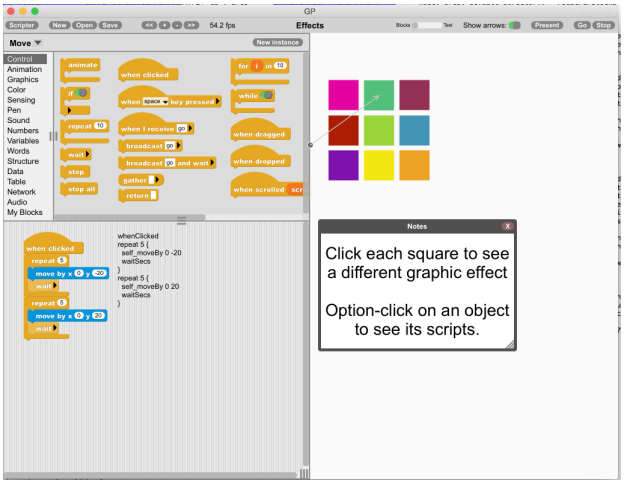


Fig. 1. A Screenshot of GP. On the left, the authoring tool is shown with a block palette and a scripting area, which is showing a stack of code in blocks and in textual code. The project being edited has some colored rectangles and an explanation in a text box.

- **Self-Sufficient:** To maximize the potential for learning about and extending GP, most of the core library and the entire programming environment is implemented in GP itself; only the GP virtual machine and low-level primitives for things such as graphics and file system operations are written in C. While not all users will want to explore GP's implementation, those of us who have used Smalltalk or Lisp systems know how much one can learn by "lifting the hood" and studying the workings of mathematical operations, collections, UI frameworks, and the programming environment.
- **Simple:** To encourage budding GP programmers as they explore and extend the GP system, GP has a carefully selected set of features. A conscious effort has been made to eliminate language features that could be obstacles for learners, while still retaining enough expressive power to allow GP to be used to build the GP system itself. For example, while GP is a class-based object-oriented language, it does not have an inheritance mechanism. GP system code uses explicit delegation, which we hope will be easier for learners to read and understand.

In short, GP is a class-based object oriented language

978-1-4673-8367-7/15/$31.00 ©2015 IEEE

(somewhat influenced by Smalltalk) without inheritance and with a syntax that is easily represented as visual, puzzle-piece-shaped blocks.

A key idea around this new language is that a future programmer may want to make her applications in the *mash-up* style; that is, she would take sprites, projects, libraries and other kind of components created by her fellow programmers, and make her own by assembling them.

Imagine how cool it would be if the authoring tool itself were the product of this process, with modules contributed by various users! Also, a modular design, where the authoring tool itself can be controlled programmatically, would allow users to create tutorials or present a collection of interacting components written by different users.

Existing blocks languages offer various extension mechanisms. Snap supports libraries of user-defined blocks [1]. Android App Inventor's components add new blocks appear to the palette when dropped into the application being constructed, but as far as the authors can tell does not allow users to create and share new components [2]. The Scratch Extension Mechanism[3] comes closer to our vision. However, Scratch extensions cannot extend the Scratch editor itself and must be written in an entirely different language (JavaScript), not Scratch itself.

To facilitate this mode of collaboration, we need a strong assurance that someone else's component (or even your own previously created component) does not break any other components in the application. In other words, we would like to have good *modularity* in the language.

We looked at some ideas of existing module systems, most notably from the Newspeak [4] language and the Node.js Package Manager (npm) [5], and designed a module system for GP. A unique challenge for GP is to provide a module system that works, but at the same time does not get in the way of users, especially beginners.

The remainder of this paper focuses on the design and implementation of GP module system. In Section II, we briefly describe the GP language. In Section III, we explain the design of the module system design and show some examples.

II. GP in a Nutshell

GP is a class-based object-oriented language. Everything in the system is an object and has a class, which defines its instances' behavior. The system contains some classes that are considered to be system-defined, such as `Array` and `Morph`, and `Turtle`.

As the system-defined classes are also written in the same language, their definitions can serve as examples of the GP language:

```
defineClass Turtle morph x y direction

method forward Turtle n {
  x += (n * (cos direction))
  y += (n * (sin direction))
  ...}
```

In this code, a class called `Turtle` is defined with four instance variables (`morph`, `x`, `y` and `direction`). Then a method called `forward` is defined for the `Turtle` class. The method takes an argument and mutates some of the instance variables.

The following code creates an instance of `Turtle` and calls the method on it:

```
aTurtle = (new 'Turtle')
forward aTurtle 10
```

As you can see, the method name (`forward` comes) first, followed by the arguments. The method look up is single-dispatch; the method dictionary of the class of the first argument is searched to find the actual implementation of the method and invoked.

Besides methods, GP has functions that are not bound to a particular class. For example, you can define a function (that does not belong to any class) called `myAdd` and call the function with arguments 3 and 4:

```
to myAdd a b {
  return (a + b)
}
myAdd 3 4
```

There is no built-in inheritance mechanism between classes, thus there is no class hierarchy. However, it is desirable to be able to have one implementation of functionality that is polymorphic to the instances of different classes. The functions are used to support this. Namely, when a method look up fails, the system looks up the function that has the same name, regardless of the first argument's class.

As you can see, calling a method and calling a function have the same syntax: the function name, or the selector, comes first, followed by arguments. This syntax makes calling a method and calling a function polymorphic and allows the unbound functions to work as fallback code.

To have better extensibility, we feel that the system-defined classes should be *open*. That is, it should be possible for an end-user to add a method to a system class such as `Array` and extend its vocabulary We will return to this point in Section III-E.

Classes and functions are first class objects as well. For example, when the user creates a sprite in the authoring tool, the authoring tool creates a class for it programatically, and assigns a name such as `MyClass1`.

But here is the fundamental problem. If class names in user extensions were globally visible then one user's class names could conflict with those of another user when both users' modules are loaded into the same project. Likewise, without a good module system, one user's extensions to system-defined classes might conflict with another user's extensions or change the behavior of core classes in unexpected ways.

The module system proposed in the next section addresses this problem by providing a well-defined mechanism for namespace isolation between modules.

III. The Module System

A module in GP is a unit of encapsulated code and data. A module may have classes and functions inside, and their names are unique within the module and don't collide with classes and functions in other modules. Classes and functions that are for use by code outside the module can be *exported*, while the rest can be hidden.

The term "module" has been used in different ways in different systems. Compared to other module systems, GP's module system has the following characteristics:

- **Instance-based:** At runtime, a module is represented as an object (an instance of the `Module` class). It differs from being a compile-time name resolution mechanism, such as that of the Modula family of languages [6]. It is more similar to a module in Node.js package manager (npm), but with a distinct difference: loading a GP module multiple times creates a fresh instance for each time, whereas the npm returns the same cached instance. See below for the rationale for this distinction.
- **Local classes and functions are objects:** As GP represents classes and functions as objects (instances of `Class` and `Function` class), a runtime module instance stores the classes and functions local to it in its data structures.
- **Supports meta-operations:** The module system does not aim to make a module be a total black box. While local classes and functions cannot be used directly from the outside in normal circumstances, we feel it essential to be able to write introspection tool in GP itself that analyze and manipulate modules.
- **Objects can escape:** Instances of local classes are allowed to escape to the outside of the module. For example, an instance of a private helper class can be returned from a method call.
- **No nesting:** At the moment, we are not considering supporting statically nested module definitions. A module can use other modules by instantiating them dynamically, but there is no static hierarchy of modules.
- **A module is an object but an object is not a module:** This design is not pursuing a unification between objects and modules. The description of module as a unit of encapsulated code and data could also apply to an object, and some other languages, most notably Newspeak, unify these concepts.

A. A Module Example

A module definition for a paint editor is shown here to explain the proposed module system. Following the npm tradition, a module object behaves as a dictionary in which publicly visible items are stored. But it also has fixed instance variables to hold classes and functions and "Expanders" (see Section III-E) defined in this module. These variables are called `classes`, `variables`, `functions` and `expanders`.

```
module 'PaintEditor'
moduleVariables DefaultPenColor CurrentColor Palette
moduleExports Editor
moduleExports changePenColor

// the initializer of the module
to initializeModule {
  DefaultPenColor = (color 0 0 255) // blue
  CurrentColor = DefaultPenColor
  Palette = (readPNG 'palette.png')
}
// classes and functions
defineClass Editor a b c
method handMove Editor x y {...}
defineClass Pen a b c
method drawCircle Pen x y r {...}
to changePenColor newColor {
  tmpVar = CurrentColor
  ...
  CurrentColor = newColor
}
```

The `module` command indicates the start of a module definition, and the subsequent code up to the end of the compilation unit (or file) defines the content of the module.

When a module is loaded, an instance of `Module` class is created, and its instance variables (`classes`, `variables`, `functions`, and `expanders`) are filled with the items declared in the module definition. In this example, classes called `Editor` and `Pen` are defined, and a function called `changePenColor` is also defined in this module. Note that even a common name such as Editor can be used without fear of name clashes, because the module provides an isolated namespace.

The `moduleVariables` command declares module variables. In this example, `DefaultPenColor`, `CurrentColor`, and `Palette` are declared. These variables are visible from classes and functions defined in the module.

The `moduleExports` command specifies the items to be exported. The arguments are the names of classes, functions, or variables that are to be exported. Recall that the module object itself behaves as a dictionary; this dictionary is populated with items that are visible to the client of this module. In this example, the `Editor` class and the `changePenColor` function are exported, while the `Pen` class is not.

The module initializer may take arguments to initialize module variables:

```
module 'AnotherEditor'
moduleVariables MyPen

to initializeModule aPenModule {
  MyPen = (new (at aPenModule 'Pen'))
}
...
```

In this example, `aPenModule` is a module that exports a class named `Pen`, and the `MyPen` module variable is initialized with an instance of `Pen`.

Meta-language features to introspect classes and functions, such as `class` and `functionNamed`, accommodate the notion of modules. For example, there is a fucntion called `class` that takes a string as its argument and returns a class

object. It first searches the list of classes in the module where the call occurs. If the name does not appear in the module's list of classes, then the search is delegated to the top-level module (explained in section III-B). This allows all modules to use core classes such as Array.

With this name resolution rule in place, all code in a module can be written with "short names"; in other words, the creator of a module need not be concerned with how the module will be referred to and used.

Loading a module creates a new instance of the `Module` class and fresh copies of classes and functions that belong to the module. This is important as a user may want to create two copies of the same module and then modify one of them. Imagine that a user wants to improve the paint editor, using the existing paint editor to create and edit button icons. In that case, being able to have two independent copies is very helpful.

In npm, a `require()` function call returns the same cached instance when called with the same argument multiple times. This allows modules that have circular dependencies to load, and allows sharing. However, by passing in module parameters at module creation time (as arguments for the `loadModule` function call), or by setting module variables afterwards, we can build any network of modules. In other words, creating a fresh instance by default seem to be helpful for common use cases.

With this code snippet:

```
painterModule1 = (loadModule 'PaintEditor.gpm')
painterModule2 = (loadModule 'PaintEditor.gpm')
painter1  = (new (at painterModule1 'Editor'))
painter2  = (new (at painterModule2 'Editor'))
```

the variables `painterModule1` and `painterModule2` stores instances of Module initialized with the definition in `PaintEditor.gpm`, and `painter1` and `painter2` each have an instance of `Editor`, thus they can have distinct values for `CurrentColor`.

The following scenario illustrates how a project would evolve. Imagine that making improvements to the (current version of) paint editor requires to change the palette object as well as code. In that case, the authoring environment would load two instances of the paint editor module. One of these would be used as a stable working version to edit the new palette. The other would be used as the development version. The authoring environment is able to modify the development version on the fly so that the programmer could experience the edit-and-continue style work flow. Even if some change to the development version causes it to break, the stable version will be intact.

Note that the module instances don't need to be stored in variables. The above code could simply be written as:

```
painter1 = (new (at
                 (loadModule 'PaintEditor.gpm')
                 'Editor'))
painter2 = (new (at
                 (loadModule 'PaintEditor.gpm')
                 'Editor'))
```

On the other hand, sometimes it is useful to keep a reference to the module in a variable. For example, one could call a function from painterModule1 in the following manner:

```
// get the function
changeColor = (at painterModule1 'changePenColor')
// call it
call changeColor (blue)
```

or, more succinctly:

```
call (at painterModule1 'changePenColor') (blue)
```

B. The Top Level Module

GP comes with a set of core classes the define basic data types (Integer, String), collections (List, Dictionary), the Morphic UI framework, and the programming environment itself. These classes are stored in a module, called the top-level module, that is visible to the code in all other modules. Similar to the method look up rule, the look up rule for a class or function first checks the module where the look up starts. Then, if it does not find an entry there, it looks for it in the top-level module.

C. Sprites and Projects as Modules

The simple formulation of the module system allows us to represent user projects as modules. When a user enters the visual authoring environment and starts making a project, a module instance is created to store the work in the project.

A project is a set of sprites, each represented as a class, along with additional code (such as functions) and data. It is therefore a natural match for being represented as a module. The user may interactively add more sprites, variables, and classes. When a project is saved, these user classes and the serialized sprite data are stored into a module definition. As sprite properties can be modified by direct manipulation in the authoring environment, code alone may not be able to recreate the resulting state. So, keeping the Sprites' state as serialized data is necessary.

An open question is that how far such a user can go without needing to know about the module system. For for simple projects, the authoring tool can build module "behind the scenes", creating classes, variables, and functions inside that module as the user works and saving the entire project as a module. A more advanced user can import and use modules designed to extend GP by making new blocks available in the palette without needing to understand the module mechanism in detail.

Ideally, only when a user wants to create GP extensions that add features to the programming environment or export libraries of blocks meant to be used by other users would would they need to learn how modules work in more detail.

D. A Module Using Another Module

The fact that modules are just objects allows great flexibility in how one module uses others. Let us say the `readPNG` function used in the `PaintEditor` example was actually defined in a separate module called `PNGReader`. The line in the initializer section for the `Palette` image might load it like this:

```
Palette
  = (call (at (loadModule 'PNGReader.gpm')
              'readPNG') 'palette.png')
```

Once the job of reading the PNG file (`palette.png`) is done, the instance of `PNGReader` module is garbage collected without leaving a trace.

In other cases, an exported class or a function may be stored into a variable on the client side:

```
myReader = (at (loadModule 'PNGReader.gpm')
               'readPNG')
Palette = (call myReader 'palette.png')
```

In this example, a function from the `PNGReader.gpm` module is stored into the variable `myReader`, but the module itself does not get stored into any variable. Because the function has a pointer to the module object (as described in III-F), the module instance will be kept in memory as long as it is referenced from such a class or a function.

E. Expanders

As mentioned earlier, the core classes in the top-level module are visible in all modules. If a module defines a class with the same name as one of the core classes, then that definition will replace the core class within that module. However, there are times when a module may simply want to add a few additional methods to a system class.

The desired behavior is that a method added to a top-level class by one module should not effect any other modules, even when another module adds a method with the same name to the same class. In other words, we would like each module to be its own isolated universe with it's own private extensions to the core classes.

There have been research efforts in the past on this problem, including Expanders [7] and ClassBoxes [8]. We've adapted the Expander idea to GP.

The original Expander design (for Java) added a new language construct to specify the scope of expanders. However,the GP module mechanism already supports isolated code units, so by using a module as the scope for Expanders, no new construct is necessary. Any method definition for a class that is not defined in the module is only visible from within that module. In other words, such a method definition automatically becomes an expander.

For example, if you have the following definition in a module:

```
method f1 Array x y {..}
```

and a definition of `Array` does not exist in the module, `f1` becomes an expander. Code defined in the module can call this method on any instance of `Array`, but it does not change the behavior of Array for the code in other modules.

F. Implementation of Modules

In the current implementation, each function or method holds a pointer to the module it belongs to. When a call happens, the selector is looked up in the following order, and the first implementation found is invoked:

1) The expander data structure for the receiver's class in the module of the method where the call occurred (this is statically determined.)
2) The receiver's original class.
3) The `functions` of the module where the call occurred (this is statically determined.)
4) The `functions` of the top-level module.

As you can see, the look up rule is quite simple. The only complication over a language such as Smalltalk is that the method lookup needs to check the expanders for the module. Also, instead of looking up the superclass chain for an implementation, GP looks up the functions.

As the lookup result can be cached, the extra step of looking up in the expander data structure does not incur a significant performance penalty.

We are aware that the expander concept and unbound function could be unified. A function defined in a module could be thought as an expander for the class of `nil`, and the lookup mechanism can work accordingly.

IV. Future Work

Our work is still its initial state, and some features have yet to be designed, implemented, and tested. Some of the (obvious) missing pieces are described in this section.

A. Module Dependency

A module can depend on other modules. But since GP modules are created and loaded dynamically, we cannot statically determine the dependencies among modules. To detect such dependencies, when the serializer enumerates objects in the project (or a module), it needs to check which modules the objects' classes belong to. The other modules that the serializer detects are considered *dependencies* of the module, and those modules will be recorded.

B. The Externalized Form of Modules

When a module is saved to disk, the saved content would need to contain enough information to recreate it in memory without losing any information.

We will need a proper scheme to identify and store networks of modules. This is not yet implemented, but here are some goals we hope to achieve:

- **No Loss of Information:** Code may be written visually as well as textually. Data may be assembled by directly manipulating objects. The textual code with the `module` keyword in this memo represents code written textually and when that is what the user wrote, it should be

faithfully stored. At the same time, there is data (such as pictures, movies, sound, etc.) that also need to be saved.

- **Version Control and Dependency:** As noted above, a module may depend on other modules. These dependencies will need to include the identification of the modules' versions.
- **Self-contained "Timeless" module files:** Also, we would like to support the notion of "timeless" software, at least as good as Smalltalk's image files are capable of. We would like to be able to "bundle" all dependencies into one blob (file) and have a guarantee that that one blob can be run on a conformant virtual machine bit-identically, perhaps many, many, years later.
- **The Community Site:** Eventually, we envision a shared GP repository of projects, sprites and libraries with a complete history of past versions, a sort of hybrid of GitHub and the Scratch web site.

Toward these goals, one possible design of such a file is as follows: The file that represents a module would be a zip compressed directory structure. A sub-directory stores the resource data files, and the textual module definition. Another sub-directory stores a binary file that is the pre-processed file, in which resource files are internalized and code is parsed and converted to the runnable form. For a module to be properly stored and reloaded, some meta-information such as the version (perhaps the ancestor tree of versions in SHA1), and its dependencies will be needed.

The module might contain another optional sub-directory to contain other modules, thus the resulting zip file can be fully self-contained.

To ensure that a version of a module is uniquely identified globally, we would have a naming scheme. It may be assumed that a user (at least when publishing things) registers himself at the central server and obtains a unique user name. The user then is responsible for naming published projects uniquely. And we may assume that we don't support branches of a project but only a linear sequence of versions of a project. Thus, a project, (or a module), can be identified as a triple of (user name, project name, version) from the program. In the above, we used a simple file name, such as "PaintEditor.gpm", as the descriptor of a module; in the real system, we may specify a module with the fully qualified name.

V. Conclusion

This paper presented the design of the GP module system. We believe this design is simple and flexible, yet provides sufficient isolation to allow modules to be mixed together without fear of unexpected interactions.

The GP programming environment makes the concept of modules invisible to beginners by automating module creation. Our implementation shows that this module design does not impose significant complexity or performance penalties on the GP execution model.

In the future, we hope to use modules to support sharing of components, block libraries, and even programming environment extensions. We plan to facilitate sharing of such things within the GP community by creating a shared module repository.

Acknowledgments

The authors would like to thank the members of CDG, especially Todd Millstein, Alex Warth, Mahdi Eslamimehr, Aran Lunzer, Jonathan Edwards, and Alan Kay for valuable suggestions. The authors also thank the anonymous reviewers of the paper.

References

[1] Snap! `http://snap.berkeley.edu/`.
[2] App Inventor. `http://appinventor.mit.edu/`.
[3] Sayamindu Dasgupta, Shane M. Clements, Abdulrahman Y. idlbi, Chris Willis-Ford, and Mitchel Resnick. Extending Scratch: New Pathways into Programming. In *Visual Languages and Human-Centric Computing (to appear)*, 2015.
[4] Gilad Bracha, Peter von der Ahé, Vassili Bykov, Yaron Kashai, William Maddox, and Eliot Miranda. Modules as objects in newspeak. In *ECOOP 2010 - Object-Oriented Programming, 24th European Conference, Maribor, Slovenia, June 21-25, 2010. Proceedings*, pages 405–428, 2010.
[5] npm: Node.js Package Manager. `http://www.npmjs.com`.
[6] Niklaus Wirth. MODULA. A Language for Modular Multiprogramming. Technical report, 1976.
[7] Alessandro Warth, Milan Stanojevic, and Todd D. Millstein. Statically Scoped Object Adaptation with Expanders. In *Proceedings of the 21th Annual ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications, OOPSLA 2006, October 22-26, 2006, Portland, Oregon, USA*, pages 37–56, 2006.
[8] Alexandre Bergel, Stéphane Ducasse, Oscar Nierstrasz, and Roel Wuyts. Classboxes: Controlling visibility of class extensions. *Comput. Lang. Syst. Struct.*, 31(3-4):107–126, October 2005.

Robotics Rule-Based Formalism to Specify Behaviors in a Visual Programming Environment

Daniela Marghitu
Computer Science and Software Engineering
Auburn University
Auburn, USA
marghda@auburn.edu

Stephen Coy
FUSELABS
Microsoft
Redmond, USA
scoy@microsoft.com

Abstract*—Microsoft's Kodu is a visual programming environment for children inspired by robotics rule-based formalism to specify behaviors. The reactive nature of this approach means that systems just react to what they are currently sensing, they do not need to have a perfect model of the world, and they can easily respond to external changes in the world. This resilience makes behavior-based systems perfect for real world robotics use. When developers chose this approach for Kodu, the hope was that this resilience would help minimize the potential frustration of young users. This paper introduces the Kodu developers' strategy, results and future plans in developing a user-friendly programming environment that successfully introduces children in a fun and easy way to the fundamental concepts of computer science and computational thinking.*

Keywords*—Kodu; rule-based formalism; computational thinking*

I. Introduction

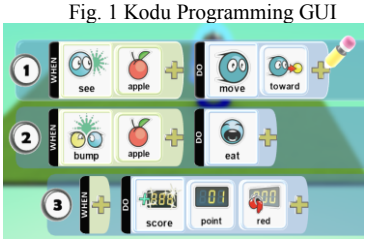
Several theoretical justifications are available for the successful use of manipulative resources that enable students' progress through a concrete-representational-abstract learning sequence [1]. In computer science (CS), physical manipulatives represent the foundation of CS Unplugged activities [2], and virtual manipulatives (e.g., shaped blocks corresponding to syntactic classes such as imperative statements, Boolean expressions, and loops), are used to develop the Graphical User Interfaces (GUI) for drag and drop based programming frameworks [3].

When developers decided to create the Kodu language [4] in late 2006, the goal was to address the challenge of introducing children to CS and computational thinking in a fun way while developing 3D simulated worlds. At that time Scratch was already well established and has since generated numerous imitators. Instead of following Scratch's example, Kodu developers took as inspiration the robotics rule-based formalism to specify behaviors [5] [6]. The reactive nature of behavior-based programming means that the systems just react to what they are currently sensing, they do not need to have a perfect model of the world, and they can easily respond to external changes in the world. This resilience makes behavior-based systems perfect for real world robotics and its numerous applications. When Kodu developers chose this approach, the hope was that this resilience would help minimize the frustration of introductory, young programmers.

II. The Past

The Kodu language consists of a series of rules that are evaluated for each frame (nominally 60 times per second). Each rule consists of a *WHEN* clause and a *DO* clause. The *WHEN* clause consists of a *Sensor and zero* or more filters. The *DO* clause of the rule contains an *Actuator* and zero or more *Modifiers*. For instance, in Fig. 1 the first rule shows the *See sensor* along with an *Apple filter*. This will sense all apples in the world. The list of apples in the world is internally reduced to a single instance using a *Selector* which chooses the closest instance. If an apple is seen then the *DO* clause of the rule is applied. In this case the *Move Actuator* is activated. The direction of the movement is controlled by the *Toward Modifier* which uses the location of the apple detected in the *WHEN* clause as the target destination. Similarly, in the second rule, if the *Bump Sensor* detects an apple that apple is passed to the *Eat Actuator* to be acted upon.

Fig. 1 Kodu Programming GUI



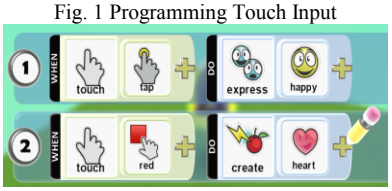
The third rule demonstrates another aspect of Kodu programming. Rules can be indented to indicate that they are a *child* of the preceding rule. Child rules are only executed when the *WHEN* clause of their parent evaluates to true, in this case when an apple is bumped. The empty *WHEN* clause in rule 3 implies that the *WHEN* always evaluates to true. One of the results of the reactive nature of the language is that the movement system in Kodu is both easier to work with and yet less accurate. Precise movement, as seen in Logo [7], is not possible. Instead of moving in precise increments Kodu characters move "toward" other characters. Since various characters have differing movement capabilities the speed and accuracy of the resulting undertaking may vary. If the target

character changes position during this process, the reactive nature of the system comes into play and the movement of the programmed character adjusts automatically. Another advantage of the language's robotics roots is that the tiles in the system have comprehensible, physical counterparts. Kids readily understand seeing an apple or eating an apple. The high-level nature of the tiles empowers users to create interesting behaviors with only a few lines of code. This model is quickly providing a positive feedback cycle for new programmers.

Another way that Kodu eases the transition for new programmers is that the language doesn't use word *variable*. Instead, users are presented with *scores* that can be added to, subtracted from and compared. From a CS perspective, there is no difference. However, for new users, the concept of a score and how it can be manipulated is already an established, and therefore, concept. By playing to the familiar for these young programmers, Kodu helps to eliminate one more potential stumbling block. Similarly the *when/do* terminology was also carefully considered and chosen over the more obvious *if/then*. The use of *when* implies that the Boolean value of the statement may change over time whereas *if* implies a more static value. For the actuator clause *do* is clearly more action oriented, than *then*. In some ways the difference may seem trivial but when trying to reach new programmers the developers wanted every advantage.

Kodu's origins on the Xbox dictated the GUI to be fully accessible via the Xbox Game Controller. This constraint has proven beneficial since it ensures a minimalist interface. With the move to the PC, full support for keyboard and mouse was added. More recently, touch support has been added, which makes Kodu more accessible for the user with hand motion control impairments {see Fig. 2 [8]}. In each case, the GUI responds dynamically as the user switches the input devices. The support for multiple input devices also extends to the Kodu programming language.

Fig. 1 Programming Touch Input

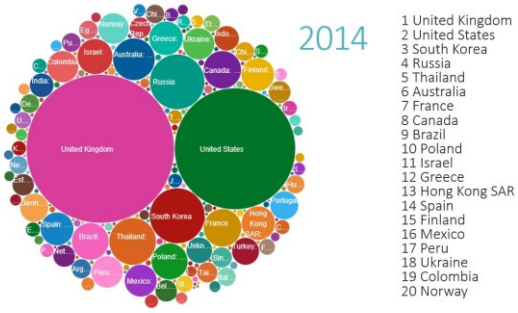


The Kodu language has been extended to support mode specific input within the games created by the users. As shown in Fig. 2, Kodu characters can be programmed to respond to touch input. Additionally, when the user programs a character to respond to a touch on a button, in this case the red button, the button will then appear on the screen and respond to both touch and mouse input.

III. The Present

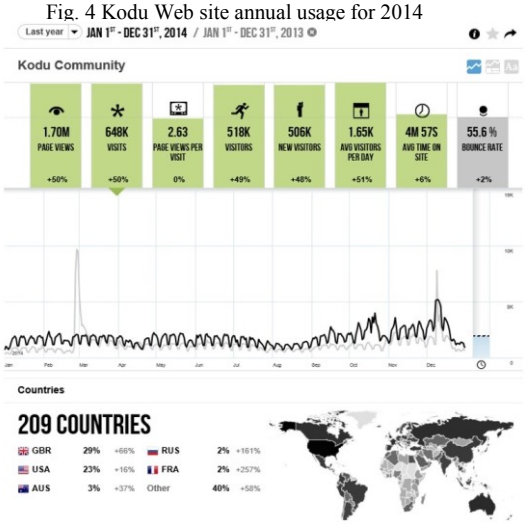
To date, Kodu has been translated into seventeen languages (by the Kodu community members), and has been successful around the world. Kodu is one of the platforms recommended for K-8 educators by Code.org, along with Alice and Scratch [9]. It is also one of three languages, along with Scratch and Logo, recommended by the Computing at School Working Group for the new United Kingdom (UK) primary school computing curriculum [10].

Fig. 3 Kodu top 20 downloads for 2014 by country



Kodu is currently downloaded in over two hundred countries. Fig. 3 shows the top twenty countries where it is used. Currently about 135k games have been uploaded to the KoduGameLab web site [4]. The UK surpasses the US, and this is mainly due to the UK laws requiring CS education at all K-12 grade levels. It is estimated that as much as 1/3 of school age children in the UK have been exposed to Kodu. It's also interesting to note that South Korea and Thailand both show up in the top 5 despite not having localizations available. There's clearly a demand in Asia for something like Kodu, and the developing team hopes to fill it. Fig. 4 shows the 2014 annual usage of the Kodu web site. The remarkable thing indicated by this report is that Kodu is clearly used more during the week than the weekend. For the developing team, this is another indicator that Kodu is being used largely in schools or after school programs.

Fig. 4 Kodu Web site annual usage for 2014



IV. The Future

In 2014, Kodu developers released its commercial successor Project Spark (projectspark.com). The language used in Project Spark showed one possible progression of the Kodu language where the priority focused more on functionality and flexibility rather that simplicity and ease of use.

For the next generation of Kodu, the developing team plans on continuing Kodu's focus on young, first-time programmers. One of the key improvements developers hope to make is to eliminate the possibility of incomplete statements. Whereas the Kodu language does prevent syntax errors, it still allows incomplete statements to be created. For instance, in Fig. 1 rule 3 shows incrementing the red score by one point. If the *Red tile* is not placed then it is ambiguous which score should be incremented. Similarly in the *WHEN* clause we often see kids programming *WHEN GamePad DO Move* to control their characters when they need *WHEN GamePad LeftStick DO Move*. The first option looks like it should be functional but isn't. The developing team plans to demote ambiguous tiles like *Score+* and *GamePad* and make them groups in the tile selection menu. This would then make *LeftStick* a sensor rather than a filter on *GamePad*. Likewise, instead of *Score+ Red* we would have *Red+,* which would eliminate the ambiguous case. The incomplete statement syntax was not seen as a problem (reasonable defaults were chosen for these cases) until the Kodu developing team started extending the language. Recently, the Kodu team has doubled and that will allow a bit more aggressive work on updates. However, this will still be a large task and will, unfortunately, break backwards compatibility for our existing users.

Closely related with these ambiguous tiles is a problem that occurs when users have filters than could possibly apply to multiple objects. The users' worst case for this is the *ShotHit* sensor that detects when a shot hits a character. With the program *WHEN ShotHit Red* it is not clear if the red filter should apply to the shooter, the target, or the shot itself.

V. CASE STUDY

Kodu, aimed at middle and elementary school students [10] [11] [12] [13], was integrated in the Auburn University (AU) Laboratory for Education and Assistive Technology Lab (LEAT) K12 computing outreach and research inclusive program since 2008. Kodu curriculums were developed for formal and informal K12 programs.

A three-step adaptive model and related curriculum for introduction to computing was developed for AU K-12 Robo Camp [14]. In this model of computing instruction, students start with the highly scaffolded programming environment, Kodu, and progress to more challenging frameworks (e.g., Alice or App Inventor and Lego EV3). While moving forward and sometime backwards, deepening their individual abilities and preferences, between the three steps of the model, students are encouraged by instructors to explore how concepts such as variables, conditionals, and looping are implemented toward building the foundation of their computational thinking. Fig. 5 shows a Kodu world developed by Robo Camp middle school students as an implementation of the CS Unplugged Binary code activity [2]. Another informal curriculum was developed in 2013 for the Computer Science for all Girls (c4allg.eng.auburn.edu) summer camp [8], funded by the 2012 National Center for Women in Information Technology Academic Alliance Seed Fund.

Fig. 5 Kodu world implementing the CS Unplugged Binary activity

A formal Kodu curriculum was developed for 7th grade, (cs4allb.eng.auburn.edu) with funds from Microsoft Research and AccessComputing, and has been taught since 2013 [14]. Feedback obtained from instructors, participants and participants' parents, reinforced by the results of the pre and post programs evaluations, clearly indicated all these programs as a success. Therefore, the LEAT team will continue to refine its formal and informal Kodu curriculum for middle school and explore moving the first programing instruction to elementary school.

REFERENCES

[1] ETA hand2mind; "Why Teach Mathematics with Manipulatives?" http://www.hand2mind.com/pdf/Benefits_of_Manipulatives.pdf. Accessed 2015 July 1.

[2] CS Unplugged, http://csunplugged.com/, Accessd 2015, July 1.

[3] Ward, B., Marghitu, D., Bell, T., Lambert, L.: Teaching computer science concepts in Scratch and Alice, Pages: 173-180, Journal of Computing Sciences in Colleges, Volume 26, Issue 2, December 2010

[4] Kodu, http://www.kodugamelab.com/. Accessed 2015 July 1.

[5] J. Jones, Robot Programming – A Practical Guide to Behavior-Based Robotics, New York, McGraw-Hill, 2004.

[6] Coy, S. "Kodu game lab, a few lessons learned", XRDS: Crossroads, The ACM Magazine for Students, Pages 44-47, Volume 19 Issue 4, Summer 2013, ACM New York, NY, USA

[7] S. Papert, Mindstorms: Children, Computers, and Powerful Ideas, New York, Basic Books, 1980.

[8] Marghitu, D., et-al.: "Attracting Girls, Special Needs, Minority and Underserved K-12 Students to Computing Majors and Careers", SITE 2014 Conference, Jacksonville, FL, March 17-19, 2014

[9] Code.org,"3rd Party Educator Resources;" http://code.org/educate/3rdparty. Accessed 2015 June 30.

[10] Berry, M. "Computing in the national curriculum: A guide for primary teachers." Computing At School, at http:// computingatschool.org.uk/. Accessed 2014 September 20.

[11] Touretzky, D. S., Marghitu, D., et-al.:"Accelerating computational thinking using scaffolding, staging, and abstraction", Proceedings of SIGCSE '13, Denver, CO. Association for Computing Machinery, pp. 609-614, 2013

[12] Touretzky, D. S.: Teaching Kodu with physical manipulatives. ACM Inroads, 5(4):44-51, 2014

[13] Fristoe, T., et-al.: Say it with systems: Expanding Kodu's expressive power through gender-inclusive mechanics. Proceedings of Foundations of Digital Games, France, June 2011

[14] Marghitu, D., et-al., "Kodu, Alice and Lego Robotics: A three-step model of Effective Introducing Middle School Students to Computer Programming and Robotics", SIGCSE 2013, Denver, CO, March 6-9, 2013

Ten Things We've Learned from Blockly

Neil Fraser
Google
Mountain View, CA, USA
fraser@google.com

***Abstract*—Over the last four years the Blockly team has learned many lessons which are applicable to block-based programming in general. The following are a collection of ten mistakes we have made, or mistakes commonly made by others. Each issue is presented as noncontroversial folk knowledge without supporting data.**

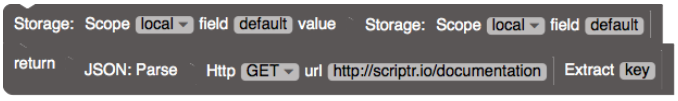
Keywords—block programming; UX; usability; user interface

I. Conditionals vs Loops

The most difficult blocks for new users are conditionals and loops. Many block environments add to the difficulty by making both blocks (already similarly shaped) the same colour and placing them in the same category. Users get caught in a spiral of confusion when they accidentally use the wrong block, which adds to the confusion when it behaves contrary to expectation. Blockly moved conditionals into the logic group, changed the colour, and the problem went away.

II. Border Style

In the 2000s the 'Aqua' look was in style and every onscreen object was decorated with highlighting and shadows. In the 2010s the 'Material Design' look is in style and every onscreen object is simplified to a clean, flat, borderless shape. Most block programming environments have highlighting and shadows around each block, so when today's graphic designers see this they invariably strip off these outdated decorations.

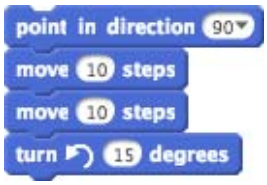


As can be seen in the above example of five blocks (from scriptr.io), these 'outdated decorations' are vital for distinguishing connected blocks that are the same colour.

III. Bumping Syntax Errors

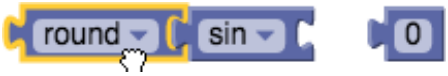
One of the great selling features of block-based programming is the lack of syntax errors. But the screenshot on the right (from Scratch) has an invisible syntax error: the 'turn' block is not connected to the 'move' block above it. There are many ways for this to happen (e.g. place a couple of unconnected 'point' and 'turn' blocks on the workspace, then connect a pair of 'move' blocks to 'point').

The solution is every time a block is moved, look for unattached connector pairs that are ambiguously close to each other and bump one block away from the other.



IV. Transitive Connections

Dragging a block should enable that block to connect to others. But what about dragging a block connected to a block and trying to make a connection on the child block? The transitive closure seems like a good idea (which Blockly implemented). But in practice it is a disaster. When working with a large program users will often drag a large assembly partially off-screen to clear room. If transitive connections are allowed, the user will often hear a click sound as the assembly randomly connects to something that was left on the workspace. Transitive connections turn large assemblies into Fluorine atoms they bond with everything.



V. Nesting Sub-stacks

'C' shaped blocks invariably have a connector on the inside-top, but some environments also have a connector on the inside bottom (e.g. Wonder Workshop) whereas others do not (e.g. Blockly and Scratch). Since most statement blocks have both a top and bottom connector, some users do not immediately see that statements will fit inside a 'C' that does not have a bottom connector.

Once users figure out that one statement block fits inside a 'C', they then need to figure out that more that one statement will also fit. Some environments nest the first statement's lower connection into the bottom of the 'C' (e.g. Wonder Workshop and Scratch) whereas others leave a small gap (e.g. Blockly). Snug nesting leaves no hint that more blocks can be stacked.

These two issues interact badly with each other. If an inside bottom connector exists (Wonder Workshop) then the initial statement's connection is made more obvious, but at the

expense of the ability to discover stacking. If no inside bottom connector exists (Blockly) then the initial statement's connection is not obvious, but stacking is discoverable. Having no inside bottom connector and nesting the statement's bottom connector (Scratch) appears to combine the worst discoverability attributes of both issues.

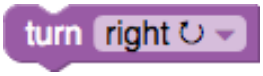
Our experience shows that the initial statement's connection is a lesser challenge for users than discovering stacking. And once discovered, the former is never forgotten, whereas the latter needs prompting. Blockly tried both the Wonder Workshop and the Scratch approaches until one day a rendering bug occurred which added the small gap. We saw a marked improvement in user studies due to this bug (now a 'feature' we are proud of).

VI. Live Block Images

Documentation for blocks should include images of the blocks it is referring to. Taking screenshots is easy. But if there are 50 such images, and the application is translated into 50 languages, suddenly one is maintaining 2,500 static images. Then the colour scheme changes, and 2,500 images need updating again.

To extract ourselves from this maintenance nightmare, Blockly Games replaced all screenshots with instances of Blockly running in readonly mode. The result looks identical to a picture, but is guaranteed to be up to date. Readonly mode has made internationalization possible.

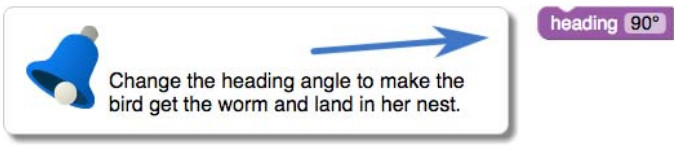
VII. Your Other Left



Feedback from children in the US (though interestingly not from other nations) revealed rampant confusion between left and right. This was resolved with the addition of arrows. If direction is relative (to an avatar, for example) the style of arrow is important. A → straight arrow or a ↱ turn arrow is confusing when the avatar is facing the opposite direction. Most helpful is a ↻ circular arrow, even in cases where the angle turned is smaller than the arrow indicates.

VIII. Instructions

Blockly Games is specifically designed to be self-teaching, no teacher or lesson plan needed. To accomplish this, the first version of Blockly Games had instructions on each level. Students would not read them. We reduced them to a single sentence, increased the font size, and highlighted them in a yellow bubble. Students would not read them. We created modal popups with the instructions. Students instinctively closed the popups without reading them, then were lost.



Finally we created popups that cannot be closed. They are programmed to monitor the student's actions and only close themselves when the student has performed the proscribed action. These contextually aware popups are challenging to program, but quite effective.

IX. Code Ownership

Exercises designed to teach a specific concept often provide partial solutions which the student needs to modify to reach the desired effect. A class of non-editable, non-movable, non-deletable blocks was created in Blockly to support this. However, students hated these fill-in-the-blank exercises. They have no sense of ownership over the solution.

Designing free-form exercises that teach the same concepts is more challenging. One technique that has proven successful is to use the student's own solution for one exercise as the starting point for the next exercise.

X. Exit Strategy

Block-based programming is often a starting point for programming. In the context of teaching computer programming, it is a gateway drug that gets students addicted, before moving them on to harder things. Nobody ever got a job because they had three years of Scratch experience. How long this block-based programming period should last for students is hotly debated, but that it is temporary is not debated.

Given this, block-based programming environments used for teaching programming must have an off-ramp appropriate to their students. Blockly Games has four strategies:

1. All text on the blocks (e.g. "if", "while") is lowercase to match text-based programming languages.
2. The JavaScript version of the student's code is always displayed after each level to increase familiarity.
3. In the penultimate game the block text is replaced with actual JavaScript (as shown to the right). At this point the student is programming in JavaScript.
4. In the ultimate game the blocks editor is replaced with a text editor.

Block-based programming environments used for teaching programming need to have a concrete plan for graduating their students. A solid exit strategy also goes a long way towards placating those who argue that block-based programming isn't "real programming".

Blocks at Your Fingertips: Blurring the Line Between Blocks and Text in GP

Jens Mönig
jens@moenig.org
CDG

Yoshiki Ohshima
Yoshiki.Ohshima@acm.org
Communications Design Group (CDG), SAP Labs

John Maloney
jmaloney@media.mit.edu
CDG

***Abstract*—Visual blocks languages offer many advantages to the beginner or "casual" programmer. They eliminate syntax issues, allow the user to work with logical program chunks, provide affordances such as drop-down menus, and leverage the fact that recognition is easier than recall. However, as users gain experience and start creating larger programs, they encounter two inconvenient properties of pure blocks languages: blocks take up more screen real-estate than textual languages and dragging blocks from a palette is slower than typing.**

This paper describes three experiments in blurring the line between blocks and textual code in GP, a new blocks language for casual programmers currently under development.

I. Introduction

We are currently developing a new general purpose blocks programming language, code named "GP", aimed at "casual programmers" (teen to adult). We hope to welcome new programmers with a blocks-based authoring system that is as easy to use as Scratch and to support them as they grow in expertise. GP has been designed to allow the code for complete applications, including the GP programming environment itself, to be viewed, edited, and debugged as blocks. Thus, the budding programmer need not learn a new language or even switch from blocks to textual code as their abilities and ambitions grow.

II. Problems

A. The screen real estate problem

Blocks-based programming languages replace text with graphical objects representing programming language elements such statements, expressions, and control structures. These graphical program blocks typically have borders ornamented with notches and indentations to suggest how the blocks fit together. Blocks contain embedded labels, icons, editable text fields, and interactive widgets such as drop-down menus and color pickers. As a result, a block representing a single statement usually takes more space than its textual counterpart. The actual amount of extra space depends on the visual design of the blocks, the hardware platform, and target audience. For example, a blocks language that targets young children using touch-screens might use larger blocks than one aimed at adults using laptops.

Short block scripts can be quite readable. Unfortunately, even a small increase in statement size multiplies with the number of blocks in a stack and the number of stacks in a window. This expansion spreads blocks code over a larger area than its textual equivalent, making it harder to get an overview of the code at a glance, and increasing the burden of scrolling and navigation. This screen real estate problem was pointed out long ago by Peter Deutsch[1]. In addition, the colors, borders, and graphical elements of blocks can be visually distracting, making it harder to scan the textual labels on the blocks that carry most of the meaning.

B. The input problem

A blocks palette helps newcomers quickly discover what commands are available. (Some blocks systems, such as Scratch, allow blocks to be tested right in the palette, further facilitating discovery and understanding.) However, experienced programmers who use blocks languages often complain that, once they know what commands are available, assembling scripts by dragging blocks out of a palette is cumbersome and takes much longer than it would take to type the code. Searching for a block in the palette involves searching one or more categories, visually scanning the blocks in each category and possibly scrolling to find the desired block. In addition to taking time, this process interrupts the users flow of thought about the code.

The input problem gets worse as the number of blocks in the palette grows. The first version of Scratch had about 80 blocks. The current versions of Scratch 2.0 and Snap each come with about 140 blocks, and they can be extended with external modules and user-defined block libraries that add additional blocks. Since GP is aimed at a wider spectrum of applications, its palette already includes 250 blocks, and since GP is designed to be easily extended, that number will grow.

We seek to combine the benefits of blocks with the speed of reading and writing textual code. The rest of this paper describes three experiments that explore ways to address the screen real estate and input problems.

III. Experiments

A. Switching between blocks mode and text mode

Internally, GP code is represented as abstract syntax trees that can be rendered easily as either blocks or text. Thus, the first experiment (inspired in part by D. Anthony Bau's Droplet Editor for Pencil Code [3], which we saw in 2014) is to allow the user to switch between blocks and text modes in place

[1] wikipedia.org/wiki/Deutsch_limit

978-1-4673-8367-7/15/$31.00 ©2015 IEEE

Fig. 1. The script for a gray-scale gradient in both blocks mode and text mode. The text mode version has been edited to remove the closing curly-bracket of the "for" loop, so it is no longer syntactically correct. The vertical line at the end of the text is the cursor.

Fig. 2. The same code viewed as normal blocks (top) and as "text blocks" (bottom). The downward arrowhead in the first line is a drop-down menu widget. The horizontal arrowheads can be used to expand blocks to show optional parameters.

(Figure 1). In text mode, one can type, delete, and position the cursor at the character level, as one would in any text editor. Clicking outside of the script parses the text and converts the script back into blocks—assuming it is syntactically correct.

Text mode addresses the screen real estate problem: text consumes less space than the blocks. With the current font choices, the vertical space required for the text is about half that of the blocks. Text mode also addresses the input problem: an expert can type code quickly without having to drag blocks out of the palette. Text mode also allows an expert to create blocks for GP functions that are not advertised in the palette, a "feature" that has sometimes saved us during demos.

However, we found that text mode has a number of disadvantages. First, it re-introduces the possibility of making syntax and spelling errors. Second, switching to text loses the interaction affordances we enjoy with blocks. For example, some blocks include drop-down menus. color picker widgets, or numeric input slots that can be "scrubbed" to adjust the input value; with text, one must type expressions or constants to generate the desired values.

Finally, editing code as pure text requires using the internal form of GP code, a LISP-like function call format that does not support inline keywords. Furthermore, one must use the internal function names. In the example, the internal function name for the statement in the loop body is "self_fillRect" rather than "fill rectangle x _ y _ w _ h _". Dropping inline keywords and exposing the internal function names is confusing enough in English, but it poses even bigger programs for translating GP blocks into other languages. A huge benefit of allowing block labels to be independent of the underlying function names is that one can render the same code into multiple spoken languages. Scratch supports over 60 languages, and scripts written in one language (say, Spanish) can viewed in another (say, Japanese). Scratch even flips blocks to support right-to-left languages like Hebrew and Arabic. We want the option of creating a Scratch-like translation system for GP.

B. Blocks that look like text

The second experiment is an attempt to combine the benefits of blocks with the compactness of textual code. The idea is to retain the graphical object structure of blocks code but change its appearance and layout by removing all borders and graphical ornaments except those needed to show structure, such as in nested subexpressions. This "text blocks" mode (currently controlled by a "blocks-text" slider in the UI that operates globally on all blocks and scripts) condenses the blocks into roughly the same screen real-estate as textual code and minimizes visual distractions, thus improving the readability of larger pieces of code. However, the blocks are still there and active. They can be dragged, dropped, duplicated and assembled in different ways. To make this clear, faint outlines of the block shapes appear as the mouse cursor hovers over them. The blocks also retain any interactive input widgets such as drop-down menus, color pickers, and, as seen in Figure 2, the arrowheads used to reveal optional block parameters.

With GPs current font and layout choices, a stack of blocks in "text blocks" mode takes up about half the vertical space as that same stack viewed as normal blocks. Surprisingly, text blocks code can also take less vertical space than its textual equivalent. When viewed in TextWrangler, a popular textual code editor, a 39-line GP "quicksort" method actually required 40% *more* vertical space than it did when viewed as text-blocks using the same font. The main reason for this is that TextWrangler uses a generous amount of space between lines, possibly to help programmers locate errors by line number, whereas GP's current line spacing is somewhat cramped. However, these are details; the key point is that code rendered as "text blocks" can be at least as dense as the equivalent textual code in a conventional code editor, and thus the Deutsch limit is not an issue.

Inspired by a demo of Etoys given by Alan Kay at the 2004 OOPSLA conference, we parameterized the transition between conventional blocks and text blocks. This allows us to provide a "blocks-text" slider so that the user can set the blocks appearance to any intermediate point along the blocks-text continuum, as Alan showed in his talk. It also allows the transition to be animated, as it is in PencilCode.

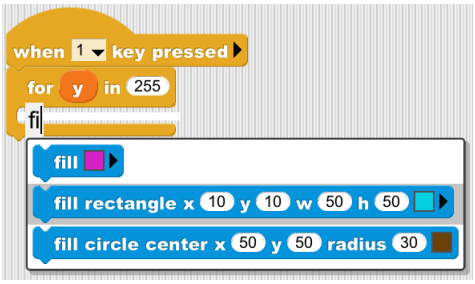


Fig. 3. Keyboard editing. A block is being inserted at the block editing cursor (white bar). The user has typed "fi" and the menu shows several possible matching blocks. The user can either type enough letters that the desired "fill rectangle" block is the only match or, as shown here, use the down arrow key to select the desired block. Once the desired block is selected the user can press the enter key to insert it.

C. Keyboard-based block editing

The third experiment explores ways to input and edit blocks code using only the keyboard. This effort was initially inspired by an interest in making GP accessible to users with visual or physical impairments, but we quickly realized that keyboard-based block editing addresses the input problem and thus benefits all GP users.

For this experiment, we added a movable "block editing cursor" to the scripts editor (Figure 3). The block editing cursor can be moved through all blocks in the scripts editor using the arrow and tab keys, and the block before the cursor can be deleted using the backspace key. Pressing a letter key allows the user to type the name of a block to be inserted at the cursor. [2] As they type, the system shows a short list of potentially matching blocks that is updated after every keystroke. Matches are determined by comparing the letters typed with the subset of blocks from the palette that would be syntactically correct at that input location. For example, when the cursor is in an input slot, only reporter blocks (expressions) are offered as possible matches. The enter key can be pressed to select and insert the top-hit in the match list, or the arrow keys can be used to select one of the other alternatives. This mechanism is similar to the auto-completion feature found in some textual code editors, although in this case the user must choose a valid block, whereas in a text editor the user can ignore the auto-completion suggestions and type something else.

Keyboard editing makes inputting blocks code much faster for experts. Features for experts can make a system less welcoming for beginners, but not in this case. A new GP user can easily ignore the keyboard editing features. They can explore the block palettes to discover what commands are available and can use drag-and-and drop to assemble blocks into scripts, just as they do in Scratch. However, keyboard editing may be useful even for a relative newcomer to GP, since it leverages the fact that recognition is easier than recall. The user need only remember (or guess) enough of a block name to make the desired block appear in the list of possible matches.

Keyboard editing supports translation to different spoken languages, since block matching is based on the (translated) block labels, not on the internal function names.

[2]Block matching was inspired by Snap's search-bar, originally prototyped by Kyle Hotchkiss (pull request #403 for Snap) and by Greenfoot3's frame editor by Michael Kölling, Neil C. C. Brown, and Amjad Altadmri [2].

IV. Reflection

These three experiments have provoked some reflections. The first experiment, converting between text and blocks seemed promising until we tried it. However, simply editing blocks code as text re-introduces the potential for syntax and spelling errors, loses the convenience of input widgets such as menus and color-pickers, and poses problems for block translation to other languages. The second experiment suggests that we can eliminate visual distraction and achieve the same compactness as textual code by changing the graphical appearance and layout. By retaining the underlying block structure, users still enjoy freedom from syntax and spelling errors, the benefits of structural editing, and the convenience of input widgets. The third experiment suggests that entering and editing blocks code can be done efficiently using only the keyboard. Furthermore, in contrast to the free-form text mode editing of the first experiment, keyboard-based blocks editing eliminates the potential for syntax and spelling errors and supports translation.

Of course, other projects have explored ideas similar to those discussed here, include StarLogo TNG [1], Greenfoot [2], and Pencil Code [3][4]. The Greenfoot paper includes an excellent discussion of other related work, including several structure-based code editors from the 1980's, Alice, and Touch Develop.

While blocks languages are a tremendous boon to beginners and casual programmers, experienced programmers often prefer text-based programming tools. While it is too soon to tell how well the techniques described in this paper—along with other techniques yet to be discovered—will serve programmers, it is our fond hope that experienced programmers may eventually find blocks programming environments more convenient and productive than the text-based tools they currently use.

References

[1] Corey McCaffrey. StarLogo TNG: The Convergence of Graphical Programming and Text Processing. Master's thesis, Massachusetts Institute of Technology, 2006.

[2] Michael Kölling, Neil C. C. Brown, and Amjad Altadmri. Frame-Based Editing: Easing the Transition from Blocks to Text-based Programming (to appear). In *The 10th Workshop in Primary and Secondary Computing Education (WiPSCE)*, 2015.

[3] D. Anthony Bau. Introducing the Droplet Editor, 2014. `https://youtu.be/PGDj1IzOtoo`.

[4] David Bau, D. Anthony Bau, Mathew Dawson, and C. Sydney Pickens. Pencil Code: Block Code for a Text World. In *ACM Interaction Design and Children (IDC)*, pages pp. 445–448, 2015.

Integrating Droplet into Applab – Improving The Usability of a Blocks-Based Text Editor

David Anthony Bau
Phillips Exeter Academy
Exeter, New Hampshire 03833
Email: dbau@exeter.edu

***Abstract*—Droplet is a programming editor that allows dual-mode editing in blocks and text for any text program. This paper presents observations and improvements to Droplet based on integrating Droplet into Applab, Code.org's JavaScript sandbox learning environment. Droplet's unique interactions with both text and blocks create several unusual problems and opportunities for improvement.**

I. Introduction

This paper describes a series of usability improvements for the dual-mode block/ext editor, [1]. Droplet provides a visual editing mode similar to Scratch [2], Alice [3], and Blockly [4]. However, Droplet is unique because it works as a text editor and provides a block interface on top of parsed text code. In previous work, Droplet the following features were implemented in Droplet:

- Blocks based on parsed text, allowing lossless conversion between blocks and text.
- A palette of short prewritten code fragments, represented as blocks.
- An editor supporting drag-and-drop assembly and editing of the blocks in a program.

Because Droplet is a text editor, many features of other block languages were initially unimplemented. For example, Weintrop [5] found that a key benefit of block languages is that the two-dimensional surface allows bottom-up assembly of code. Neilsen's usability heuristics [6] include user control and freedom through undo commands, as well as the need for guidance when choosing socket values. Finally, since Droplet can edit any text program, with the potential for errors, Dropet needs good support for identifying and recovering from errors.

This paper describes solutions for these four usability issues, as worked out in the context of integrating Droplet's JavaScript mode into Code.org's [7] Applab environment, and compares Droplet's approach, necessitated by its core identity as a text editor, to those taken by Scratch and Blockly, which are primarily block editors.

II. Background

A key motivation for Droplet is to allow students to edit any program with blocks. Droplet is built as a block editor framework that supports multiple text languages. Droplet's layout algorithm is designed to allow students to see source code in blocks the same way they would in text code. For example, text is always placed in the same rows in blocks as it appeared in the text source. This allows Droplet to achieve a smooth animation between blocks and text.

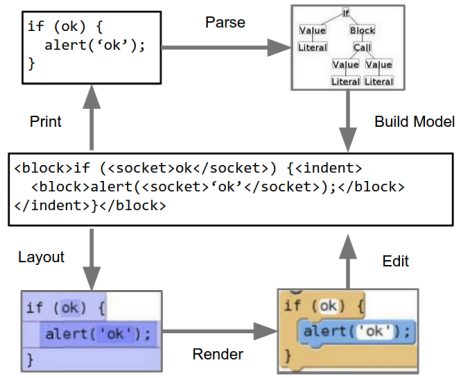


Fig. 1. Droplet's Lifecycle for a JavaScript Program

Figure 1 shows the lifecycle of a Droplet program in JavaScript. When the user opens a file, the language adapter for JavaScript runs the code through a standard JavaScript parser. It uses the resulting syntax tree to annotate the text stream to indicate the text ranges of blocks and sockets with tokens such as `blockStart` or `blockEnd`. The adapter also annotates information about color, shape, and droppability rules. Droplet then lays out and renders the resulting stream. When the user saves or runs the file, the markup is discarded to recover the original text.

III. Two-dimensional Editing

Two-dimensional editing surfaces, like Scratch supports, are beneficial to students, according to a study by Weintrop [5]. They allows students to try out different ways of performing the same task, and to compose programs in a "non-linear" way. Maloney et al. [8] refer to this as "tinkerability," and say that it supports "a bottom-up approach to writing scripts where small chunks of code are assembled and tested, then combined into larger units."

Both Scratch and Blockly support two-dimensional editing, but in different ways. Blockly runs all floating code in top-left to bottom-right order, while each Scratch block stack is associated with an event handler and runs whenever the attached event is fired. Scratch also runs a stack when it is double-clicked.

Two-dimensional editing is inconsistent with the linear nature of text programs, and our solution to this dilemma

978-1-4673-8367-7/15/$31.00 ©2015 IEEE

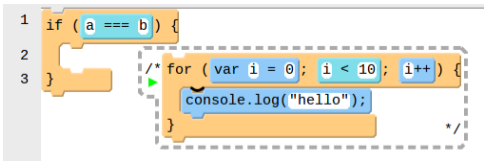


Fig. 2. An Example of Droplet's Floating Block Graphics

in Droplet is to allow the construction of "floating" blocks to the right of the main program, in the empty space in the editor. These are not executed when the program is run, and are surrounded by a dotted line and the comment symbol (fig. 2). Droplet displays a "play" button (fig. 2) that allows students to run individual stacks, but the stacks are not included in the main program.

In the future, Droplet may represent these blocks in the code by inserting them as comments. This would allow Droplet show an animation between the floating stacks in text mode and in block mode. Currently, floating blocks are lost whenever programs are converted to text or saved and loaded, since only the text code is saved.

IV. User Control and Freedom

Especially in untyped languages like JavaScript, it is easy to accidentally drop expression blocks into the wrong socket, so it is important for users to be able to recover from mistakes. Other block languages do not have this problem because sockets with information like variable names or long strings are not usually drop targets for other blocks. Scratch supports single-level undo, but neither Scratch nor Blockly supports a full undo stack. In contrast, the flexibility provided by Droplet needs to be balanced by robust support for recovery from mistakes. There are two interactions where recovery is helpful. One is when a block is removed from a socket, and the other is when the user wants to undo a previous action.

A. Remembering Old Socket Values

When a student accidentally drops a block in the wrong location, their natural reaction is to remove the block and continue dragging it to its intended location. To make editing smooth, and avoid requiring users to press the undo key, Droplet now restores old socket values to a socket whenever a block is removed from it (fig. 3). This occurs even after other alterations to the document, or after the socket has been moved.

Implementing this restoration requires a good locations model. Because Droplet frequently reparses blocks, attaching the remembered value data directly to the socket is not possible: the same socket instance may not exist after an editing operation has been done. Instead, Droplet maintains a map from socket locations to remembered values. However, the method by which locations should be serialized is subtle. The structure of a Droplet document can drastically change when Droplet reparses blocks, but the text of the Droplet document remains unchanged. This suggests that a locations should be based on text offsets.

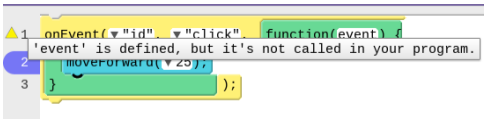


Fig. 4. An Example of Droplet Gutter Decorations in Applab

B. Full Support for Undo and Redo

The second natural action when a user makes a mistake is to use the undo command. Undo stacks are important to usability, and are included in Neilsen's widely-recognized user interface heuristics [6]. However, Droplet faced two obstacles in implementing full undo stacks for Droplet. First, because of Droplet's text-based affordances, Droplet had a large number of types of mutations, which were difficult to track and maintain. Second, Droplet did not have a way to unambiguously serialize the locations at which operations were happening.

Droplet mutations can be reduced to combinations of inserts and deletes. Locations, however, present a difficulty. A text-based location, like the remembered socket mechanism might use, is ambiguous when blocks or sockets are adjacent without intervening text. Because the undo stack would track reparses, the structure of the document when the location is retrieved is would be identical to that when it was serialized. This suggest that locations should be based on token offsets.

C. Resolving the Location Dilemma

To permit both socket value restoration and a full undo stack, Droplet uses two location models and converts between them as necessary. To assist this, Droplet has a third type of fundamental mutation: replace. A replace operation is used only for reparsing, and requires that the text content of the replaced section does not change. Droplet stores all locations as token-based offsets. When a replace operation occurs, Droplet converts any locations inside the replaced section that need to be persisted to text-based locations and converts back afterward. This allows Droplet to preserve socket locations across most reparses, but for the primary location model to be unambiguous.

V. Error Prevention and Recovery

A. Breakpoints and Line Annotations

Droplet allows users to work with arbitrary program text as blocks, which means that users can create runtime errors or code that deserves warnings. It is therefore important to support annotations and debugging tools. Line breakpoints and live annotations are a part of most major professional development environments, including Applab's text mode and Eclipse [9]. A study by Murphy [10] found that over 70% of Eclipse users use breakpoints. In 1986 Baecker [11] proposed "Metatext" or annotations as one of the five main principles of program visualization.

Applab had existing support for live errors and warnings and debugging breakpoints in text mode. Because Droplet blocks have a one-to-one relationship with text code, adding breakpoint and live line-annotation support to Droplet could easily take advantage of Applab's existing debugging infrastructure.

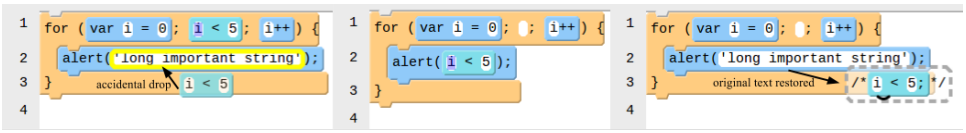


Fig. 3. An Example of Droplet Restoring Old Socket Values

Fig. 5. Droplet's New Behavior on Syntax Errors

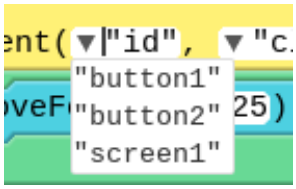


Fig. 6. An Example of Droplet Dropdowns in Applab

Droplet now supports breakpoints and annotations in the gutter the same way major text editors do (fig. 4). Droplet mimics Ace editor's API to allow Applab and other embedders to easily convert their existing debugging infrastructure from Ace editor to Droplet.

B. Handling Syntax Errors

Droplet allows users to type free-form text into sockets, which it will reparse on-the-fly and turn into blocks. This helps give students the experience of writing text without switching fully to text mode. However, it also means that users can create syntax errors by typing into sockets, unlike in other major block languages. Scratch and Blockly will only allow valid inputs in text areas. Droplet will now outline the violating input when a syntax error is created, and supports error annotations to help users identify the error (fig. 5).

VI. Recognition Rather than Recall: Dropdowns in a Text-Based Editor

Because all Droplet blocks are generated from text, Droplet did not have good support for dropdowns from sockets, which other major block languages do. Dropdowns, like autocomplete in text code, help students remember what parameters are valid, in accordinance with Neilsen's heuristic of recognition vs. recall [6].

Both Scratch and Blockly implement dropdowns for their text inputs. Both have special selectors for colors, allowing users to use a color picker or to "eyedrop" existing pixels on the screen.

Droplet added new configuration to allow the embedding application layer to specify dropdowns. Embedders may specify dropdowns by function name and argument position in JavaScript and CoffeeScript mode – for instance, in Figure 6, the "fd" function has a dropdown specified at argument 0. Dropdowns can be dynamically generated – in Figure 6, a list of element ids is generated using information taken from Applab's WYSIWYG HTML Design Mode.

VII. Acknowledgements

The author would like to thank Code.org for their support of this work, and Sarah Filman at Code.org and David Bau at Pencilcode for their advice.

References

[1] Bau, D. A. Droplet, A Blocks-Based Editor for Text Code. Journal of Computer Science in Colleges. 30, 6 (June 2015).

[2] Scratch. https://scratch.mit.edu/. Retrieved July 24th, 2015.

[3] Alice. http://www.alice.org. Retrieved July 24th, 2015.

[4] Blockly. https://blockly-games.appspot.com/. Retrieved July 24th, 2015.

[5] Weintrop, D. and Wilensky, U. To Block or Not To Block, That is the Question: Students' Perceptions of Block-based Programming. IDC '15 proceedings (June 2015).

[6] Nielsen, J. (1994). Heuristic evaluation. In Nielsen, J., and Mack, R.L. (Eds.), Usability Inspection Methods, John Wiley & Sons, New York, NY

[7] Code.org. http://code.org. Retrieved July 24th, 2015.

[8] Maloney, J., Resnick, M., Rusk, N., Silverman, B., and Eastmond, E. 2010. The scratch programming language and environment. ACM Trans. Comput. Educ. 10, 4, Article 16 (November 2010), 15 pages. DOI = 10.1145/1868358.1868363. http://doi.acm.org/10.1145/1868358.1868363.

[9] Mars Eclipse. http://eclipse.org. Retrieved July 24th, 2015.

[10] Murphy, G. Kersten, M. and Findlater, L. How Are Java Software Developers Using the Eclipse IDE? IEEE Software (July/August 2006) 72-82.

[11] Baecker, R. and Marcus, A. Design Principles for the Enhanced Presentation of Computer Program Source Text. CHI '86 proceedings (April 1986).

Position Paper: Lack of Keyboard Support Cripples Block-Based Programming

Neil C. C. Brown
School of Computing
University of Kent
Canterbury, UK
nccb@kent.ac.uk

Michael Kölling
School of Computing
University of Kent
Canterbury, UK
mik@kent.ac.uk

Amjad Altadmri
School of Computing
University of Kent
Canterbury, UK
aa803@kent.ac.uk

***Abstract*—Block-based programming is very popular with beginners, but it has failed to gain traction among intermediate and expert programmers. The mouse-centric interfaces typically found in block-based programming environments make edit interactions (especially in large programs) tedious and awkward. We propose that adding keyboard support is a key step to extending the applicability of block-based programming ideas and would allow their use by intermediate and expert programmers, extending some of their benefits to new user groups. We describe an implementation of this idea, 'frame-based programming', which leads to a number of benefits in error avoidance and edit efficiency.**

I. Introduction

Block-based programming has gained significant popularity in the last ten years. The success of systems such as Scratch, Snap, StarLogo TNG, Blockly, App Inventor, Alice and many more demonstrate the great interest in this programming style, especially for young age groups (pupils between eight and twelve years old). This success, however, has been restricted to these young learners and is not mirrored in older age groups. It is interesting to consider why this is. After all, many of the advantages that block-based programming offers – easier manipulation, freedom from syntax errors, reduced memorisation of syntax and commands – would be of benefit to all programmers.

The reluctance of more experienced or ambitious programmers to adopt this programming style is rooted in a number of fundamental limitations of block-based programming systems, most of all those affecting the ease of manipulation and systematic organisation of the program. In this paper, we argue that the lack of keyboard support for program manipulation in block-based programming systems causes many of the weaknesses that prevent scaling to more proficient use. We describe how adding support for keyboard controlled manipulation can remove many of the hurdles and make some of the advantages of block-based systems available to more proficient programmers.

Our design of this novel interaction method, called frame-based programming, not only improves on block-based systems, but may also lead to increased efficiency for expert programmers compared to working with text-based systems.

II. Advantages of Block-Based Programming

Block-based programming introduces several significant advantages over text-based programming which have the potential to provide benefits not only for young learners, but for all programmers. The pre-determined, uneditable structure of the blocks prevents syntax errors. In text-based programming, keywords of the language as well as syntactic structures, such as brackets, parentheses and punctuation, have to be recalled and can be mistyped or mismatched. Slips and small syntax errors distract and slow down developers and require significant mental effort for novice and intermediate programmers. In block-based environments, these errors are impossible to make, and no mental or manual effort has to be expended in memorising, recalling or typing these structures.

Some type errors are also prevented: many blocks mismatched in type or semantics cannot be snapped together, avoiding potential errors that otherwise would have to be detected and fixed. For example, an expression block cannot be inserted where a statement block is expected. While the severity of these errors decreases with increasing proficiency of the programmer, it is obvious that preventing these errors entirely is preferable to allowing such mistakes.

Blocks make it easier and quicker to create and manipulate entire syntactic constructs. Adding a fully-formed if-statement or a loop using a single gesture is faster than typing the whole construct as text. A statement can easily be dragged as a single entity and dropped into a new, valid location. In a text-based environment, the statement must first be selected, either using a combination of keyboard shortcuts or by selecting with the mouse, and then dropped at a carefully chosen location. Both of these parts are more tedious and error prone than with blocks: When selecting, it is easy to miss part of a compound statement, or to accidentally include or omit trailing line break characters. When dropping the selection, it may be placed into syntactically invalid locations. Blocks have no such issues, and deletion is similarly faster and less error-prone.

Block-based editors also remove a lot of lower-level tedium or trivial style decisions. There are no issues regarding depth of indentation, whether to use tabs or spaces, or the correct placement of curly brackets. Many other similar style issues become inapplicable.

978-1-4673-8367-7/15/$31.00 ©2015 IEEE

Fig. 1. Complex expressions in Scratch are written by dragging many individual blocks together. This expression is composed of eight blocks.

Many of these advantages would benefit proficient programmers as well as beginners. However, block systems as a whole do not scale sufficiently to proficient programmers' style of work and size of programs. For anyone but beginners, the limitations of block-based systems outweigh their advantages.

III. Limitations of Block-Based Programming

One of the main limitations in block-based programming is the speed of entry and manipulation. Although it is *easy* to drag blocks, it can also be laborious and time-consuming. Performing a relatively small calculation such as the hypotenuse of a triangle (e.g. for the distance between two objects), $\sqrt{x \times x + y \times y}$ involves assembling eight blocks (as shown in Figure 1). Each block requires a sequence of gestures: finding the appropriate palette, selecting and dragging of the prototype, to dropping at the target location. In a text-based editor, the equivalent code entry requires 13 keypresses, representing significantly less interaction effort. Text-based approaches to formulas have been shown to be more usable than purely block-based approaches [1].

The lack of expressive flexibility in what is being dragged also causes slower manipulation. There is no easy way to select, for example, two or more adjacent blocks in the body of a control structure and drag them elsewhere. The contents must be unpicked by individual drag gestures, or a complex set of drags to detach and subsequently re-attach the trailing blocks which the user did not wish to manipulate.

The effort required to create or edit a program is directly related to its size. Larger programs require more entry and manipulation, and slowly the balance tips: text-based programming becomes the easier medium to write and maintain large programs. This is the key aspect which limits the use of block-based programming by intermediate and professional programmers: the ease is outweighed by the lack of speed. Here, ease refers to the low cognitive load and motor skills required to plan and execute the operation, while speed is how long the operation takes. For example, entering the expression $(1+2)\times(1+2)$ into a calculator is easy, but for those proficient in arithmetic it is faster to calculate the answer mentally.

Block-based programming also tends to have poor support for code navigation. While the free placement of program code segments (e.g. event handlers) in many block-based environments is very flexible, it often leads to disorganisation. Navigation of code written by other people – an activity rarely performed by novices, but frequently by experts – is not well supported. Key navigation of larger systems, such as jumping (both ways) between the usage and definition of an entity, is expected in professional programming environments but usually unsupported in block systems.

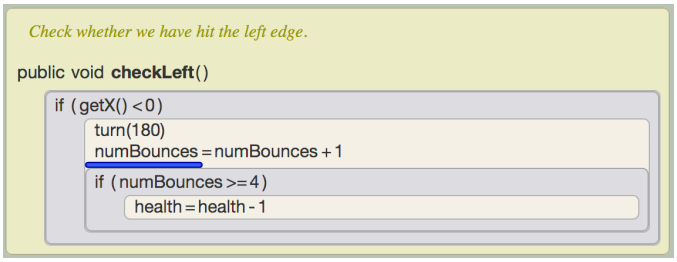


Fig. 2. The frame cursor is a thin horizontal blue line which occupies a small vertical space between frames, in the same way that a text cursor occupies a small horizontal space between characters.

IV. Frame-Based Programming

Frame-based programming is our own design of source code manipulation, combining aspects of blocks and text-based programming. One of its significant features is a combination of block-like entities – frames – with support for keyboard entry and manipulation. Keyboard control encompasses two elements: statement-level key support via a *frame cursor*, and expression-level support via *slots*.

A. Frame Cursor

Entering a new frame (akin to a statement-level block) requires two choices from the user: which frame to add, and where in the program code to add it. The former is straightforward, with different frames bound to different keys. The latter is achieved using a frame cursor. Just as textual entry on a computer involves a text cursor (or caret) indicating where the typed characters will be inserted, frame entry involves a frame cursor indicating where a new frame will be added.

The frame cursor occupies a small vertical space between frames (see Figure 2) and can be moved using the cursor keys. There are further shortcuts and modifiers: for example, pressing ctrl-up/down moves the cursor only at the current scope level, skipping the body of compound statements or entire methods. The frame cursor also allows selection: The user can use the shift modifier to select a contiguous block of frames, which can then be moved via mouse-dragging – or cut, copied, or pasted using standard keyboard shortcuts. Other keys perform logical actions: backspace deletes the current selection if there is one, or otherwise deletes the previous frame. The cursor can also be placed through clicking the mouse, and dragging can be used to create a selection.

There is only ever one cursor on screen: either a text cursor in a slot (see below) or a frame cursor. Because of this distinction, we can use single-key shortcuts for inserting frames when the frame cursor is selected. Pressing '`i`' inserts an if statement without the need for further modifiers. This makes entry of frames fast and convenient. To add a while-true loop, the user only needs to press five keys: `w t r u e`. The initial '`w`' creates a while loop and focuses the slot for the condition, and then '`true`' is typed into the slot.

B. Slots

In block-based programming, expressions are represented by blocks, with all the manipulation disadvantages discussed above. In frame-based programming, we allow expressions (e.g. conditions in loops or the right-hand side of assignments) to be entered textually with the keyboard.

Textual entry speeds up the creation of expressions, but it also re-introduces the possibility of syntax errors. The text is, however, not entirely free-form; paired symbols, such as brackets and quotes, are treated as a single entity. When the user enters an opening bracket, the closing bracket is automatically created at the same time. These brackets are paired forever: deleting one also deletes the other, and they can never nest incorrectly. Thus, some errors that may occur in traditional text editors are prevented.

Other tools typically provided in text-based environments are also provided in the frame editor: code completion, automatic corrections for mis-spelt variable names, real-time error annotations, etc. This allows the frame editor to support professional workflows and program sizes.

C. Navigation Improvements

Frame-based programming uses a structured presentation and layout for code much more similar to text than to the arrangement in block-based systems. All code is laid out vertically in classes, rather than freely placed on a larger canvas for each class. Specific segments in the code (such as field declarations, constructors, and methods) have a specified order and location. Navigation between elements (such as moving focus to the declaration of a variable, or showing the locations of uses of a variable) is supported via menu commands and keyboard shortcuts. The view pane of a frame-based class automatically scrolls to keep the frame cursor in view. Thus, the keyboard can be used to navigate the source code, by moving the frame cursor up and down the class.

V. Frames vs. Structured editing

Frame based programming is a specific variation of structure editing, an idea decades old, with a period of specific popularity and interest in the late 1980s and early 1990s. All structure editors have in common the principle that edit operations are performed on the underlying syntactical *structure*, not the textual representation on screen, and the goal of avoiding entry of many syntactically invalid programs.

Relevant examples of structure editors include GNOME [2], which used menus for entering low-level content (similar to our slots) with entry restricted to previously declared values, and Boxer [3], which used "boxes" instead of pure text – a construct that shares some aspects with our frames.

Existing structure editors managed to prevent many errors, but typically locked users into fixed workflows. Syntactically or semantically incorrect code could not be entered, even temporarily, preventing various methods of development and legitimate editing styles. This overly restrictive nature of many of the systems made them unpopular and led to failure to gain traction in the programming community.

Task	Scratch	Alice	NetBeans	Frames
Insertion	4.9	6.6	5.1	**1.6**
Modification	5.6	7.1	5.5	**5.0**
Deletion	5.4	2.6	7.8	**2.4**
Movement	5.5	**3.1**	6.0	4.8
Replacement	9.8	8.9	5.1	**2.3**

TABLE I
MEAN TIMES IN SECONDS (1 D.P.) FOR PROGRAM MANIPULATION TASK TYPES [4]. LOWER IS BETTER, BEST IN EACH ROW IS BOLDED.

In our frame editor, we do not prevent entry of many incorrect code segments, choosing instead to passively indicate erroneous code (via a red underline) without blocking the programmer from additional manipulations. Thus, we hope to provide better support while maintaining flexibility.

VI. Initial Results

An initial study evaluating the effectiveness of frame-based programming, using an earlier prototype of our editor [4], provides some first insights into its potential. The study compares cognitive models of different program modifications (insertion, modification, deletion, movement and replacement) in a prototype frame-based editor with various other systems, including Scratch, Alice, and NetBeans. Cognitive modelling computes a measure of task time by recording and analysing keystroke level interactions (such as key presses and mouse clicks) as well as "mental" operations (such as eye movement and reading time). The study was performed using CogTool, a software system that automates the recording and analysis of interaction sessions. The prototype frame-based editor was found to be the fastest in four out of five categories. Relevant results are reproduced here in Table I.

VII. Conclusion

The benefits of block-based programming have not yet been transferred to intermediate or professional programmers because the mouse-centric user interfaces make working with large programs too difficult. By combining aspects of block-based programming with keyboard support (along with several other innovations beyond the scope of this brief paper), frame-based programming removes the obstacles and extends those benefits to more proficient programmers.

References

[1] R. Koitz and W. Slany, "Empirical comparison of visual to hybrid formula manipulation in educational programming languages for teenagers," in *PLATEAU '14*. ACM, 2014, pp. 21–30. [Online]. Available: http://doi.acm.org/10.1145/2688204.2688209

[2] P. Miller, J. Pane, G. Meter, and S. Vorthmann, "Evolution of novice programming environments: The structure editors of carnegie mellon university," *Interactive Learning Environments*, vol. 4, no. 2, pp. 140–158, 1994. [Online]. Available: http://dx.doi.org/10.1080/1049482940040202

[3] A. A. diSessa, "Twenty reasons why you should use Boxer (instead of Logo)," in *Learning & Exploring with Logo: Proceedings of the Sixth European Logo Conference*, M. Turcsnyi-Szab, Ed., 1997, pp. 7–27.

[4] F. McKay and M. Kölling, "Predictive modelling for HCI problems in novice program editors," in *BCS-HCI '13*. BCS, 2013, pp. 35:1–35:6. [Online]. Available: http://dl.acm.org/citation.cfm?id=2578048.2578092

Thinking in Blocks: Implications of Using Abstract Syntax Trees as the Underlying Program Model

Daniel Wendel, Paul Medlock-Walton
Scheller Teacher Education Program
Massachusetts Institute of Technology
Cambridge, MA, USA
{djwendel, paulmw}@mit.edu

***Abstract*—This paper examines the implications of using Abstract Syntax Trees (ASTs) as the underlying model for program editors and source control. For editors, working at the level of the AST enables error prevention, efficient auto-completion, and seamless use of multiple representations (e.g. blocks-to-text-to-blocks). An AST-based system also lends itself to both real-time and asynchronous collaborative editing, through intention-preserving algorithms much simpler than Operational Transformations. AST-based asynchronous collaborative editing makes several improvements to source control compared to Git, notably: reducing conflicts even in same-position edits, and eliminating diffs (and therefore conflicts) due to changes in formatting, spacing, or method ordering. Even text-based languages can reap these benefits, simply by changing the underlying program representation from text to AST.**

Keywords—AST; blocks-based programming; multiple representations; real-time collaboration; source control

I. INTRODUCTION

In this paper we examine the implications of using Abstract Syntax Trees (ASTs) as the underlying model for program editors and source control. We first describe ASTs and their similarity to blocks. We then examine the implications of AST-based thinking on editor design. Finally, we examine the possibilities presented by ASTs in terms of real-time and asynchronous collaboration.

In an Abstract Syntax Tree [1], elements of the program are represented as nodes in a tree, with arguments and nested scopes being children of enclosing commands or structures. These children can either be named/ordered, as in the case of arguments/operands, or unordered, as in the case of methods in a class. In this form, whitespace, braces, tabs, semicolons, and even the particular names of commands/keywords are unnecessary, as the logical program structure is contained within the tree itself.

Interestingly, blocks-based programming languages already use a representation that is quite close to an AST. Consider an "if" block, for example. The block either exists, or does not; there is no "i" block created as an intermediate step, and when the block is created, it brings with it the knowledge that it requires one "test" expression and one "body" statement (or list of statements). If a "test" expression is provided, it links to the "test" section of the block; boolean expressions do not fit into the "command"-type socket where the "body" statement connects. If a user moves the "if" block from one part of the program to another, its "test" and "body" arguments move with it. Indeed, in the blocks-based languages we are familiar with, each block represents one node of an AST, and the arguments (sometimes called sockets, slots, or parameters) of each block represent the named, type-specific branches of the node.

The mapping from text-based languages to ASTs is less trivial, but obviously fully solved, as the concept of ASTs comes directly from text-based languages [1]. Compilation/interpretation in common languages proceeds not directly from a flat array of characters, but rather from an AST generated by a prior stage in the pipeline. In the other direction, modern debuggers are proof that mapping from compiled code (often with embedded metadata) back to text is also a fully-solved problem.

In fact, as discussed in Section 2, ASTs serve as an ideal intermediate format for switching between blocks and text representation. App Inventor's [2] use of S-expressions as an intermediate language when compiling apps is an example of a similar idea, at least in the blocks-to-text direction.

In the following sections we examine several benefits to using ASTs as the internal representation of programs, and of using AST nodes as the "thought unit" when thinking about editing and collaboration.

II. EDITING

Creating and editing code using AST nodes prevents users from creating syntax errors, allows atomic reordering of commands in a program, can enable efficient auto-completion, and enables users to view code with multiple representations. Modern IDEs such as Eclipse and IntelliJ already use ASTs to provide a variety of error-prevention and efficiency improvement features, but they do not operate on AST nodes as a "thought unit" as blocks-based languages do.

We use the phrase "thought unit" to mean that users are able add, remove, or modify nodes of the AST, but are not able to create partial versions of these nodes. Traditional text based programming environments do allow users to create partial AST nodes by typing incomplete expressions, which causes common syntax errors such as mismatched curly braces in nested "if" statements. Atomic AST node creation using blocks-based programming, typeblocking [3], or context-sensitive command selection prevents syntax errors by only

978-1-4673-8367-7/15/$31.00 ©2015 IEEE

allowing users to add and remove complete commands and expressions.

AST node editing also enables users to reorder nodes while maintaining a syntactically correct program. Blocks-based programming environments like Scratch [4], Blockly [5], and StarLogo [6] accomplish this by allowing users to place blocks in "stacks" on a workspace. Stacks then remain "connected" if dragged from the top block in the stack, allowing whole portions of code, AST sub-trees, to be moved at once. Reordering commands in text-based languages, on the other hand, requires bulk copy/paste or "move line" operations, which often result in incomplete—or worse, complete but logically incorrect—syntax, as braces or parentheses are accidentally left behind.

Context-sensitive command selection, implemented in both Greenfoot 3 [7] and Microsoft's TouchDevelop [8], provides the user with possible commands that can be connected at the location of the cursor. The cursor is located where AST nodes can be inserted, and users select from a list of possible commands that are available. This allows for efficient, auto-complete-based editing, as often only a few keystrokes are required to disambiguate between the available options.

Editing code using AST nodes also enables users to view multiple representations of the code. Greenfoot's editor can switch to a Java text based representation, while TouchDevelop has three different representation modes, one using blocks, another as easy to read text, and a third as a JavaScript-like language. Pencil Code's Droplet [9] editor, which uses an annotated parse tree not unlike an AST, uses an animated transition between blocks and text, making the integration between the representations truly seamless. And GP [10], a new language being demonstrated at this workshop, uses an AST model and parameterized representations to allow users to actually control via a slider the degree to which the program appears as "blocks" or "text".

III. Collaboration

Direct collaboration—working on the same copy of code—is a complex topic that most of our tools so far have only barely touched upon. None of the code.org tools support direct collaboration. Scratch [4], the most popular blocks-based platform, has a very effective community ecosystem designed around "remixing" and the sharing of ideas, but does not support co-ownership of projects. And StarLogo Nova [6], which does support co-ownership of projects, uses a locking mechanism to prevent more than one person from editing the project at a time. To our knowledge, only Zero Robotics [11] uses true real-time collaboration. In this section, we examine the ways in which an AST-backed model can enable collaboration both in real-time and asynchronously, more easily and more effectively than current systems for text-based collaboration.

A. Real-Time Collaboration

Real-time collaborative text editing has been a topic of research for many years, with Operational Transformations (OT) [12] taking over as the primary algorithm for maintaining consistency and causality in the 2000's. Notable editors using some form of OT include SubEthaEdit [13], Google Docs [14], Word Online [15], and Cloud9 [16]. While OT is considered to have solved the real-time collaboration problem, it is also considered to be complicated and difficult to implement correctly [17]. This barrier has prevented many tools from incorporating real-time collaboration, and even tools like Microsoft Office are only beginning to introduce these features now, some 25 years after OT's invention.

Additionally, real-time direct collaboration has made few inroads in programming, perhaps due to low perceived value. While Pair Programming has been widely studied and adopted, it uses a "driver/navigator" approach of trading (real or virtual) keyboard access, rather than giving both collaborators simultaneous access [18]. Tools like Cloud9 allow multiple simultaneous users to edit a program file, but, tellingly, the video on the marketing page shows one user typing comments while another types code. Indeed, it is hard to imagine two people being able to co-edit a text file in any way except turn-taking, due to the fact that ideas and even syntax are incomplete for a majority of the time spent editing, meaning that no incremental testing is possible until all authors complete their changes.

AST-based editing, especially in the context of a blocks-based editor, has the potential to overcome both the issue of difficulty and the issue of value.

1) Simpler Alternatives to OT Made Possible through ASTs

Since the original OT paper in 1989, the algorithm has been modified in many ways to patch flaws and to add stronger guarantees about system behavior with regards to user expectations [19]. Meanwhile, though, alternate approaches have also been proposed, promising similar consistency and intention-preservation, but with much simpler models and implementations [17, 20, 21, 22].

Three approaches in particular are particularly relevant to AST-based systems: Wantaim [20], WOOT (WithOut OT) [21], and Commutative Replicated Data Types [22]. Each of these approaches is based on the insight that the primary source of complication with OT is that edit operations identify their locations based on an index into the total array of characters, which can change depending on edits by other users. By treating data as a linked structure of uniquely-named nodes, the new approaches eliminate most of the complication involved in OT. Conveniently, ASTs can be implemented exactly that way - as a linked structure of uniquely-named nodes.

To further simplify things, platforms like Meteor [23] and Google Drive Realtime API [24] already exist, which are designed to abstract away the process of model synchronization. In order to use such a system, all that is required of the editor is that it must update the UI in response to changes to the model. While such a solution might lack some of the intention-preserving features of an application-native, OT-like system for co-located edits, it would nevertheless enable useable collaboration with very little work on the part of the blocks editor creator.

2) Value of Real-time Collaboration in Blocks-based Environments

While real-time collaboration is clearly simpler to implement in an AST-backed editor than in plain text, what is less obvious is that the value of the collaboration itself may be higher in a block-based environment than in a text one. This stems from two important features of blocks-based editors: syntax atomicity and built-in scratch space.

Syntax atomicity means that a blocks-based program is not syntactically "broken" in intermediate states. That is, even as a user is editing, the compiler always knows the meaning of each syntactic node in existence. Some languages, such as Scratch [4], go one step further and even fill in default values for empty arguments, meaning that the program remains in a valid, executable state at all times. This is clearly not a feature of traditional text environments, and is unique to blocks-based or AST-based systems.

Built-in scratch space is also a uniquely blocks-based idea. In many blocks-based editors, the exact location of the blocks is insignificant, and spare or stray blocks or stacks of blocks can be scattered around the page without affecting the program execution. Only when a block is attached to a "root" block (such as an event block) does it become a part of the program and begin to execute. For this reason, changes or additions to a particular section of code can easily happen "all at once," as new, complete code is moved from scratch space into the main stack of blocks. Indeed, in studying the programming habits of Scratch users, [25] found that most tended to build their programs in this (uniquely blocks-based) "bottom-up" fashion.

Taken together, these features mean that blocks-based code remains in an executable state during most edits. While changes to system-wide designs will of course prevent system-wide execution while they are being completed, small portions of code (for example, individual procedures) can still be run and tested independently, even with multiple authors editing multiple locations in the program. This makes the case for real-time collaboration more compelling in a blocks-based environment than in a traditional text-based one.

B. Asynchronous Collaboration

Asynchronous collaboration (e.g. source control) also stands to gain substantially from moving to an AST-based model. While Git [26] and similar systems are adept at enabling large groups of people to contribute to the same codebase, merge conflicts still exist and require structures outside the system (for example, social norms or additional check-in procedures) to ameliorate them. While multiple authors changing the same code in incompatible ways must still require intervention and resolution, two common sources of merge conflicts, from changes that are not actually conflicting, can be avoided by using ASTs: formatting conflicts and cut/paste conflicts.

Formatting conflicts occur for many reasons, but one of the most common in StarLogo Nova's underlying codebase (with which the authors are intimately familiar) is due to new developers making a change and then undoing the change (via CTRL+Z) in different editors. When the line is edited, some editors swap tabs for spaces (or vice-versa), creating a change that is invisible to the developer but causes conflicts with other, real changes to the same line. While any large team will use code conventions and software checks to enforce formatting compliance, such tools solve a problem that need not exist at all; formatting conflicts like these are by definition eliminated in an AST-based system, since spacing (and all other textual representations of syntax structure) is not even stored in the system. Indeed, code conventions could be rendered obsolete immediately, as individual preferences for spacing, line wraps, and even method ordering could be stored as preferences in a developer's profile, rather than in the code repository.

Cut/paste conflicts, or positional change conflicts, are another common source of frustration that could be automatically resolved in an AST-based system. For example, imagine an "if" block with a condition that checks a property of an object. One author may realize that another property also needs to be checked due to a corner case, and make that change. Another author may realize that the entire block needs to be moved one spot down, to ensure that a pre-condition is met, and so cuts and pastes the block. In text-based editors, these operations would result in conflicts at the line of the condition. But in an AST-based system, the edits are performed on different nodes (the "if" node and its condition node), and are not in conflict at all.

Clearly, an AST-based system can reduce reported conflicts from some of the common cases where no actual conflict exists. But what about the case where a true conflict arises due to logically incompatible edits? While ASTs alone do not solve this problem, combining them with an intention-preserving, real-time system would provide significant benefits through its operation log. Whereas diff-based systems merely report the end result of a change, operation-log-based systems allow rewind and playback of a particular author's changes, and some [27] even allow selective undo of past actions. This provides much greater insight and conflict resolution power than current text-diff tools allow.

IV. Conclusions

In this paper we have explored a few of the benefits of using ASTs as the underlying model for programs. These include error-reduced editing, real-time collaboration, and conflict-reduced asynchronous collaboration. Additionally, we have observed that blocks-based editors already use a model that is quite close to ASTs. This means that powerful new features should be able to be added to our blocks-based systems with perhaps less effort than we may have anticipated.

Additionally, blocks-based systems are not the only ones that can benefit from an AST model. Text-based languages too can reap the benefits of improved editing and source control management; in particular, the elimination of whitespace and formatting from source control seems to be a strict improvement.

Using ASTs as the underlying model for programming, regardless of whether the language is natively blocks- or text-based, will enable improvements to our tools and to our reasoning about our tools. While this paper is merely exploratory, highlighting potential areas of impact, our hope is that our community will further investigate and develop these ideas, and ultimately bring about a better experience for our

users that follows in the blocks-based philosophy of "getting to the hard fun part" of programming more quickly.

ACKNOWLEDGMENTS

The idea of representing programs as nodes in an AST rather than as blocks or as a string of characters, and the implications of this subtle difference in thinking, developed over several months through conversations with many people. In particular, we thank Michael Kölling of Greenfoot for the inspiration of frame-based editing, Evelyn Eastmond of Scratch for having a wedding attended by John Maloney of Scratch/GP and Matt Green, who first called the AST an "AST" rather than a "block tree", and many colleagues at MIT STEP (Eric Klopfer, Joshua Sheldon, Wendy Huang, Eli Kosminsky, and others) who helped to develop the ideas over many lunch breaks.

REFERENCES

[1] Joel Jones (2003). "Abstract Syntax Tree Implementation Idioms" http://www.hillside.net/plop/plop2003/Papers/Jones-ImplementingASTs.pdf

[2] MIT App Inventor. http://appinventor.mit.edu/explore/ retrieved Sept. 2015

[3] McCaffrey, Corey (2006). "StarLogo TNG: The Convergence of Graphical Programming and Text". MIT Master's Thesis, 2006. SSRN:http://ssrn.com/abstract=1639257

[4] Scratch. https://scratch.mit.edu/ retrieved Sept. 2015

[5] Blockly. https://blockly-games.appspot.com/about?lang=en retrieved Sept. 2015

[6] StarLogo Nova. http://www.slnova.org/ retrieved Sept. 2015

[7] Greenfoot. http://www.greenfoot.org/ retrieved Sept. 2015

[8] TouchDevelop. https://www.touchdevelop.com/ retrieved Sept. 2015

[9] Bau, D. A (2014). "Droplet, a Blocks-based Editor for Text Code". Whitepaper. http://ideas.pencilcode.net/home/htmlcss/droplet-paper.pdf retrieved Sept. 2015

[10] John Maloney. "GP". Personal demonstration at MIT STEP lab, August 28, 2015. Unpublished.

[11] Zero Robotics. http://zerorobotics.mit.edu/ retrieved Sept. 2015

[12] Ellis, C.A.; Gibbs, S.J. (1989). "Concurrency control in groupware systems". ACM SIGMOD Record (2): 399–407.doi:10.1145/66926

[13] SubEthaEdit. http://www.codingmonkeys.de/subethaedit/ retrieved Sept. 2015

[14] Google Docs. https://www.google.com/intl/en/docs/about/ retrieved Sept. 2015

[15] Microsoft Office Online. https://office.live.com/start/default.aspx retrieved Sept. 2015

[16] Cloud9. https://c9.io/ retrieved Sept. 2015

[17] Neil Fraser (2009). "Differential Synchronization". Whitepaper. http://neil.fraser.name/writing/sync/ retrieved Sept. 2015

[18] Williams, L. and Kessler, R. (2003). Pair Programming Illuminated. Boston: Addison-Wesley Professional.

[19] Chengzheng Sun; Xiaohua Jia; Yanchun Zhang; Yun Yang; David Chen (1998). "Achieving convergence, causality preservation, and intention preservation in real-time cooperative editing systems". ACM Trans. Comput.-Hum. Interact. (1): 63–108.doi:10.1145/274444.274447

[20] Edya Ladan-Mozes, Qian Liu, Jonathan Rhodes, Daniel Wendel (2006). "Wantaim - A Distributed Real-time Collaborative Text Editor". MIT 6.824. Robert Morris, professor. https://www.academia.edu/14441025/ retrieved Sept. 2015

[21] Gérald Oster, Pascal Urso, Pascal Molli, Abdessamad Imine. Real time group editors without Operational transformation. [Research Report] RR-5580, 2005, pp.24.

[22] Marc Shapiro, Nuno Preguiça. Designing a commutative replicated data type. [Research Report] RR-6320, 2007.

[23] Meteor. https://www.meteor.com/features retrieved Sept. 2015

[24] Google Realtime API: https://developers.google.com/google-apps/realtime/overview retrieved Sept. 2015

[25] Orni Meerbaum-Salant, Michal Armoni, and Mordechai Ben-Ari. 2011. Habits of programming in scratch. In Proceedings of ITiCSE '11. ACM, New York, NY, USA, 168-172. DOI=10.1145/1999747.1999796 http://doi.acm.org/10.1145/1999747.1999796

[26] Git. https://git-scm.com/ retrieved Sept. 2015

[27] Chengzheng Sun (2002). "Undo as concurrent inverse in group editors". ACM Trans. Comput.-Hum. Interact. (4): 309–361.doi:10.1145/586081.586085

Position Paper: Towards Making Block-Based Programming Accessible for Blind Users

Stephanie Ludi
Department of Software Engineering
Rochester Institute of Technology
Rochester, USA
salvse@rit.edu

***Abstract*—Block-based programming environments are not accessible to users who are visually impaired. The lack of access impacts students who are participating in computing outreach, in the classroom, or in informal settings that foster interest in computing. This paper will discuss accessibility design issues in block-based programming environments, as well as present research questions and current design revisions being undertaken in Blockly.**

Keywords—accessibility; aria; user interface design; visually impaired

I. Introduction

Block-based systems have gained prominence in recent years as a means of introducing novices to programming. In some cases, such as MIT's Scratch, online communities exist and the tools are integrated into pre-college computer science curricula (e.g. Exploring Computer Science, CSPrinciples) [2, 4]. As these systems have become components of curricula, after-school camps, and outreach, the lack of accessibility for many students with disabilities creates an obstacle for participation in these activities that are devised to increase participation in computing. In particular users with visual impairments are generally not able to use block-based tools unless they have enough vision to view the screen comfortably.

Visually impaired individuals may have some sight or have little to no vision. Assitive technology varies according to the degree of sight a person posseses. People who can read magnified text may use screen magnification software in order to zoom in on the contents of the screen by a specified magnification factor, adjust forground and background colors, or increase the size of the cursor. These users typically use the mouse alongside the keyboard. Individuals who are blind do not use the mouse. Instead navigation is typically via the keyboard (often through keyboard shortcuts). Screen readers (and for some refreshable Braille displays) are the means to access information that is displayed on the screen. When a website or program is designed correctly, the screen reader will read the content, including menus and other navigational elements. In the case of typical block-based programming environments, the screen reader reads no content or navigation.

This paper will present accessibility issues with the design of Block-based languages, as well as research questions being explored in an ongoing project. Users with visual impairments, including blindness are the focus of this paper.

II. Block-based Programming Environment Features that affect Accessibility

Each block-based project team makes design decisions for their project, that in some cases have unforseen consequences. Like any softwear project, the team prioritizes features based on a variety of needs that are considered. Examples of these design decisions are:

A. Technology and Platform Choices

The technologies used to develop block-based systems can have a large impact on accessibility of said systems to the disabled. The impact can be significantly positive or negative.

For example, Scratch 2.0 was developed in Actionscript/Flash. A positive implication is that Scratch runs in the browser, making installation seamless and thus easy to access for many people using various operating systems. At the same time the technology selection makes accessiblity impossible. In 2010, Adobe announced accessibility suport for Flash/Flex in terms of ARIA [3]. While ARIA is supported in terms of roles and states for HTML, Flash objects and actionscript does not support (ARIA) roles and states so presenting and interacting with a system is not possible. Another issue for screen reader users, who rely on keyboard shortcuts, is that Flash can override those keyboard shortcuts.

The Lego Mindstorm block-based robotics programming environment is traditionally a desktop application that uses LabView as the underlying technology. While the software runs on the Windows and Mac operating systems, the software is not compatible with screen readers. As a result, users who are blind will not hear anything.

On the positive side, Blockly is developed in CSS, SVG, and JavaScript. As such Blockly also runs on the web browser. However Blockly does not have the same accesibility issues as it can leverage ARIA (Accessible Rich Internet Applications) specification from the W3C's Web Accessibility Initiative. Developers need to adhere to the ARIA specification as compatibilty does not happen automatically for any web application. By following ARIA, a screen reader can read the structure of the webpage, labels on buttons and menus, as well

978-1-4673-8367-7/15/$31.00 ©2015 IEEE

as graphical objects (e.g. blocks). In addition, widgets can be described (e.g. slider, treeitem), keyboard navigation can be provided, as well as clearly articulated properies for drag-and-drop, widget states, or areas of a page that can be updated over time or based on an event. [6]

B. Mouse-centric Input

Many block-based systems rely on mouse input as the primary means of accessing features, including selecting blocks and adding them to programs, selecting attributes, and running the created program. For example, one cannot use the keyboard to locate, select and place a block onto the workspace in Scratch or Blockly as they were originally designed to be used with the mouse.

Many users with visual impairments rely on the keyboard to access software. As such, keyboard-focused commands and shortcuts are key to making interaction possible. Systems must be designed in order to utilize the keyboard as input in terms of menu navigation, programming, and accessing various panes in the programmng environment (e.g. changing focus to access specific information). In addition, the keys used to access features and information needs to be consistent with said standards and not conflict with keyboard shortcuts used by screen readers. In order to provide appropriate access to both sighted and visually impaired students, designing the system to allow for interaction via the mouse or keyboard is needed.

C. Feedback

User feedback in block-based programming environments are often visual in nature (e.g. a visual change on the workspace, pop-up messages, dialog boxes) without an audio feedback mechanism. Examples include a successful compilation or incompatible blocks that the user tries to connect together. Providing associated audio-based feedback can take various approaches depending on the nature of the feedback.

Students who use screen readers need to have all content including errors and any status messages provided audibly. Audio-based feedback can be in the form of speech or sound. As a pane or dialog box gets focus, the error or status text can be read (assuming it is programmed to enable a screen reader to access the text). Other feedback may be in the form of sounds (e.g. an audio icon or earcon) that correspond to meaning. An example of an audio icon is the sound of crumpling paper when a file is moved to the trashcan [1]. An example of an earcon is a tone or chord that is abstract in terms of the sound itself, but it is given meaning according to the association such as a deep sound may correspond to an unsuccessful download of the program to the robot [5]. The author is leveraging related work has been conducted to assess the use of audio cues in programming for programmers, though the study was conducted in a traditional, text-based programming environment [5].

D. Block-based Programs Created by the User

Each block-based program is designed for a particular purpose. Scratch programs can be in the form of animations or games. Lego Mindstorms NXT-G programs allow a robot to move and interact with its environment. Blockly programs can be translated into other languages such as JavaScript or Dart (though derivitives have been designed for specific domains such as music). As such, when a system is designed to be accessible, the devised programs themselves should be accessible to their users.

The audio capabilities of many block-based systems are limited, whether it be the ability for a robot to play a tone or recorded sound or for a Kodu game to play a sound effect when an event occurs. The audio capabilities, including the ability for a form of audio description when an animation is played or dialog is displayed is needed. Tapping into the location attributes or dialog text to enable compatibility with a screen reader is possible in many technologies (especialy JavaScript, Java, C++, C#).

III. Research Questions

Using the Blockly platform, the author and her team is re-engineering the system to enable use by users who are visually impaired. As such, our preliminary research questions are:

- Do visually impaired users want access to a block-based programming environment?
- How can Blockly be made accessible to the visually impaired, while at the same time remain usable to sighted users?
- What features will both visually impaired and sighted users appreciate?

IV. Design Implications

Modifying Blockly to provide access to users who are visually impaired is underway. The following sections provide a high-level view of the modifications to the system. The team uses only the JavaScript libraries that Blockly has originally used. The only exception to be made may be due to specific audio feedback that is being studied at this time. This addition would be a single JavaScript library.

A. Program Presentation

The visual and audio-based presentation of blocks and their content is fundamental. The key design revisons are:

- Allowing the user to change the highlight color for the current block or the connection point of a block in the workspace to improve discernability.
- Allowing the user to change the color of the workspace in order to minimize visual discomfort and improve readability.
- Enabling the blocks to be read on the workspace and in the toolbox (the menu where blocks are chosen)
- Visually linking blocks with any associated comments, where the user can jump from the block to the comment and back as desired.
- Providing a unique identifier with each block to enable visual and audio-based understanding of blocks.

B. Program Creation

Blockly requires the user to click on a block in the toolbox and place it in the desired place on the workspace using a mouse. Using the mouse, blocks can be moved as desired and attributes and parameters are set by selecting fields and entering a value via text or selection. This is typical of block-based programming environments. To make Blockly accessible, the following revisions are being implemented:

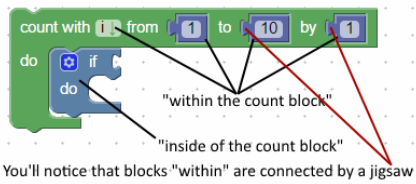
- The toolbox menu can be accessed and navigated via the keyboard.
- The desired block can be selected in the toolbox via a keyboard or mouse and it will appear at the insertion point on the workspace.
- Blocks can be moved within a program with a keyboard or mouse.
- Valid blocks that can be connected to a specified block are conveyed to the user visually and audibly.

C. Code Navigation

While program creation is critical, one tends to program incrementally in terms of adding to the features or complexity of the program, as well as fixing defects. As such, the need to enable visually impaired users to be able to navigate their code is critical. The features to facilitate the navigation of code are:

- Use of the keyboard to navigate between blocks and within a block (e.g. to fields), an shown in Figure 1.

FIG 1. Annotated View of a block with vertical and a nested block.



You'll notice that blocks "within" are connected by a jigsaw

- Provide each block (including vertical blocks) with a unique identifier to provide a visual and auditory structure for the program. The identifier is presented in the tree view (described in the next section).
- The use of audio cues in order to reinforce the level of nesting. A comparitive study is currently underway.
- Each block as well as each block part (e.g. field or inner block) can be read as a single block or in the program as a whole, depending on user need. When vertical blocks are read such as the 'count with' block presented above, the entire line can be read without all of the block identifiers if desired.

D. Comment Presentation

Program documentation is an afterthought to many programmers, but it is a feature that is in many block-based programming environments. Providing comments can help when a teacher wants to prepare lessons with starter programs or when a student may work on a program over time. While comments are text-based, activating the ability to add a comment requires the mouse in Blockly.

As part of redesigning the user interface, the user will have a box on the side of their screen to view comments in a tree view. The screen reader reads the comments as desired. The tree view will be updated automatically, and the structure will match that of the program using the hierarchy of identifiers associated with each block. In addition, a line will connect the current block with the comment line in the tree view.

Since the blocks in the workspace have their own identifying prefixes assigned to them (e.g. 1.1, 1.2) and the comment tree view will display the block prefix followed by that blocks comment if it has a comment. They will also have an info box that tells more information about the block they are currently on in a larger font for a better understanding of where they are in the structure. The user will be able to jump between a comment they have highlighted and a block in the workspace.

V. NEXT STEPS

The initial version of accessible Blockly should be completed during Fall 2015. The results of an audio feedback study that assessed the impact of various types of audio feedback modalities for code navigation and the understanding of nesting will be used in implementing code-based audio feedback during code navigation, in conjunction with the option for screen reader use, if needed. The accessibility features will be also studied in order to compare the usability impact for users with and without sight in order to ascertain what value may be found for sighted users as well as those who are visually impaired.

REFERENCES

[1] Dingler, T., Lindsay, J., & Walker, B. N. (2008). Learnability of sound cues for environmental features: Auditory icons, earcons, spearcons, and speech.Proceedings of the International Conference on Auditory Display (ICAD 2008), Paris, France (24-27 June).

[2] Goode, J. (2011, Summer). Exploring computer science: An equity-based reform program for 21st century computing education. *Journal for Computing Teachers*. Retrieved from http://www.iste.org/store/magazines-and-journals/downloads/jct-downloads.aspx

[3] Kirkpatrick, A. (2010) Adobe Accessibility / Flash Player and Flex Support for IAccessible2 and WAI-ARIA. Retrieded from: http://blogs.adobe.com/accessibility/2010/03/flash_player_and_flex_support.html

[4] Maloney, J., Resnick, M., Rusk, N., Silverman, B., Eastmond, E. (2010). The Scratch Programming Language and Environment. ACM Transactions on Computing Education, November 2010

[5] Stefik, A., Hundhausen, C., & Patterson, R.. An Empirical Investigation into the Design of Auditory Cues to Enhance Computer Program Comprehension. The International Journal of Human-Computer Studies, 69 (2011) 820-838.

[6] W3C-WAI (2014). WAI-ARIA Overview. Retrieded from: http://www.w3.org/WAI/intro/aria

Block-Based Programming Abstractions for Explicit Parallel Computing

Annette Feng,* Eli Tilevich,† Wu-chun Feng*†
*Department of Computer Science
†Department of Electrical and Computer Engineering
Virginia Tech, Blacksburg, U.S.A.
{afeng, tilevich, wfeng}@vt.edu

***Abstract*—With the majority of computing devices now featuring multiple computing cores, modern programmers need to be able to write programs that utilize these cores in parallel to extract the requisite levels of performance. Despite the need for such *explicit parallel computing*, few programmers are properly groomed in the mindset and the practices of parallel computing. Block-based programming languages, such as Scratch and Snap!, have proven to be a highly effective means of teaching fundamental programming concepts to a wide student audience. Nevertheless, the rich feature-set of mainstream block-based programming environments lack abstractions for explicit parallel programming, thus missing the opportunity to introduce this increasingly important programming concept at a time when the students' minds are most receptive. This paper reports on the results of an NSF-sponsored project for adding and integrating explicit programming abstractions, including producer-consumer, master-worker, and MapReduce, to block-based languages. We describe our reference implementation of adding the producer-consumer abstraction to Snap! and an educational project that utilizes this abstraction. This project clearly demonstrate the key features of parallel processing, without unduly burdening the programmer with the low-level details that this programming model typically entails. Our initial results show great potential in introducing the key concepts of parallel computing via block-based programming.**

***Keywords*—explicit parallel computing; computer science education; block-based programming; visual programming; parallel computational patterns**

I. Introduction

The traditional pillars of K-12 education are the so-called three *R's*: *R*eading, w*R*iting, and a*R*ithmetic. We postulate that the three R's above need to be supplemented with a fourth *R*: *R*easoning, or if you will, a renaissance in complex reasoning that embodies computational thinking. As complex (or higher-order) reasoning skills are now driving advanced economies, as shown in Figure 1), manual tasks and routine cognitive tasks are being increasingly automated. As a result, higher-order skills requiring complex reasoning and communication must become a major focus of educational strategies. Indeed, the College Board, in partnership with NSF, recently announced the fall 2016 launch of their new Advanced Placement Computer Science Principles course. In development since 2009 with funding from NSF, the AP Computer Science Principles course "is designed to broaden the number and diversity of students who participate in computing" and to empower them to "develop skills that will be critical to the jobs of today and tomorrow" [1], [2].

Many of these higher-order reasoning skills can be acquired in the context of computing, particularly parallel computing. Because computing has emerged as a third pillar of science, complementing the traditional pillars of theory and experimentation, it can accelerate discovery and innovation and create a fundamental change in how research, development, and technology transfer in the sciences, engineering, business, humanities, and arts will be conducted in the 21st century. For example, in a study conducted by the U.S. Council of Competitiveness, 97% of surveyed U.S. businesses noted that they could *not* exist or compete without the innovative use of high-performance parallel computing (HPC) [3]. Unfortunately, those same companies lament the dearth of a trained workforce that is familiar with parallel computing concepts.

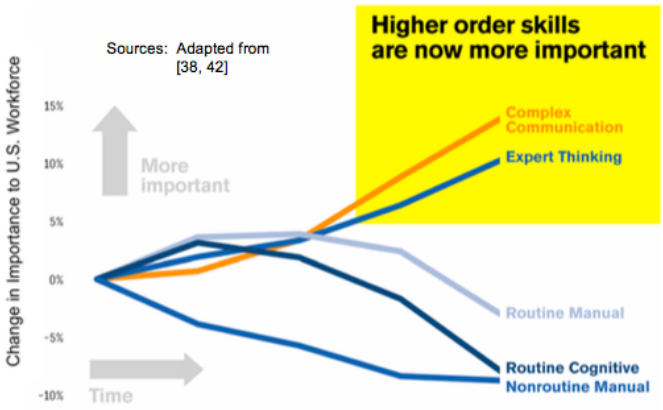


Fig. 1. Technological Changes Affecting U.S. Workforce Skills.

Block-based programming environments, such as Scratch [4] and Snap! [5], have been used effectively as powerful educational aids to introduce beginners to computing. We see the broad appeal of these environments due to the following two features. First, block-based languages have a very low barrier to entry. That is, students with no prior programming experience can quickly grasp the skills required to build programs that capture their interest, thereby motivating them to keep learning how to program. Second, block-based programming scales well with respect to students' ages and their level of programming experience. Block-based languages are expressive enough to support the ingenuity of quite advanced students of computing, while still providing enough basic blocks to provide a rewarding programming experience to novices. It is because of these properties of block-based languages that we see them as fertile ground for introducing parallel computing concepts to a wide range of computing students.

To address the need of improving the teaching of parallel computing concepts, we have been pursuing a project whose goal is to add explicit parallel abstractions to block-based programming languages. The thesis of this research is that the teaching of parallel computing does not need to be postponed until students have mastered the fundamentals of sequential programming. In fact, at this point,

978-1-4799-5944-0/14/$31.00 ©2014 IEEE

it may be too late to groom students to think truly in parallel. Instead, we posit that explicit parallel abstractions, such as producer-consumer, should be viewed as fundamental to programming as the `for` loop. By exposing explicit parallel programming via key language abstractions, we aim to harmoniously introduce students to parallel computing from the very start.

The rest of the paper is organized as follows. Section II covers background including discussion of the Snap! programming environment and concurrency paradigms. Section III presents work we've done to demonstrate the viability of this approach. Finally, in Section IV we present our conclusions and future work.

II. Background

Figure 2 shows the typical Snap! environment. Users program the behavior of actors called *sprites*, which appear on the *stage* area in the upper right. The *palette* area along the left side contains template blocks that users drag and drop into the *scripts* area in the middle. Users connect the blocks together linearly to form programs, which they do for each sprite in the application. When activated, the scripts run and the resulting output of behaviors of the sprites can be observed on the stage. In the screen shot shown in the figure, the sprites are the bees, the bears, the hives, and the honey jars. The displayed script defines the behavior of the bees, which is to perform a little "bee dance," make honey, and deliver it to the bears.

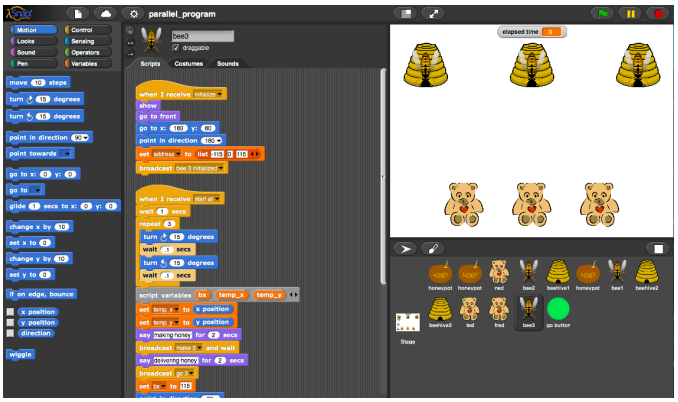


Fig. 2. The Snap! Graphical User Interface.

Each sprite has its own associated set of scripts which run simultaneously when the user clicks on the go button. In this manner, the system supports *implicit* parallelism, in that all sprites execute their scripts concurrently, with the control of the entire application being managed by the underlying JavaScript implementation. However, to code *explicit* parallel behavior that is purposefully coordinated using the Snap! code blocks themselves, while supported in Snap! in a cumbersome and rudimentary fashion, requires a knowledge of computer science concepts that are difficult for novice users to understand and apply successfully in order to correctly achieve the desired behavior.

Many different concurrency paradigms exist, with each pattern tailored to solving a specific kind of problem. For instance, the producer-consumer paradigm is used when the application produces multiple sets of data that must be processed in order, as with a program that handles network communication. The producer task receives incoming data packets from the network interface and stores them to a buffer where consumer tasks retrieve them for further processing. Because processing the data likely takes longer than acquiring it, placing the producer and consumer tasks in separate threads makes such an application much more efficient. Communication between the producer and consumer threads occurs via a shared, finite buffer implemented as a first-in, first-out (FIFO) queue. This creates a loosely-coupled system wherein the producers and the consumers do not need to know about each other beyond the producer knowing that something is taking the data out of the buffer to make room for more and the consumer knowing that something is putting the data in.

Indeed, the producer-consumer problem, also known as the bounded-buffer problem, is defined as one or more producer threads creating data and placing it in a shared and finite buffer, and one or more consumer threads removing the available data and operating on it. Constraints on the system stipulate that a producer with ready data must wait until an empty buffer slot is available before depositing the data, and a consumer ready for new data must wait until it is deposited in the buffer before retrieving it. Access to the shared buffer must be carefully orchestrated, i.e., synchronized, to prevent corruption of the data when multiple threads attempt to update it at the same time.

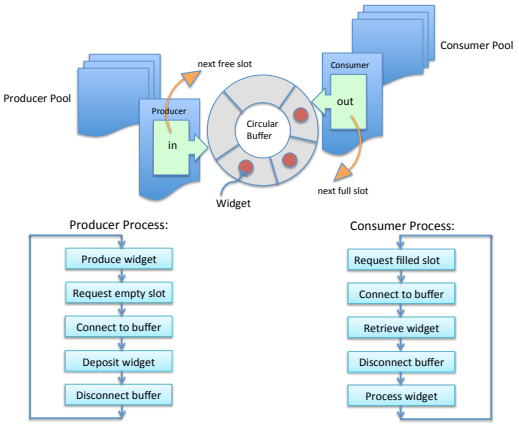


Fig. 3. Producer-Consumer Paradigm.

Figure 3 illustrates the producer-consumer paradigm. It shows the shared FIFO queue as a circular buffer with the head, or *in*, pointer maintaining where the producer is to insert the next data item, and the tail, or *out*, pointer maintaining where the consumer is to retrieve the next available data item to be processed.

Due to their complexity, explicitly parallel programs (in which the user himself defines the coordinating logic) are prone to errors and can be notoriously difficult to debug. Coding such parallel behavior requires a certain sophistication of logic and programming skills that more novice programmers generally do not possess, even though parallel behavior is easy enough to understand intuitively at a high level. We posit that teaching concurrency to computer science students is artificially delayed due to the cumbersome, low-level constructs that are currently available in commonly-used, text-based languages such as POSIX threads (Pthreads) in C. We propose that appropriate programming abstractions for a visual language such as Snap! would allow novice users to implement explicit parallel applications at a much earlier stage, before they become too rooted in the sequential way of thinking about programming.

It is the logical thinking required to produce parallel codes that we wish to promote by making the development of such programs more accessible and less error-prone to the novice user. To achieve this, we seek to provide the tools necessary for learning and employing these important concepts in an age-appropriate manner. Part of our approach involves utilizing a feature intrinsic to Snap! that allows the user to define and add his own code blocks to the Snap! programming environment and to define these new blocks using existing blocks. In computer science parlance, this capability is known as *encapsulation*. The basic idea behind encapsulation is to hide complexity, as it

tends to distract from our understanding of the larger system. As any developer can attest, ”Programs must be written for people to read” [6].

In the next section we present some of the high-level constructs that we have developed for explicit parallel programming in Snap!

III. Parallel Snap!

The first exposure to parallel programming for a computer science student would normally occur in a systems class and would involve writing a solution to the producer-consumer problem in C using POSIX threads (Pthreads), an industry standard for the C programming language for creating and manipulating threads. A typical Pthreads implementation, as shown in Figure 4, will follow along the lines of the producer and consumer patterns illustrated in Figure 3, but with additional code required to perform the task coordination necessary to ensure program correctness through a property called *mutual exclusion*. This property guarantees the condition that only one task at a time accesses the shared buffer.

The Pthreads code in Figure 4 implements the producer-consumer pattern appearing in Figure 3. The solution utilizes a shared, circular buffer and three indices: a `head` index that indicates the next slot for a producer to fill, a `tail` index that indicates the next slot containing data for the consumer to remove, and a `num_items` index that indicates when the buffer is either empty or full. Access to the shared buffer must be coordinated using a construct called a *mutex*, short for *mut*ual *ex*clusion, that implements the thread synchronization needed to protect the shared buffer from potential corruption. To write a more complete and general solution requires significantly more code and a greater degree of programming complexity.

Alas, programming similar explicit parallel behavior in Snap!, while feasible, is not a task achievable by novice users and requires more ingenuity on the part of the programmer to achieve than even the Pthreads version. In order to promote an ease of use of explicit parallel constructs that is on par with learning, say, looping mechanisms or conditional statements, we abstract out the low-level details that are not critical to understanding parallel behavior. These abstractions we present as basic building blocks that hide details not pertinent to understanding explicit parallelism in Snap!

Figure 5 through Figure 9 show a series of screen shots of the Snap! implementation of the producer-consumer problem. Figure 5 shows the program launch with the producer bees at home in the upper left corner of the stage, and the consumer bears at home in the lower right corner. In the middle is the shared buffer through which the bees and the bears communicate. The buffer shown is of size four (4). The white circle indicates an empty buffer slot, whereas a black circle indicates a full slot.

The next frame of the program, shown in Figure 6, shows two producers, having already acquired their ”honey data,” enroute to make their deposit to the shared buffer. As the producers near the slots, the slots sense their presence and a connection between a producer and a slot is made and the data is transferred. The buffer then signals the consumer that data is ready and assigns it the next filled slot. Note the values and locations of the data that the producers are depositing.

Figure 7 shows the producers on their way home after making their deposit, and two consumers already enroute to take the ”honey data.” This Figure 7 also shows that a third consumer, i.e., bear, is ready for data, but as no more buffer slots are ready, that consumer must wait.

In Figure 8 we see that the first two consumers have retrieved their data. Compare the value and location of the data items. In Figure 6 we saw that the producer with value 3 placed that item in the first buffer slot. In looking at Figure 8, we can indeed verify that the consumer retrieved data value 3 from the first slot, and in doing so, the buffer sensed the proximity of the consumer, transferred the data value, and changed its slot appearance from black to white to show that it is now empty and available for another item.

```c
#include <stdio.h>
#include <pthread.h>
#include <stdlib.h>
#include <unistd.h>

#define BUF_SIZE 5

int buffer[BUF_SIZE];
int num_items = 0, head = 0, tail = 0;
pthread_mutex_t mutex = PTHREAD_MUTEX_INITIALIZER;
pthread_cond_t produce = PTHREAD_COND_INITIALIZER;
pthread_cond_t consume = PTHREAD_COND_INITIALIZER;

void* producer(void *ptr) {
    int i, data;

    for (i = 1; i <= 2; i++) {
        data = (rand() % 10);                    // Produce
        sleep(data);
        printf("Produced: %d\n", data);
        pthread_mutex_lock(&mutex);              // Request
        if (num_items == BUF_SIZE) {
            pthread_cond_wait(&produce, &mutex); // Connect
        }
        buffer[head] = data;                     // Deposit
        head = (head+1) % BUF_SIZE;
        num_items++;
        pthread_cond_signal(&consume);
        pthread_mutex_unlock(&mutex);            // Disconnect
    }
    pthread_exit(0);
}

void* consumer(void *ptr) {
    int i, data;

    for (i = 1; i <= 2; i++) {
        sleep(rand() % 3);
        pthread_mutex_lock(&mutex);              // Request
        while (num_items == 0)
            pthread_cond_wait(&consume, &mutex); // Connect
        data = buffer[tail];                     // Retrieve
        tail = (tail+1) % BUF_SIZE;
        num_items--;
        pthread_cond_signal(&produce);
        pthread_mutex_unlock(&mutex);            // Disconnect
        printf("Consumed: %d\n", data);          // Consume
  }
  pthread_exit(0);
}

int main(int argc, char **argv) {
    pthread_t p1, p2, p3, p4;
    pthread_t c1, c2, c3, c4;
    int pid = 1, cid = 1;

    pthread_create(&c1, NULL, consumer, NULL);
    pthread_create(&c2, NULL, consumer, NULL);
    pthread_create(&c3, NULL, consumer, NULL);
    pthread_create(&c4, NULL, consumer, NULL);
    pthread_create(&p1, NULL, producer, NULL);
    pthread_create(&p2, NULL, producer, NULL);
    pthread_create(&p3, NULL, producer, NULL);
    pthread_create(&p4, NULL, producer, NULL);
    pthread_exit(0);
}
```

Fig. 4. Solution to Producer-Consumer Problem Using Pthreads.

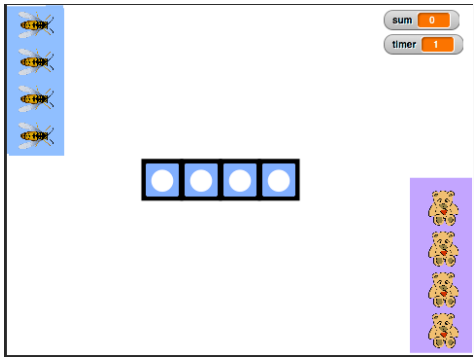


Fig. 5. Launch of the Snap! Producer-Consumer Program.

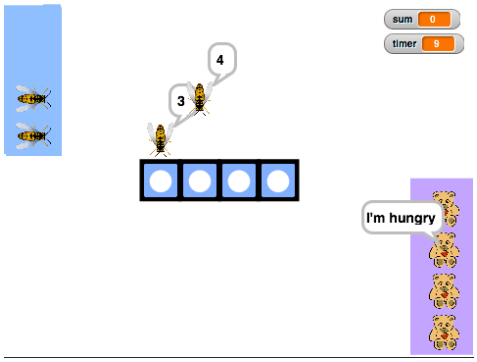


Fig. 6. Bees Making a Deposit.

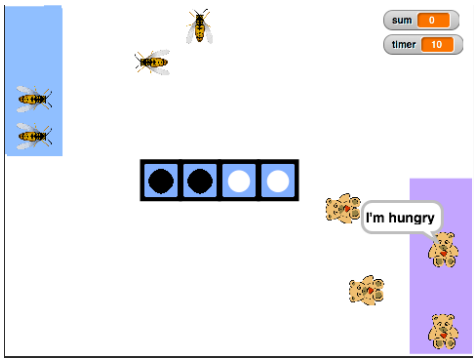


Fig. 7. Bears Retrieving "Honey Data."

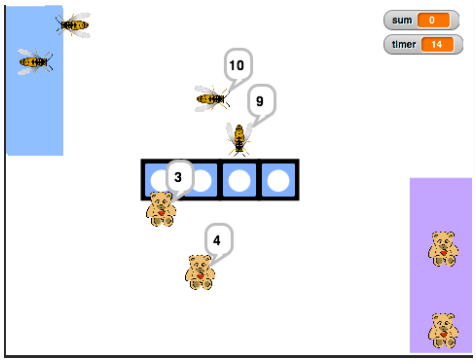


Fig. 8. Bees and Bears Coordinating Access to the Shared Buffer.

Figure 9 displays the final frame of the program showing the last consumers returning home to consume their data.

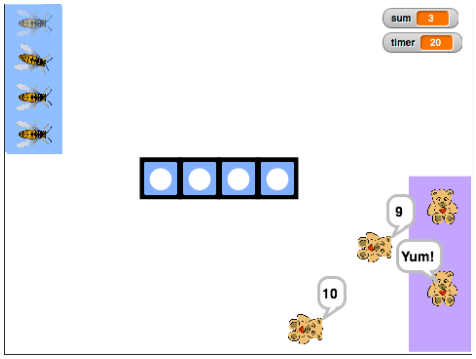


Fig. 9. Bear Consuming Honey.

Below we discuss the details of our approach for introducing explicit parallelism to Snap! When the program starts, the system is initialized, and when the buffer is ready, it signals the bees and bears to begin. The iterative behavior of the bees follows the normal producer pattern: (1) produce data, (2) request empty buffer slot, (3) acquire slot, (4) deposit data, and (5) repeat. The bears are similarly coded to realize the normal consumer pattern: (1) request filled slot, (2) acquire slot, (3) retrieve data, (4) consume data, and (5) repeat. The blocks implementing the basic solution for the bees and the bears are interspersed with code blocks that achieve the various animation effects. The potential for Snap! to visually illustrate parallel systems behavior through animation makes it particularly desirable as a teaching tool for children, given age-appropriate examples such as "the bees and the bears" demo.

Fig. 10. New "make buffer" Definition for Snap!

For the preceding example, the shared buffer has a capacity of four (4) slots; however this can be changed programmatically to accommodate any size buffer between 1 and 10. Figure 10 shows the special reporter block, `make buffer`, that was introduced to encapsulate the implementation details of the buffer. The `make buffer` block returns a buffer object containing five methods that can be called on it, namely

1) start
2) connect_producer
3) send
4) connect_consumer
5) receive

Figure 11 shows the code block that a producer with id `id` would use to send a `connect_producer` request to the buffer.

Fig. 11. Calling the `connect_producer` Method on the Buffer Object.

The shared buffer is a perfect example of the types of abstractions that we seek to develop for block-based languages such as Snap! The programmer need not understand the actual implementation details of the buffer in order to use it; the abstraction is at an appropriate

level and provides a general solution that can be reused in other programming scenarios.

IV. Conclusions and Future Work

Our work seeks to provide enhancements to the Snap! visual programming language that would allow novice programmers to learn about and to define more sophisticated behaviors in their applications using *explicit* parallel constructs at an earlier point in their programming careers. Our position is that current parallel constructs are not at a high enough level of abstraction to be accessible to the novice user and that this is an artificial barrier that should be eliminated.

Coordinating the complex interactions of parallel systems is non-trivial, and our initial work shows promise in our approach. As proof of concept, our implementation of the producer-consumer paradigm in Snap! successfully demonstrates that explicit parallel behavior can indeed be achieved in a block-based language such as Snap! The key to making it accessible to non-expert users is to provide abstractions at an appropriate level, such as the "shared buffer" object, and to make sure that such solutions are general enough to be useful in virtually any given programming scenario. A visual language such as Snap! has broad appeal and the costumes and animations give it the capacity to introduce an element of whimsy into any programming exercise.

However, one must be careful not to overlook the more serious applications that languages such as Snap! can facilitate. As an example, instead of the preceding "bees and bears" producer-consumer demo, what if we sought to implement, say, a package delivery system. Instead of bees and hives, we have human workers and warehouses, respectively, and instead of buffers and bears, we have delivery trucks and businesses, respectively. Now we are looking at the potential for modeling serious real-world, grown-up applications such as a product distribution network for a retail store or a supply-and-demand chain for a global enterprise.

V. Acknowledgement

The work was support in part by NSF ACI-1353786.

References

[1] The College Board, "College Board Officially Launches New AP Computer Science Principles Course to Increase Student Engagement in Computing," Dec 2014. [Online]. Available: https://www.collegeboard.org/college-board-officially-launches-new-ap-computer-science-principles-course

[2] ——, "College Board and NSF Expand Partnership to Bring Computer Science Classes to High Schools Across the U.S." June 2015. [Online]. Available: https://www.collegeboard.org/releases/2015/college-board-and-nsf-to-bring-computer-science-classes-to-high-schools

[3] E. Joseph and A. Snell and C. Willard, "Council on Competitiveness Study of U.S. Industrial HPC Users," July 2004. [Online]. Available: http://www.compete.org/pdf/HPCUsersSurvey.pdf

[4] M. Resnick, J. Maloney, A. Monroy-Hernández, N. Rusk, E. Eastmond, K. Brennan, A. Millner, E. Rosenbaum, J. Silver, B. Silverman *et al.*, "Scratch: programming for all," *Communications of the ACM*, vol. 52, no. 11, pp. 60–67, 2009.

[5] B. Harvey, D. Garcia, J. Paley, and L. Segars, "Snap!:(build your own blocks)," in *Proceedings of the 43rd ACM technical symposium on Computer Science Education*. ACM, 2012, pp. 662–662.

[6] G. Sussman, H. Abelson, and J. Sussman, "Structure and interpretation of computer programs," 1983.

Blocks In, Blocks Out: A Language for 3D Models

Chris Johnson
Department of Computer Science
University of Wisconsin, Eau Claire
Eau Claire, Wisconsin
Email: johnch@uwec.edu

Peter Bui
Department of Computer Science and Engineering
University of Notre Dame
Notre Dame, Indiana
Email: pbui@nd.edu

***Abstract*—Madeup is a programming language for making things *up*—literally. Programmers write sequences of commands to move and turn through space, tracing out shapes with algorithms and mathematical operations. The language is designed to teach computation from a tangible, first-person perspective and help students integrate computation back into the physical world. We describe the language in general and reflect specifically on our recent implementation of its block interface.**

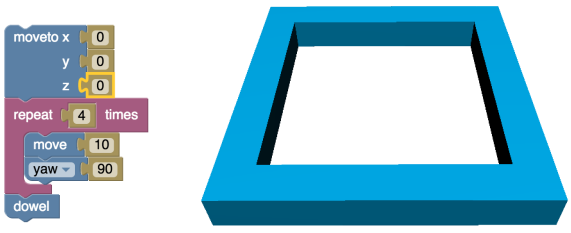
I. Introduction

Madeup is a programming language and development environment for teaching computer science and mathematics. Users write programs that walk along the skeletons or cross sections of geometric shapes and then issue commands to expand these paths into printable 3D models. Madeup was initially a text-based language, but now both blocks and text may be used to compose programs. The language facilitates an imperative and functional style of programming using standard expressions, conditionals, loops, functions, arrays, and turtle geometry commands. An example program is shown in Figure 1.

In this paper, we describe the Madeup language and our experiences in adding to it a block-based editor. In Section II we offer a detailed description of Madeup and some example programs. In Section III, we reflect on our implementation of a blocks interface. In Section IV, we summarize our work. Discussion of related work is integrated throughout the text.

II. Description

Madeup facilitates the generation of 3D models through a Logo-like imperative language and a browser-based development environment, the details of which we describe below. Introducing a new tool or language invokes a variety of reactions in the educational and technology community, and we also attempt to offer some justification for Madeup's entry into an already-crowded marketplace of educational tools.



(a) Madeup Source (b) Generated Model

Fig. 1. A frame generated by walking a square path and surrounding it with a square dowel structure.

A. Origin

Madeup was born out of a desire to generate 3D models in an algorithmic manner. Many 3D modeling programs provide excellent support for mouse-based sculpting or box modeling, but interaction with these programs tends to be driven by aesthetic feel and less by mathematical precision. Furthermore, altering an edit made several levels back in the undo stack is typically a destructive operation: in popping off the undo stack, the user loses the intervening edits. Reverting an edit made in a previous open-save-close cycle is usually impossible. In these programs, the user pays a high cost for mistakes made earlier but discovered later in the modeling process.

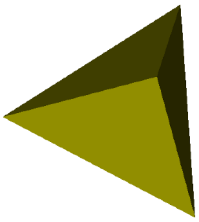
The code-based generation of Madeup solves both of these problems. First, Madeup narrows the scope of models that it can generate to "imperative shapes"—those that can be described by an imperative algorithm using mathematical and logical operators and flow control. Second, the entire process used to produce a model is expressed as a program instead of an ephemeral sequence of keypresses and mouse movement. New commands can be inserted anywhere in the program without eliminating subsequent commands. Parameters of existing commands can be tweaked without undoing or modifying others.

Other tools for programmatic modeling do exist. OpenSCAD and FormWriter [1], for example, provide textual scripting languages with which users may assemble complex models by combining simpler ones. These tools fit nicely in a professional workflow, but its mostly declarative language and its rich library of premade shapes reflects their primary purpose: to enable programmers to make shapes quickly. In Madeup, we are as interested in emphasizing *how* shapes are built as the shapes themselves. Several Logo-like languages exist for laser cutters [2], [3]. Given the constraints of this technology, programmers using these languages are restricted to 2D movement. The Beetle Blocks [4] project is quite similar to Madeup in spirit, but it supports fewer mesh generators and uses different mechanics for generating.

B. Mechanic

Most 3D graphics systems and 3D printers view a solid as a closed mesh of triangles. Internally, such meshes are represented as a list of vertex positions in Cartesian space and

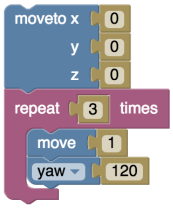
978-1-4673-8367-7/15/$31.00 ©2015 IEEE

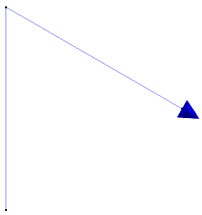


```
v 1 1 1
v -1 1 -1
v 1 -1 -1
v -1 -1 1
f 1 3 2
f 4 1 2
f 3 1 4
f 2 3 4
```

(a) Tetrahedron Model (b) Internal Representation

Fig. 2. A rendered tetrahedron and its internal representation. The internal representation is the OBJ model format, which lists the vertices as 3D positions and faces as 1-based indices into the vertex list.





```
0 0 0
0 1 0
0.866 0.5 0
0 0 0
```

(a) Triangle-walking code (b) Path after 2 moves (c) Path vertices

Fig. 3. A program that walks a triangular path is shown in (a). A preview of the path generated immediately after the second `move` command is shown in (b). The list in (c) is the final enumeration of vertices.

a list of triangular faces, with each face consisting of three integer indices into the positions list. Models of very simple shapes can be generated by hand, as was the tetrahedron shown in Figure 2. Mouse-based modeling programs do give users the power to add individual vertices to a model and connect groups of them into faces. However, most models aren't simple, and working with individual vertices is labor-intensive.

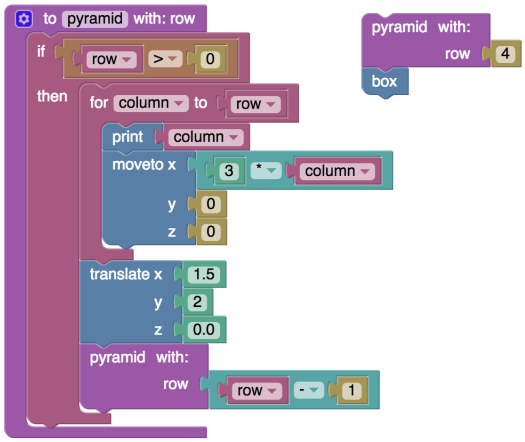
Madeup provides a simple mechanic to alleviate the burden of enumerating vertices. Instead of visiting each and every location on a mesh, the user walks a 1D path through 3D space using move and turn commands. An example is shown in the short triangle-walking program in Figure 3. The walked path is then interpreted in different ways by a handful of solidifiers to produce a solid model. Some solidifiers view the path as a cross section of the model to be expanded, others as a skeleton or list of centroids to be surrounded by solid geometry.

Vertex locations can be computed through standard expressions comprised of mathematical, relational, and logical operations; conditional logic; loops; array manipulation; and functional decomposition. A modest builtin library provides logarithmic, trigonometric, and other mathematical functions.

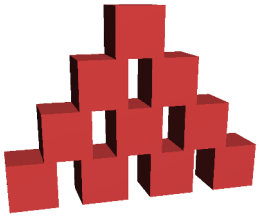
C. Solidifiers

Once the user has finished issuing `move` and `moveto` commands in a Madeup program, the visited vertices are given over to one of several solidifiers to produce a solid model. We describe a few of these supported solidifiers.

After a path has been used to generate a model, the list of vertices is cleared and a new path may be walked and used



(a) Madeup Source



(b) Generated Model

Fig. 4. A pyramid of boxes generated recursively. The first call to `pyramid` visits 4 locations, the next 3, and so on, up to the pyramid's top.

to generate a subsequent model. In this way, a single Madeup program may generate many models.

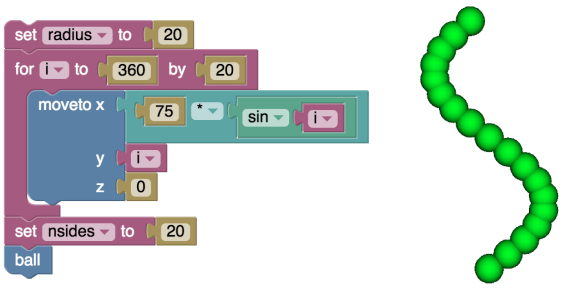
1) Box: The `box` generator surrounds each vertex in the walked path with a rectangular prism. The size of each box is determined by the value of the builtin variable `radius` at the time of the `move` or `moveto`. An example of this generator is shown in Figure 4.

2) Ball: Similar to `box`, the `ball` generator surrounds each vertex in the walked path with a faceted sphere. The radius of each sphere is determined by the value of the builtin variable `radius` at the time of the `move` or `moveto`. The number of facets on each sphere is determined by the value of the builtin variable `nsides` at the point where the `ball` command is issued. An example of this generator is shown in Figure 5.

3) Dowel: The `dowel` generator interprets the walked path as the skeleton of a filled polytube structure. The path is surrounded by geometry whose cross section is a regular polygon. The number of sides of the polygon is determined by the variable `nsides`. The radius of the polygon may change from vertex to vertex by assigning differing values to `radius`. An example of this generator is shown in Figure 1.

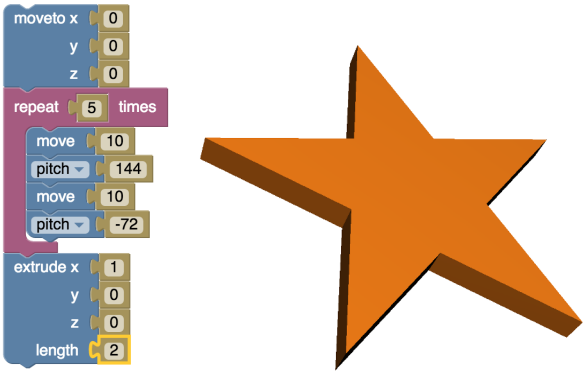
4) Extrude: The `extrude` generator interprets the walked path as the cross section of a solid to be extended along a given axis for a given length. An example of this generator is shown in Figure 6.

5) Revolve: The `revolve` generator interprets the walked path as the cross section of a solid to be revolved or spun



(a) Madeup Source (b) Generated Model

Fig. 5. A program that uniformly samples a sine wave every 20 degrees. The `ball` generator surrounds each vertex in the path with a sphere of a given radius.



(a) Madeup Source (b) Generated Model

Fig. 6. A program that walks a star path in the $x = 0$ plane. The `extrude` generator expands the path in a given direction for a given length.

around a given axis a given number of degrees. An example of this generator is shown in Figure 7.

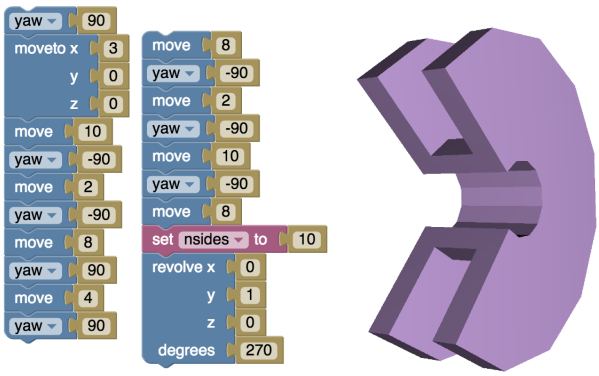
6) Surface: The `surface` generator interprets the walked path as the serialization of an implicit 2D grid structure. The visited vertices position the nodes of the grid, and these vertices join with adjacent nodes to form a surface. An example of this generator is shown in Figure 8, with the implied connectivity illustrated in the wireframe rendering in Figure 8c. This generator is useful for composing 2D parametric surfaces like cones, cylinders, spheres, planes, Möbius strips, Klein bottles, and so on. Unlike the other solidifiers, `surface` may produce models that are not actually solid.

D. Printing

Once the programmer's paths have been solidified, the models may be downloaded in an OBJ mesh format, which most 3D printing and computer graphics packages recognize. At present, many printers are managed by proprietary software, and we make no effort to connect to a printer directly.

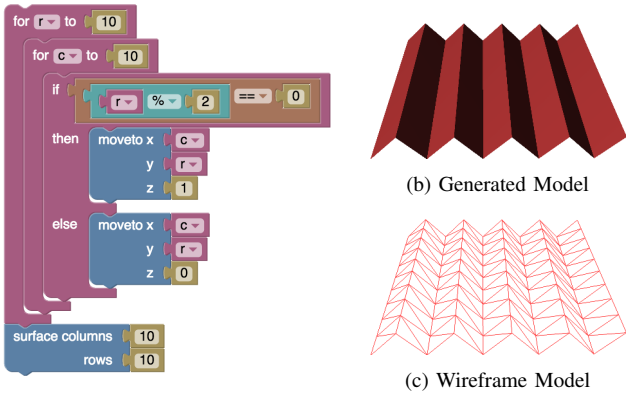
E. Education

As we began developing Madeup, we saw that it provided a context for tangibly exploring mathematics, algorithms,



(a) Madeup Source (b) Generated Model

Fig. 7. A program that walks a C-shaped path 3 units to the right of the y-axis. The `revolve` generator spins the path around the y-axis 270 degrees, stopping 10 times.



(a) Madeup Source

(b) Generated Model

(c) Wireframe Model

Fig. 8. A program that walks a C-shaped path 3 units to the right of the y-axis. The `revolve` generator spins the path around the y-axis 270 degrees, stopping 10 times.

technology, and other STEM-related disciplines. We informally validated its usability and motivational power through several workshops and camps for young learners.

Madeup joins a crowd of existing teaching tools, including Scratch, App Inventor, Pencil Code, and many others. What sets Madeup apart from existing projects is its physical product. The model that a programmer creates does not remain virtual. It can be printed, felt, carried in a pocket, and handed to a parent or friend—all of which may make computation real and relevant in the eyes of the programmer.

We believe the spatial domain is of particular importance to scientific learning. In fact, two specific spatially-oriented cognitive processes have frequently been found to associate with successful participation in STEM: mental rotation and cross section inference.

Shepard and Metzler [5] define *mental rotation* as the ability to imagine how an object would appear if rotated about an axis. An individual's performance on mental rotation tasks has been found to be a strong predictor of success in STEM domains [6]. Strong mental rotation skills have been found amongst dentists [7] and pilots [8]. Bodner and

Guay [9] observed chemistry students and found that spatial ability accounted for 15% of the variance on the students' exam scores. Interestingly, students with lower spatial abilities performed just as well as those with higher spatial abilities on problems requiring only memorization. However, on problems that required more advanced program solving, students with higher spatial abilities significantly outperformed their peers—even on problem solving exercises without a spatial component.

Cohen and Hegarty [10] define *cross section inference* as the ability to infer a 2D cross section given a 3D representation of an object. This skill has been found to positively correlate with success in anatomy [11], radiology [12], geology [13], [14], geometry [15], [16], and engineering [17], [18], [19].

Both rotations and cross sections play significant roles in the generation of solid objects in Madeup. The paths that a Madeup user programmatically generates are interpreted as cross sections that are extruded or rotated to form a 3D model. As its users reverse engineer existing objects that they encounter, they infer cross sections and mentally rotate them in the design process. We hypothesize that students' abilities in these two activities will increase significantly by using Madeup.

Stieff [20] demonstrated that when students have awareness of both spatial and higher-level, non-spatial problem solving strategies, the non-spatial strategies may not be activated until they have been contextualized in a particular setting. This suggests that spatial strategies like mental rotation are versatile and present a lower barrier to application. Thus, spatial abilities provide an important bridge that can lead to advanced understanding of many domains.

If spatial ability is a strong indicator of success in STEM, perhaps we can increase participation in STEM by improving learners' spatial abilities? Tuckey et al. [21] found a two-hour workshop on techniques for translating 2D representations to 3D representations yielded significantly higher exam scores. Sorby [22] administered a spatial training program for university engineering students, which boosted their scores on a spatial aptitude exam by an average of 27% and improved their first-year grades.

III. Blocks

A block interface was recently added to Madeup using the Blockly framework, and we reflect on our motivation and experiences in adding support for block composition of programs.

A. Embodied Learning

Madeup builds significantly on the work of Seymour Papert and the Logo project. In one of Papert's early implementations of Logo, students were supplied a physical robot that could be programmed to move and turn, optionally tracing its path on paper. This physical manifestation of Logo faithfully promoted what Papert called *body syntonic learning*, in which a learner comes to identify closely with an external manipulative. In such learning, an otherwise third-person observation by the learner effectively transforms into a more engaging first-person experience.

The Logo environment can also be simulated virtually. Instead of a physical robot, an onscreen turtle traces out the programmed path. Certainly, a simulated environment has administrative and economic advantages: there's less hardware to manufacture, assemble, and maintain. But are virtual environments a substitute for physical ones?

Eisenberg and Buechley [23] argue that for the power of computers to be fully realized and for us to fully express ourselves, computers' output must be more than digital and printers must produce more than ink on paper. Out of these beliefs came the Lilypad Arduino, a computing platform that can be integrated into one's clothing. Further, Buechley et al. [24] state that we often falsely limit technology to two applications: automation and entertainment. If we only view technology as a vehicle to simplify our lives and increase pleasure, we neglect a large portion of the human experience. They recommend that we also consider "technology as expanding and democratizing the range of human expression and creativity."

Djajadiningrat et al. [25] identify three distinct periods of our interaction with technology products:

- The electro-mechanical period prior to World War II, in which devices were controlled directly by levers and cranks, giving rich and direct feedback to their operators.
- The analog electrical period prior to the 1980s, in which dials and sliders gave operators only indirect control over their products' behaviors accompanied by indirect feedback.
- The digital electrical period of the present, in which our primary means of interacting with a device is pushing on its buttons or screen.

As human society has moved away from direct physical control of our devices, we have fewer opportunities to develop bodily skill. Instead, we place the burden of interaction on cognition, with which we are less likely to receive direct feedback and more likely to make errors.

Programmers write Madeup code in a development environment that draws upon both cognition and physical experience. The user interface, shown in Figure 9, provides continuous immediate feedback about the programmer's location in space and the direction she is currently facing. A preview of each path shows what locations have already been visited. By shortening the time between expression of a command and its execution, we expect Madeup users to engage more readily in syntonic learning.

Block-based interfaces recognize the importance of the physical world in the learning process. Composing a program with blocks is a metaphor for assembling a puzzle. Blocks are colored by kind. Execution flows from top to bottom through a sequence. If learning is made more accessible through an interface with physical elements, then will not learning and engagement be further enhanced by physical output? Madeup espouses the principal that as is the input, so should be the output. We hypothesize that the virtual focus of many block languages is holding back the long-term and widespread adoption of computer science education. By appealing to physical metaphors in the interface, we have made great progress in

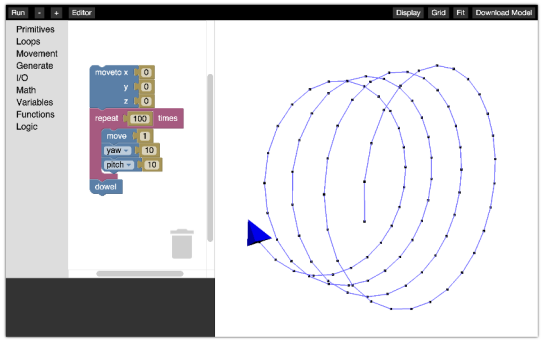


Fig. 9. The Madeup development environment. The left panel toggles between a block-based and text-based code editor. A preview of the paths generated so far appear in the canvas on the right, with the cursor's current orientation and location shown by an arrowhead glyph. Previews are shown in real-time as edits are made. The more computationally-intensive models are generated only when users hit the Run button in the top menu.

lowering the barrier to computational thinking. By further applying block interfaces to applications that produce enduring tangible output, we expect administrators and teachers to take our discipline more seriously.

Thornburg [26] identifies engineering and technology as "the glue that holds the other STEM subjects together." He holds that in the US participation in STEM is low because most K-8 schools do not have mechanical shops where young learners may tinker and build. We are sensitive to schools' limited opportunities to significantly alter their facilities, and we see Madeup and 3D printers as an affordable starting point to bring a shop-like setting into schools. These printers are small and contained, and do not require significant changes in infrastructure.

B. Statements vs. Expressions

Madeup supports a functional style of programming, with every construct of Madeup serving as an expression yielding some value. As we implemented a blocks interface for the language, we had to overcome what may be an inescapable feature of most block languages: they are geared toward imperative thinking.

In an imperative language, conditionals, loops, and procedure calls appear as statements. In imperative block languages, these constructs are represented with blocks having sequence connectors so that other statements may snap in before or after them. In a functional language, these may appear as either statements or expressions—they are *amphibious*. In many functional languages return statements are also amphibious, often appearing as a bare expression. As statements, such blocks need sequence connectors. As expressions, they need value connectors.

We considered four possible solutions to supporting amphibious constructs. Most involve altering how the blocks appear in the palette:

1) Augment blocks to simultaneously have both sequence and value connectors. This approach seems aesthetically displeasing.
2) Represent each amphibious construct in two ways: once as a statement block and once as an expression block, as demonstrated in Figure 10. If there are many amphibious blocks, this approach introduces clutter in the palette. At present, this is the option we have implemented for conditionals and function calls in Madeup.
3) Show the blocks as expressions with value connectors, but provide an expression-to-statement converter block. This is also implemented in Madeup, but we've only found it necessary to use for return expressions.
4) Show the blocks in one form or the other, but allow their connectors to be explicitly modified once placed in the editor. This approach manifestly deceives the programmer about a block's affordances.
5) Show the blocks in an undifferentiated format, but dynamically modify their connectors according to the context in which they are placed.

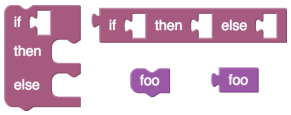


Fig. 10. Conditionals and function calls may appear as either expressions or statements in Madeup. Currently, we provide both forms in the blocks palette.

C. Cursor

A feature that text-based development environments exploit but block-based environments manage without is the notion of a cursor. In a text editor, a cursor provides location and context. The contents of the clipboard can be pasted at the cursor. A smart autocompletion system suggests only names in scope at the point of the cursor.

What could a block interface gain by adding a cursor? Currently, blocks are moved almost exclusively by dragging and dropping. This interaction is not accessible to all users. With the notion of a cursor, we can respond to double-clicking on a block in the palette by connecting it at the point of the cursor. The variables shown in the palette can be restricted to only those that are in scope at the point of the cursor. The cursor can be treated as a momentary breakpoint, and a "run until cursor" feature could be aided to aid in debugging and demonstration.

At least one blocks framework—Blockly—does support selection. (Scratch and Pencil Code do not.) Exactly one block may be selected at a time. This draws visual attention to the block, and it may be copied or deleted using keyboard shortcuts. The functionality of selection could be enhanced to provide context and aid in accessibility.

Selection by definition requires an existing block to select. A cursor on the other hand selects the "nothing" between blocks. Spreadsheet software is cursorless: to insert new cells, one must first select an adjacent cell. However, the insertion point is ambiguous, and the user must further specify whether the new cells should appear before or after the selection. A cursor is not ambiguous.

D. Sequential vs. Random Access Editing

Compared to a text-based language, statements in a block-based language are explicitly linked to one another. This tighter

coupling introduces higher editing costs compared to a textual editor. For example, to swap the order of two blocks, one must first detach any trailing blocks, drag the second before the first, and then reattach the trailing blocks. This mandatory consideration of the neighborhood of block-based code is analogous to editing a linked list: one must maintain the links. By representing code with text, we essentially gain random access and can swap two lines without disturbing any others.

Many editing operations are needed so infrequently that their higher cost can be ignored. However, as we implemented Madeup's blocks interface, one editing operation that we found missing from many block languages is that of temporarily disabling a statement by commenting it out. The importance of this mechanic is not superficial. When debugging an algorithm by commenting out portions, code must be fluid. We've seen users work around the lack of disable-by-comment in two ways:

- By physically breaking the block out of the flow. The cost of disabling a block is high enough to inhibit experimentation.
- By artificially embedding the block in an if-then statement. This enables fast toggling, but applying this to many blocks obscures the intent of the program.

We suggest that more block-based platforms enable temporary disabling of a block through a contextual menu—to keep the visible interface spare—and by maintaining for each block a status flag that can be checked by the code generator. This behavior is already available in Blockly, OpenBlocks, and App Inventor.

IV. Conclusion

We have introduced Madeup, a block- and text-based programming environment for generating 3D models. Using turtle geometry and other spatial commands, learners assume a first-person role in Papert's "Mathland" as they trace out cross sections and skeletons of shapes. Writing Madeup programs invokes cognitive processes that have been found to promote successful participation in STEM learning. The physical output of a Madeup program can help root computer science and mathematical concepts in the physical world.

Recently we added a block interface to the language, and we have offered our reflection on the user experience found in many block languages.

References

[1] M. D. Gross, "Formwriter: A little programming language for generating three-dimensional form algorithmically," in *CAAD Futures*, 2001, pp. 577–588.

[2] F. Turbak, S. Sandu, O. Kotsopoulos, E. Erdman, E. Davis, and K. Chadha, "Blocks languages for creating tangible artifacts," in *Visual Languages and Human-Centric Computing (VL/HCC), 2012 IEEE Symposium on*, Sept 2012, pp. 137–144.

[3] G. Johnson, "Flatcad and flatlang: Kits by code," in *Visual Languages and Human-Centric Computing, 2008. VL/HCC 2008. IEEE Symposium on*, Sept 2008, pp. 117–120.

[4] "Beetle Blocks," http://beetleblocks.com, [Online; accessed 5-July-2015].

[5] R. Shepard and J. Metzler, "Mental rotation of three-dimensional objects," *Science*, no. 171, pp. 701–703, 1971.

[6] J. Wai, D. Lubinksi, and C. P. Benbow, "Spatial ability for STEM domains: Aligning over 50 years of cumulative psychological knowledge solidifies its importance," *Journal of Educational Psychology*, vol. 101, no. 4, pp. 817–835, 2009.

[7] D. Moreau, J. Clerc, A. Mansy-Dannay, and A. Guerrien, "Enhancing spatial ability through sport practice: Evidence for an effect of motor training on mental rotation performance." *Journal of Individual Differences*, vol. 33, no. 2, p. 83, 2012.

[8] I. E. Dror, S. M. Kosslyn, and W. L. Waag, "Visual-spatial abilities of pilots." *Journal of Applied Psychology*, vol. 78, no. 5, p. 763, 1993.

[9] G. M. Bodner and R. B. Guay, "The Purdue visualization of rotations test," *The Chemical Educator*, vol. 2, no. 4, pp. 1–17, 1997.

[10] C. A. Cohen and M. Hegarty, "Inferring cross sections of 3D objects: A new spatial thinking test," *Learning and Individual Differences*, vol. 22, no. 6, pp. 868–874, 2012.

[11] J. Russell-Gebbett, "Skills and strategies–pupils' approaches to three-dimensional problems in biology," *Journal of Biological Education*, vol. 19, no. 4, pp. 293–298, 1985.

[12] M. Hegarty, "Components of spatial intelligence," *Psychology of Learning and Motivation*, vol. 52, pp. 265–297, 2010.

[13] Y. Kali and N. Orion, "Spatial abilities of high-school students in the perception of geologic structures," *Journal of Research in Science Teaching*, vol. 33, no. 4, pp. 369–391, 1996.

[14] N. Orion, D. Ben-Chaim, and Y. Kali, "Relationship between earth-science education and spatial visualization," *Journal of Geoscience Education*, vol. 45, pp. 129–132, 1997.

[15] E. H. Brinkmann, "Programed instruction as a technique for improving spatial visualization." *Journal of Applied Psychology*, vol. 50, no. 2, p. 179, 1966.

[16] M. Pittalis and C. Christou, "Types of reasoning in 3D geometry thinking and their relation with spatial ability," *Educational Studies in Mathematics*, vol. 75, no. 2, pp. 191–212, 2010.

[17] R. T. Duesbury *et al.*, "Effect of type of practice in a computer-aided design environment in visualizing three-dimensional objects from two-dimensional orthographic projections." *Journal of Applied Psychology*, vol. 81, no. 3, p. 249, 1996.

[18] H. B. Gerson, S. A. Sorby, A. Wysocki, and B. J. Baartmans, "The development and assessment of multimedia software for improving 3-D spatial visualization skills," *Computer Applications in Engineering Education*, vol. 9, no. 2, pp. 105–113, 2001.

[19] S. P. Lajoie, "Individual differences in spatial ability: Developing technologies to increase strategy awareness and skills," *Educational Psychologist*, vol. 38, no. 2, pp. 115–125, 2003.

[20] M. Stieff, "Mental rotation and diagrammatic reasoning in science," *Learning and instruction*, vol. 17, no. 2, pp. 219–234, 2007.

[21] H. Tuckey, M. Selvaratnam, and J. Bradley, "Identification and rectification of student difficulties concerning three-dimensional structures, rotation, and reflection," *Journal of Chemical Education*, vol. 68, no. 6, p. 460, 1991.

[22] S. A. Sorby, "Educational research in developing 3-d spatial skills for engineering students," *International Journal of Science Education*, vol. 31, no. 3, pp. 459–480, 2009.

[23] M. Eisenberg and L. Buechley, "Pervasive fabrication: Making construction ubiquitous in education," *Journal of Software*, vol. 3, no. 4, 2008. [Online]. Available: http://ojs.academypublisher.com/index.php/jsw/article/view/03046268

[24] L. Buechley, M. Eisenberg, J. Catchen, and A. Crockett, "The LilyPad Arduino: using computational textiles to investigate engagement, aesthetics, and diversity in computer science education," in *Proceedings of the SIGCHI conference on Human factors in computing systems*. ACM, 2008, pp. 423–432.

[25] T. Djajadiningrat, B. Matthews, and M. Stienstra, "Easy doesn't do it: Skill and expression in tangible aesthetics," *Personal Ubiquitous Comput.*, vol. 11, no. 8, pp. 657–676, Dec. 2007. [Online]. Available: http://dx.doi.org/10.1007/s00779-006-0137-9

[26] D. Thornburg, "Hands and minds: Why engineering is the glue holding STEM together," *Thornburg Center for Space Exploration. Retrieved from http://www.tcse-k12.org/pages/hands.pdf*, 2009.

A Blocks-Based Editor for HTML Code

Saksham Aggarwal
International Institute of Information Technology
Hyderabad, India
saksham.aggarwal@students.iiit.ac.in

David Anthony Bau
Phillips Exeter Academy
Exeter, NH, USA
dbau@exeter.edu

David Bau
Massachusetts Institute of Technology
Cambridge, MA, USA
davidbau@mit.edu

***Abstract*—This paper presents a block-programming editor for HTML code. The editor provides a block visualization of HTML syntax, allowing students to work in either blocks or text and switch freely. Our editor was created as an extension of Droplet, a dual-mode programming block editing framework that was previously used for JavaScript and CoffeeScript. We describe the process of extending Droplet to apply to HTML. We also discuss an analysis of real-world HTML tags and attributes and propose a palette based on this analysis.**

I. Introduction

Teaching HTML has long been an early step in a programming curriculum. For example Budny, et al [1] in Four Steps to Teaching C Programming, suggest "The layout of a web page allowed us to begin to teach the basic concepts of program layout... We are teaching web page design ... not for the purpose of teaching HTML, but to teach students the concept of writing code." Mahmoud, et al [2] suggest that starting with HTML is a way of teaching "programming for fun" and is a strategy for motivating students.

Nonetheless, for first-time-coder, HTML can be difficult to learn. In a workshop with English students, Mauriello, Pagnucci, and Winner [3] observed "Students are generally not careful and experienced enough in their reading of the codes to find mistakes." HTML guides for non-coding students such as Taylor and Gitsaki [4] suggest simplifying the problem by starting with a small set of about 30 HTML tags to create a basic web page.

Therefore we are interested in finding an alternative to WYSIWYG HTML tools that expose the code, while still simplifying the process of learning to use HTML tags for the first time. In recent years, block programming languages such as Scratch [5] have introduced many students to coding through a visual representation of commands and control flow. Here we investigate whether a similar approach can be used with HTML code.

We aim to meet three goals simultaneously (borrowing terminology from Nielsen's usability heuristics[6]):

- Allow students to create HTML using *recognition rather than recall*, assembling blocks to make pages.
- Permit editing of real-world wepages with blocks or text: *the system must match the real world.*
- Provide students with a *minimalist design* with a clear, useful, realistic, yet minimal set of choices.

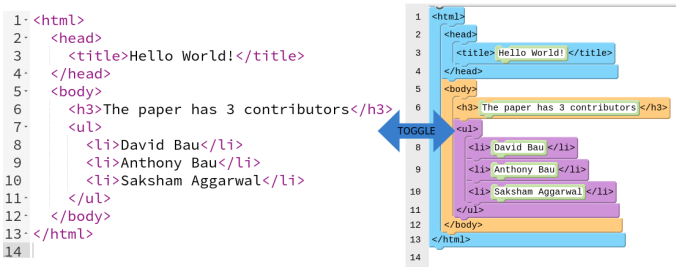


Fig. 1. Our HTML Block editor user experience

Users can switch between blocks and text freely.

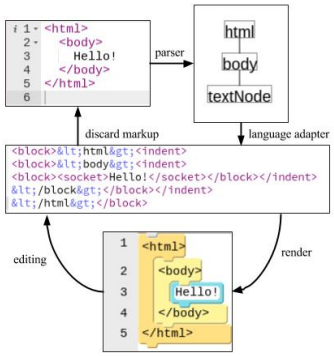


Fig. 2. Lifecycle of a Droplet Editing Session

II. Background

A. Droplet's Text-First Approach to Blocks

We built our HTML editor as an extension of Droplet. Droplet [7] is a dual-mode blocks and text editor framework that allows students to work with traditional text code syntax using either drag-and-drop blocks or by typing the text. Students can switch modes at any time.

Droplet's guiding philosophy is that the text, not the blocks, are the primary data. Thus, Droplet programs begin and end their life as text. When Droplet opens a file, the text is parsed into blocks, preserving the original text within block markup. Block editing operations perform splice operations on the text markup stream. At the end of the editing session, the markup is simply discarded and a raw text program is generated again. Figure 1 shows a typical droplet user experience for our HTML editor, and Figure 2 shows the lifecycle of a Droplet program.

978-1-4673-8367-7/15/$31.00 ©2015 IEEE

B. Adding A New Language to Droplet

Droplet is designed to be extended to any text language by creating a language adapter.

A Droplet language adapter must parse text to delinate blocks, and it can also provide rules that determine whether blocks are allowed to be dropped into specific locations. Typically the language parser relies on a standard language parser, inserting blocks based on positions of specific types of nodes in an AST. Callback functions are used to determine drag-and-drop permissibility.

New languages in Droplet also need a block palette. The Droplet palette is a reflection of useful choices, not a complete catalog of all possible blocks for a language. Similarly, dropdown choices for for sockets within blocks are curated suggestions. Users are always free to enter tags and attributes as free-form text, but a good palette design will allow all he common cases to be handled with drag-and-drop.

III. Process

A. Adapting A Parser

One of the goals of our HTML editor is to be able to visualize real-world webpages as blocks. This poses a difficulty because browsers are tolerant and many existing webpages are not strictly standards-compliant, with mismatched tags, and missing, implicit, or redundant elements.

Droplet's HTML mode adapts the parse5 HTML parser [8], which tolerates syntactically "incorrect" HTML code in the same way browsers do. We modified the parse5 parser for Droplet's purposes to add more detailed location data, so that precise source code locations for tags can be extracted.

The block parsing process for HTML proceeds as the following psuedocode:

1) Parse the text into an DOM tree using parse5.
2) Process the root of the DOM.
3) For each node, check if it is a text node, comment, empty element or an element with content.
4) Mark the range of a node based on its type
   - If a text node, make it editable.
   - If a comment, make the comment editable.
   - If an empty element, mark it as a block and make its attributes editable.
   - If an element with content, mark it as a block, make its attributes editable and add an indent to make space for its children.
   - If an implicit or error tag, do not mark it.
5) Recurse from step (3) for every child.

B. Enforcing Droppability Rules

One major advantage of a block language is that it can guide authoring so that students create standards-compliant code. Droplet's HTML mode therefore enforces droppability rules adapted from the WHATWG HTML specifications [9]. For example, a `<head>` element may only contain metadata elements such as `<title>` and `<meta>`, and a `<table>` element may only contain appropriate table elements such as `<tr>` and `<colgroup>` and `<caption>`.

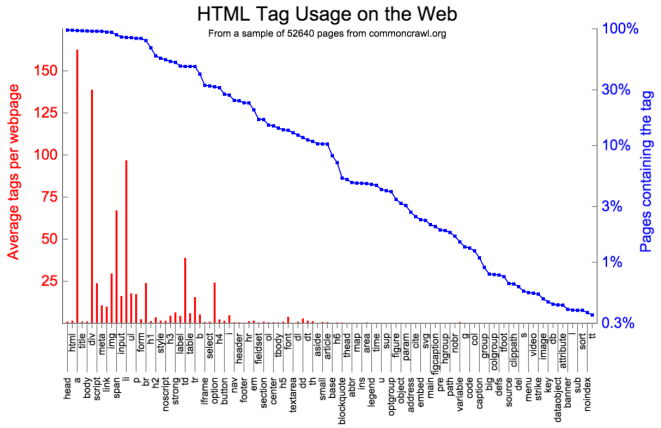


Fig. 3. HTML Tag Usage on the Web

We have implemented standard the containment rules for 92 tags. When a block is dragged into the document, only legal locations according to these rules are highlighted, and when a drop occurs, only those locations will receive a block.

C. Choosing A Palette

The palette in a block language is important to discovery and self-directed learning, because students can try new commands without having to read documentation. Having a palette that contains useful and rewarding tags in an HTML mode is therefore important.

The WHATWG HTML specifications define over 100 tags. However, most them are not used on a typical webpage. A number of developers online have informally posted HTML cheat sheets with the "most important tags," [13] [14] [15] but these are subjective and often conflict with each other. Because Droplet's philosophy is to be able to interact with real-world code on the Internet, we here determine and recommend a palette based on real-world tag frequencies.

We are not aware of any published statistics on the real-world frequency of HTML tag usage on the web, so we crawled a selection of 52460 webpages using Common Crawl [11] to create real-world frequency data. Our analysis of the top 108 tags is summarized in Figure 3. (Full data from our crawl is available on github [10].) Red bars in that graph represent a count of the number of times each tag was used per page on average in the crawled data set. The blue line, on a logarithmic scale, represents the percentage of documents in the data set that used the tag. In addition, we analyzed the most common tag-attribute pairs. These statistics are summarized in Figure 5. Red bars on that graph represent the percentage of webpages that include a given attribute on a specific tag.

We created the final palette (Figure 4) by choosing the top 40 tags from the above 2 analysis results, adding a few elements to align with courses such as [16] [17] [18], and removing some elements which have a similar behavior as another commonly-used alternative such as `<i>` versus `<em>`.

We also created dropdown lists for each element (Figure 6) to allow stduents to choose between the most common attributes. The student can choose an attribute and enter the

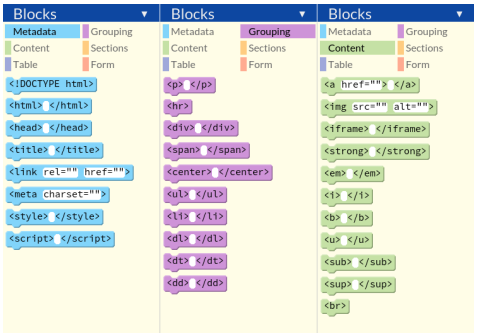


Fig. 4. Three panels from the palette

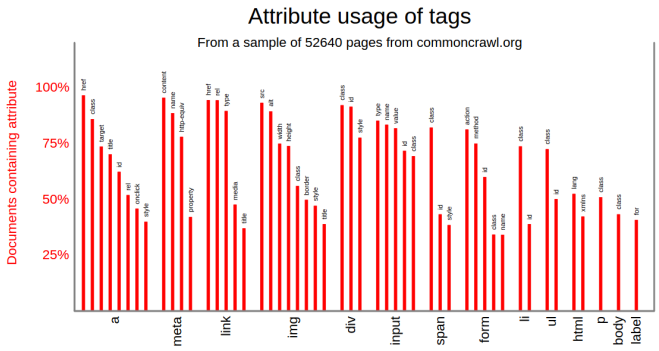


Fig. 5. Top attributes used with tags

attribute value as text, or just enter the entire attribute/value pair as text.

IV. Future Work

A. Buttons for adding and removing attributes

In HTML, every tag can accept a variable number of attributes, and the HTML mode would benefit from improved block editor support for variable argument lists. We are building improvements to make it easy to use "add/remove" buttons on a block for adding new attributes.

B. Polymorphic elements

Some elements are used in several distinct ways, and we plan to do further analysis on common tag usage to decide on a few tags to represent in multiple ways on the palette. For example, two ways of using `<a>` are with `href` or with `name`; two ways of using `<script>` are with `src` or with inline script.

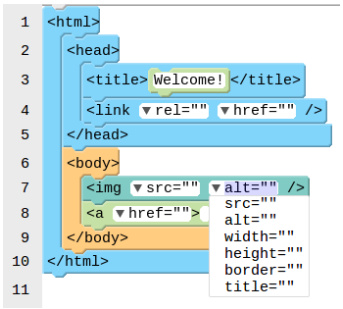


Fig. 6. Using commonly used attributes in a dropdown

C. Transparent content model

Some tags such as 'a', 'ins', 'del', 'map' are "transparent" elements, which means, according to the standard [24], their content model is derived from the content model of its parent element. This type of contextual content model is not currently implemented but provided as "flow content" [19]. We plan to add support for a transparent content model.

Acknowledgement

We are grateful to Google Summer of Code, which supported Saksham Aggarwal's open-source work on this project.

References

[1] Budny, D.; Lund, L.; Vipperman, J.; Patzer, J.L.I.I.I., "Four steps to teaching C programming," Frontiers in Education, 2002. FIE 2002. 32nd Annual , vol.2, no., pp.F1G-18,F1G-22 vol.2, 2002

[2] Qusay H. Mahmoud, Wlodek Dobosiewicz, and David Swayne. 2004. Redesigning introductory computer programming with HTML, JavaScript, and Java. SIGCSE Bull. 36, 1 (March 2004), 120-124. DOI=10.1145/1028174.971344 http://doi.acm.org/10.1145/1028174.971344

[3] Mauriello, N. Pagnucci, G. and Winner, T. Reading between the Code: The Teaching of HTML and the Displacement of Writing Instruction. Computers and Composition 16, 409-19 (1999)

[4] Taylor, R. and Gitaski, C. Teaching WELL and loving IT. New Perspectives on CALL for Second Language Classrooms, 131-147.

[5] Scratch. https://scratch.mit.edu/

[6] Nielsen, Jakob. Usability engineering. Elsevier, 1994.

[7] Bau, D. A. Droplet, A Blocks-Based Editor for Text Code. Journal of Computer Science in Colleges. 30, 6 (June 2015).

[8] Parse5. https://github.com/inikulin/parse5

[9] HTML living standard. https://html.spec.whatwg.org/

[10] https://github.com/sakagg/HTMLtagsFrequencyAnalysis

[11] Common Crawl. https://commoncrawl.org/

[12] HTML Palette in use on Pencil Code. pencilcode.net/edit/example.html

[13] Webmonkey. HTML Cheat Sheet. http://www.webmonkey.com/2010/02/html_cheatsheet/

[14] A Simple Guide to HTML. HTML Cheat Sheet. http://www.simplehtmlguide.com/cheatsheet.php

[15] Usabilla. An HTML Cheat Sheet That Never Fails. http://blog.usabilla.com/an-html-cheat-sheet-that-never-fails/

[16] Exploring computer Science - pages 105-110 http://www.exploringcs.org/wp-content/uploads/2014/02/ExploringComputerScience-v5.0.pdf

[17] W3Schools HTML starters guide http://www.w3schools.com/html/default.asp

[18] Htmldog beginner tutorial http://htmldog.com/guides/html/beginner/conclusion/

[19] WHATWG flow content list https://developers.whatwg.org/content-models.html#flow-content

[20] WHATWG metadata content list https://developers.whatwg.org/content-models.html#metadata-content

[21] WHATWG phrasing content list https://developers.whatwg.org/content-models.html#phrasing-content

[22] WHATWG interactive content list https://developers.whatwg.org/content-models.html#interactive-content

[23] WHATWG script supporting elements https://developers.whatwg.org/content-models.html#script-supporting-elements

[24] https://html.spec.whatwg.org/multipage/dom.html#transparent-content-models

Position Paper: From Interest to Usefulness with BlockPy, a Block-based, Educational Environment

Austin Cory Bart, Eli Tilevich, Clifford A. Shaffer, Dennis Kafura
Computer Science
Virginia Tech
Blacksburg, VA 24060
{acbart, tilevich, shaffer, kafura}@vt.edu

***Abstract*—As block-based environments are used for more mature audiences, the environments must mature themselves. Based on holistic theories of academic motivation, this means making the environment present itself as both interesting *and* useful, without sacrificing pedagogical power and scaffolding. We present Data Science as a potential context that satisfies all of these constraints, and describe our new block-based programming environment for education that supports data science from day one: BlockPy, available at `http://think.cs.vt.edu/blockpy/`. BlockPy features a number of powerful, authentic features meant to promote transfer for students to conventional environments as they progress. This includes mutual language translation and interactive feedback, but also powerful tools for getting real-world data and visualizing it. As we have developed the tool, we have identified a number of major research questions that should be answered in order to determine the validity of our hypothesis and the potential of our approach: in particular, how can this environment and context support educators and diverse learners as they progress into conventional environments.**

I. Problem

How do we bring introductory computing to mature, domain-identified undergraduates, who have concerns for both their own self-efficacy and for the value in learning computing? Many universities are now defining core credit hours in subjects such as "Computational Thinking", introductory computer science classes meant for solving interdisciplinary problems using some degree of programming. This means that universities now have students of every different discipline and background taking a programming course. Students with a clearly domain-identified interest (i.e. their major) may view non-major courses with doubt and suspicion – what does this have to offer them, and why should they engage?

Our position is that, in order to fully engage all undergraduate students, an introductory programming environment should be both *interesting* and *useful*, while still promoting *success*. Critically, this means that the environment should enable working with a context that students relate to, enjoy, and helps them solve useful problems. Further, they should feel that the material that they're learning will help them to transition to more authentic, serious problem-solving of real programming environments. These changes cannot come at the cost of the pedagogically valuable scaffolding, but instead should provide new opportunities for students' learning.

A. Existing Solutions

Although Block-based environments like Scratch and Snap! have already been successful with high-school students, they may not be suitable for more mature but diverse populations. In addition to side-stepping syntax headaches, Snap! and Scratch environments make it easier to get working with their motivating educational context, such as game and animation design, robots, or media computation, because they expose the range of actions afforded by the context (e.g., a "draw a circle" block for media computation) and support the experiences directly (e.g., a drawing canvas embedded within the environment). These contexts can be popular with certain students: young male children usually enjoy creating games, for example, so game and animation design is a compelling hook. However, many students at the undergraduate level may doubt the usefulness of the environment, as they consider it a toy rather than a useful tool, not only because of the puzzle-piece metaphor they utilize, but also for the evironments' contexts.

B. A New Solution: Data Science

We submit *Data Science Exploration* as an educational context that block-based environments should support as a priority, similar to how Snap! and Scratch support game design. Data Science for Introductory Computing is a growing movement, with many instructors recognizing the inherent value [1], [2] Data science provides an authentic, useful context for every kind of student, since exploring data-oriented problems is something that almost all fields are beginning to find relevant [3], [4]. Additionally, it is readily possible to find data sources that connect to the world around the student and their past experiences, establishing a sense of personalized interest. When students inevitably ask, "What am I going to be using this for?", it is possible to point to well-defined data problems in their field requiring computation. This does not mean that we are creating an end-user programming environment for data science, however, but an educational environment that allows students to learn computing principles through the context of data science (similar to how Scratch is meant to learn programming, rather than to learn game design).

C. Academic Motivation and Blocks

The prior research on block-based environments in undergraduate settings sheds some light on the limitations of environments for those populations. Mishra conducted a two-week intervention in an introductory course where students worked with Scratch before they began working with Java, reporting positive outcomes in both learning and engagement [5]. However, although the sample population was large (N=450), the students do not represent the typical body of a university:

978-1-4673-8367-7/15/$31.00 ©2015 IEEE

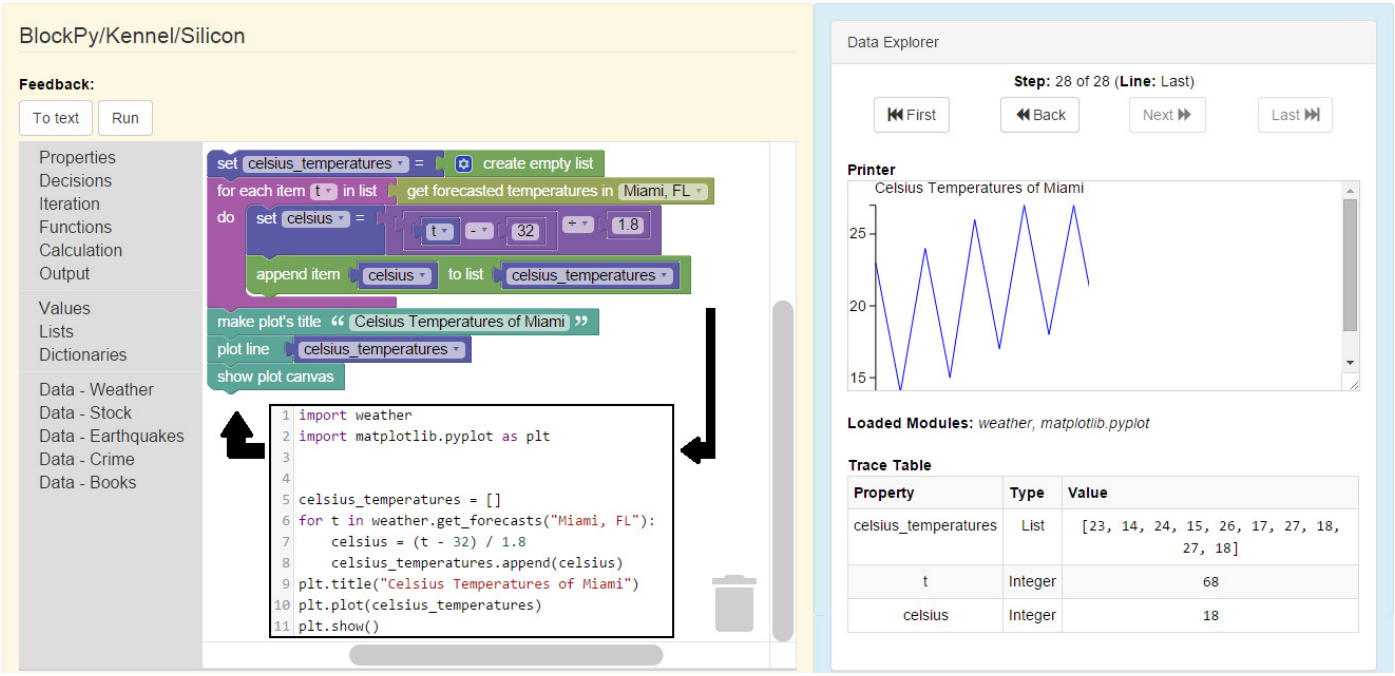


Fig. 1. A complete representation of BlockPy (`http://think.cs.vt.edu/blockpy/`)

they were engineering majors, 88% male, and “highest ranked” in “mathematics, physics, and chemistry”. The students did vary greatly on prior programming performance, but their other demographics suggest that they have uniform motivational concerns, compared to the general undergraduate population. In particular, many students cited the game design context afforded by Scratch as a major motivating factor: “[I am] thrilled to be able to code complex games” and “[coding] games helped increase my interest, [...], there was lot of room for experimentation.” These students valued the game design component because it was interesting to them, but not because they saw it as useful to their careers. It is unclear how more diverse students would react to this environment.

In our research project, we use the MUSIC Model of Academic Motivation [6] to explain the different ways students become engaged. Specifically, this model differentiates between five components of motivation: **eMpowerment**, the control that a student feels that they have over their learning experience; **Usefulness**, the student’s expectation that the material will be valuable to their short- and long-term goals; **Success**, the student’s self-efficacy; **Interest**, the situational and dispostional value the student’s feels; and **Caring**, the student’s perception of their professor’s and classmates’ attitudes toward them. We apply this theory to describe existing game-based programming environments as providing Interest to certain populations, but limited opportunities for Usefulness. We suggest that more students can be better served with a context that supports all five dimensions and, in particular, both Interest and Usefulness. To that end, we have created a new block-based environment with this in mind.

II. BlockPy

In this section, we concretely describe our work on a new web-based, dual text/block environment: **BlockPy**, a beginner-friendly programming environment that scaffolds the learner into a more mature environment while supporting a sense of Usefulness up-front. Internally, BlockPy uses a modified version of the open-source Blockly library to provide a block editor, a modified version of the open-source Skulpt library to execute Python code client-side, and an unmodified version of the open-source CodeMirror library to provide a text editor. Figure 1 demonstrates the interface: the problem presentation and feedback on the top-left, the dual program representations in the bottom left (via Mutual Language Translation), and the data science dashboard on the right (giving students powerful insight into their programs execution).

A. Data Science as a First-Class Feature

The code represented in the figure demonstrates the data science API exposed to the student, including blocks and functions to access real-world data sources and to create visualizations. These data sources include weather forecasts, earthquake reports, and stock feeds. Data returned from their interface is extremely simple – usually either primitive (numbers and text) or minimally structured (maps and lists), ensuring that students can begin working with Big Data blocks at the earliest possible points in the course. In addition to obtaining data, we support the popular MatPlotlib library to provide a set of visualization functions create simple line plots and histograms. By basing everything around the MatPlotLib API and relying on the Blocks interface for scaffolding, BlockPy seeks to maintain complete compatibility with conventional Python APIs so that all code written is authentic, as opposed to the use of simplified toy APIs in environments like CodeSkulptor.

B. Guided Practice

BlockPy is not just a code-authoring environment but also a system for guided practice. Instructors can create problems by

writing introductory text and then using an assessment API to define interactive feedback. Specifically, the instructor can define rules based on students' current code, output, and program state, and gives automatic feedback to the student. This just-in-time feedback is meant to guide students to success. Of course, the environment also supports free-form coding experiences, as you would find in traditional programming environments; as the students progress through their introductory experience, short-term feedback can decrease and then fade away.

C. Transfer to Textual Languages

Although some research is working towards creating end-user block-based environments, we view block-based languages as "Training Wheels", meant to be faded away. Work by Weintrop on the transition from Snap to Python analyzes this transition and offers a number of ways to mediate the transfer through programming tools. One of the largest findings is that being able to write inline code inside a Block-based language is extremely helpful to students' learning [7]. Another approach we support is Mutual Language Translation, devised by Matsuzawa [8], that creates an isomorphic view of students' code as both text and blocks. These features are meant to transfer students away from blocks towards text.

D. An Example Scenario

Consider a lesson for students on Iteration. The instructor could create a problem asking students to find the average temperature in their local city for this week, using the assessment API to construct rules demanding they use iteration blocks. Students would drag in a `Get Temperatures for [city]` block. If they attempt to run their code, they will be reminded that averaging requires iteration. As they continue, they switch between the block and text view as they feel comfortable. They also use the data explorer to step through their code and watch its state change. Finally, when they successfully print the answer, they are given positive feedback.

III. Research Questions

Our new programming environment offers a number of affordances to educators, but much of its promise is still unproven. Beyond just usability testing, we wish to explore questions relating to the nature of using data science in a block environment. One of the major values of a context is being relatable – it should be a metaphor for students, helping to build on their prior knowledge. Will the entire undergraduate population find data science to be sufficiently relatable? For instance, some students may possess weak math skills or have low self-efficacy with math. Will they find the necessary mathematics (e.g., finding the average of a list) too confusing?

Along similar lines, how do we quickly introduce students to a given dataset, and make them comfortable manipulating and understanding the data it contains? What interaction can the students have with the blocks in order to aid this experience? In the datasets currently supported by the environment, some are "easier" than the others: our students had no trouble working with weather data, for instance, but struggled when confronted by stock trading data. Are some datasets inherently more suitable for introductory experiences? And just how crucial are students' perceptions of interest and usefulness?

Of course, our environments' affordances also raise questions. In our experiences with using a block-based environment to scaffold learners into a conventional environment, the transfer can be rocky. Some students are eager to start using the text-based environment and do not need to pushed to move away from the blocks. However, some students may be wary about losing their training wheels and, if left to their own devices, may choose to delay trying out the text-based code. How do we gracefully transition students to coding text, based on the students' abilities, motivation, and the course's time table? How does Mutual Language Translation support and hinder this process? And how do we provide accurate block-based representations of a dynamic language like Python – consider the difficulties involved in inferring whether a variable block has the appropriate type to be connected to another block.

IV. Conclusions

In this paper, we have introduced our new environment, "BlockPy", that promotes Data Science through a block-based interface. We make a case that by relying on a more generally Useful context, rather than Interest, we can appeal to a wider range of mature learners. We describe a number of features we seek to support in our environment. Finally, we discussed the research that we are now exploring through this environment.

V. Acknowledgements

This material is based upon work supported by The National Science Foundation, grant TUES-1140318 and The National Science Foundation Graduate Research Fellowship, Grant No. DGE 0822220

References

[1] R. E. Anderson, M. D. Ernst, R. Ordóñez, P. Pham, and S. A. Wolfman, "Introductory programming meets the real world: using real problems and data in CS1," in *Proceedings of the 45th ACM technical symposium on Computer science education*. ACM, 2014, pp. 465–466.

[2] D. G. Sullivan, "A data-centric introduction to computer science for non-majors," in *Proceeding of the 44th ACM Technical Symposium on Computer Science Education*, ser. SIGCSE '13. New York, NY, USA: ACM, 2013, pp. 71–76.

[3] L. Layman, L. Williams, and K. Slaten, "Note to self: Make assignments meaningful," in *Proceedings of the 38th SIGCSE Technical Symposium on Computer Science Education*, ser. SIGCSE '07. New York, NY, USA: ACM, 2007, pp. 459–463.

[4] M. Goldweber, J. Barr, T. Clear, R. Davoli, S. Mann, E. Patitsas, and S. Portnoff, "A framework for enhancing the social good in computing education: A values approach," *ACM Inroads*, vol. 4, no. 1, 2013.

[5] S. Mishra, S. Balan, S. Iyer, and S. Murthy, "Effect of a 2-week scratch intervention in CS1 on learners with varying prior knowledge," in *Proceedings of the 2014 Conference on Innovation & Technology in Computer Science Education*, ser. ITiCSE '14. New York, NY, USA: ACM, 2014, pp. 45–50.

[6] B. D. Jones, "Motivating students to engage in learning: The MUSIC model of academic motivation," *International Journal of Teaching and Learning in Higher Education*, vol. 21, no. 2, pp. 272–285, 2009.

[7] D. Weintrop, "Minding the gap between blocks-based and text-based programming (abstract only)," in *Proceedings of the 46th ACM Technical Symposium on Computer Science Education*, ser. SIGCSE '15. New York, NY, USA: ACM, 2015, pp. 720–720.

[8] Y. Matsuzawa, T. Ohata, M. Sugiura, and S. Sakai, "Language migration in non-CS introductory programming through mutual language translation environment," in *Proceedings of the 46th ACM Technical Symposium on Computer Science Education*, ser. SIGCSE '15. New York, NY, USA: ACM, 2015, pp. 185–190.

Pushing Blocks all the way to C++

Jonathan Protzenko
Microsoft Research
One Microsoft Way
Redmond, Washington 98052
Email: protz@microsoft.com

***Abstract*—The BBC `micro:bit` project aims to teach programming to every 11 to 12-year-old in the UK, through the means of a programmable device half the size of a credit card. The device will be freely handed out to every student.**

Microsoft's TouchDevelop programming environment was picked to provide the programming experience for kids; we retrofitted the website for the `micro:bit`. TouchDevelop remains a complex beast: in order to make it easier for 7th graders to program, we added an alternative, visual code editor based on Google's Blockly [1].

This paper is an experience report about the various challenges we met when trying, at one end, to expose a visual Blocks-based programming model, while at the other end generating C++ for the device.

I. Overview of the project

The microcontroller will be handed out to students in the fall, and is about 4cm by 5cm wide. It is equipped with an ARM Cortex-M0 processor, 16k of RAM, 128k of flash memory, a compass, an accelerometer, a Bluetooth Low Energy (BLE) chip, and a series of pins to enable Arduino-style programming. The system features a 5x5 LED array and two buttons for input-output. The chip is programmable via the ARM mbed technology: upon plugging it into a computer, the chip masquerades as a USB key, meaning that it suffices to drag and drop a hex file onto the drive to perform flashing.

The BBC `micro:bit` is programmed using either ARM Cortex-M0 assembly or C++. The blessed toolchain is ARM's mbed platform, which consists of a web-based, C++ online IDE. Programs written in the online IDE are compiled in ARM's cloud; the resulting hex file appears as a browser download. We do not expect students to master the subtleties of C++; therefore, students get to write programs using TouchDevelop. Upon hitting "compile", the student's TouchDevelop program is translated to C++, which is then sent over to ARM's cloud. The resulting hex file similarly appears as a browser download.

The stated goal of the project is to teach the basics of programming to all students; as one BBC executive so accurately put it: "if we only manage to teach them that numbering starts at 0 in computing, we'll save the country a lot of bugs already". Joke aside, we expect advanced students to master basic control structures (conditionals, loops); variables; abstraction (via functions). While the `micro:bit` enables sophisticated projects that bundle sensors and displays over I2C, we expect most students to play with the onboard compass and accelerometer only.

TouchDevelop is a JavaScript-inspired, statically-typed, syntax-directed, web-based programming language. While suited to beginners, TouchDevelop still makes it possible to write faulty programs that require user intervention to fix. Therefore, in order to lower the entry barrier for 7th graders, we decided to include, in addition to the classic "TouchDevelop editor", an editor based on Google's Blockly. There are many reasons for this: we expect teachers to be more familiar with Blocks-based programming environment, due to prior experience with Scratch or Hour of Code; we also expect students to "click" better with Blocks programs.

The compilation scheme of "Blocks" programs is as follows: they are first (invisibly) translated to TouchDevelop, then the regular compiler from TouchDevelop to C++ kicks in and performs the rest of the job.

Going through TouchDevelop presents two major advantages. The first one is purely technical: we don't need two separate compilation paths. The second one is pedagogical: the student can choose to perform the conversion manually at any time. This presents a learning opportunity, showing the translation from Blocks to a more traditional programming language; it also allows the student to take their program to the next level and "unlock" more programming possibilities.

II. Programming model

One of the challenges we had when designing our Blocks language was to keep a programming model that we believe is simple enough for students to understand, while still making sure we can map Blocks programs to suitable C++. Here is an overview of the programming model we came up with.

Memory model Memory management is automatic; students do not have to allocate or free memory. Scalar values such as booleans and integers are passed by copy. Heap-allocated values such as 5x5 images and character strings are passed by reference.

Threading model Threading is cooperative, meaning that at certain program points, the current thread *yields* control back to the scheduler, hence giving other threads a chance to execute. Examples of threads include forever blocks (which fork a new computation) and event handlers.

Event model Students can either do active polling ("if button A is pressed, then do…"), or register event handlers ("when button A is pressed, do…"). Again, this is fairly convenient for the student, but makes our compilation

978-1-4673-8367-7/15/$31.00 ©2015 IEEE

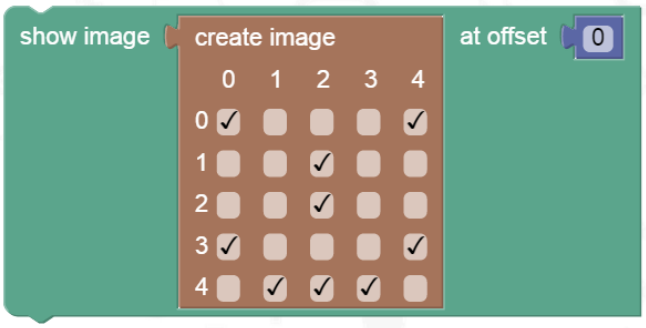


Fig. 1. User-friendly UI for designing new images

scheme more involved, as it involves closures under the hood.

Syntax model Thanks to a careful design of our blocks and their connections, issues such as starving are mitigated. Furthermore, we added some friendly blocks that allow editing 5x5 images and animations easily (Figure 1). It turns out in practice that starving is not an issue.

Type system For performance reasons which we expose below, we wish to statically type Blockly variables in order to generate efficient C++. Therefore, and quite against the Blockly spirit, we require variables to be typed. Types are inferred via a ML-style, Hindley-Milner type inference algorithm [2]. The type inference algorithm has been written outside of the Blockly codebase, meaning that it is not well integrated. Indeed, the user may get an unfriendly error message rather than visual clues. We hope to improve this.

A. Discussion of the programming model

We briefly contemplated skipping type inference, requiring instead that variables be annotated with their types at definition-time. The engineering effort would have consisted of an extra "definition" block, perhaps with some customization of Blockly to ensure the block is always provided[1]. We felt, however, that one of the great strengths of Blockly is its simplicity, and that requiring the user to understand and provide types for variables would have been a deterrent. In the current design, unless the student intentionally mixes, say, numbers and images, the compilation proceeds silently. We do perform a lot of work under the hood, but this remains invisible for the student.

Cooperative threading turned out to be essential: threads are never interrupted, meaning that we completely rule out data races. The main drawback to cooperative scheduling is that threads must yield often enough to avoid starving. In practice, we haven't had starving issues.

An alternative design would have been pre-emptive scheduling, with a hardware interrupt that runs at periodic intervals and schedules another thread for execution. A pre-emptive scheduler (as is found in most modern systems) would expose race-conditions, when two threads try to access the same variable. This would, in turn, take us towards the difficult problem of inserting locks automatically. Locks come with a memory and performance cost. We were happy to let students ignore all these issues.

Reference counting fails in the presence of cycles. It is conceivable that an advanced student indeeds succeeds in creating cycles, using advanced features of TouchDevelop (this is not possible with Blocks). However, ruling this out requires significant more run-time overhead than basic reference counting. We felt that the current tradeoff was satisfactory, and that this was a good learning opportunity for the few highly-advanced students.

[1]BlocklyDuino [3] adopts this approach, but does not seem to check that a variable is always properly defined.

III. The Blocks compiler

The compilation of Blocks, as we mentioned earlier, implies compiling first to the TouchDevelop language. There is nothing profound about choosing TouchDevelop as a target: it just so happens that we already needed a compiler from TouchDevelop to C++; furthermore, we wished to allow the user to perform conversion manually ("graduate"), so a compiler from Blocks to TouchDevelop definitely made sense.

Conjecture 1 (Correctness of TouchDevelop compilation)**.** *If a TouchDevelop program P is well-typed, then the result of compiling P to C++ compiles without errors.*

The semantics of TouchDevelop programs in the context of the `micro:bit` are fuzzy. We thus do not state any result about the preservation of semantics.

We of course want to make sure that a Blockly program never generates an (unscrutable) C++ compile error. Therefore, *compiling a Blockly program should generate a well-typed TouchDevelop program*, hence ensuring that compilation succeeds without errors.

Conjecture 2 (Correctness of Blockly compilation)**.** *If a Blockly program P is well-typed, then the result of compiling P is a well-typed TouchDevelop program.*

A. Alternative designs

We chose to generate a statically-typed C++ program that faithfully implements the original Blocks program, with TouchDevelop as an intermediary step. Some other designs are possible.

- We could forgo static typing altogether, and adopt a memory representation where all objects are tagged with their type, and have run-time checks for every operation to ensure that the operands are of the right type. This is dynamic typing. We ruled out this approach for efficiency and memory consumption reasons (the implementation of the BLE stack uses a significant chunk of memory, leaving very little of the original 16k available).
- We could compile Blocks programs to bytecode, and flash the device with a bytecode interpreter along with the bytecode that corresponds to the original program. This

is the approach used by MicroPython [4] and OCaPic [5]. We didn't have the engineering resources to design a bytecode format, and write an interpreter and run-time system for it (including a GC, most likely in Cortex-M0 assembly). Furthermore, our approach minimizes the memory and performance overhead in our constrained context.

B. Type-checking Blockly programs

Programs written in Blockly are dynamically-typed and all variables live in a common, global scope. TouchDevelop is a JavaScript-inspired, statically-typed programming language equipped with lexical scope. In order to successfully convert from the former to the latter, one needs to resolve the types of the program variables. We implemented *type inference for blocks*.

In Blockly, one can, and will want to define custom blocks. For instance, we have a "show image" block that displays an image on the 5x5 LED array. When defining a custom block, one specifies its shape; color; connections; labels, and so on. One can optionally provide types: for instance, the "+ (arithmetic plus)" block takes two numbers and returns a number. Similarly, the "show image" block takes a **`micro:bit`** image and returns nothing. These type annotations are leveraged by the user interface of Blockly: if the user tries to fill the "show image" block with a number, the smaller block "pops out" of the outer one, and the user cannot, "syntactically", write buggy code (Figure 2).

Fig. 2. Blockly's UI prevents connecting these two blocks: the "show image" block is defined as taking an image, not a number.

Not all blocks can be annotated, though. Since the types are static, and provided at *block-definition time*, one cannot provide a proper type annotation for the variable block. Therefore, the Blockly user interface performs no check for this block, meaning that invalid (per the TouchDevelop semantics) programs can be written, such as "show image item", where "item" has been assigned a number earlier (Figure 3).

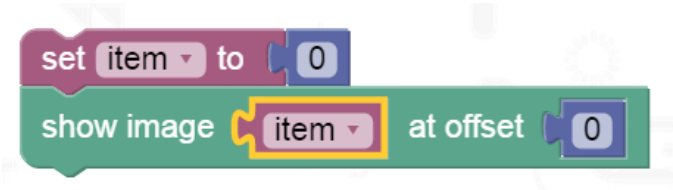


Fig. 3. This program cannot be translated to valid C++, since the `item` variable cannot be both an integer and an image.

In order to target a statically-typed language (TouchDevelop), which will in turn allow us to generate efficient C++ (with no dynamic types), we must infer the type of Blockly variables.

This is the classic type inference problem, wherein each block has an expected type for its arguments, and a return type. For instance, we may understand the "+ (plus)" block to have signature `int plus(int x, int y)`. Some blocks have a polymorphic type: for instance, equality comparison is meant to take two operands of the same type, so one may understand it to have type `bool equals<T>(T x, T y)`. Our type-checking algorithm therefore must perform unification, where upon comparing `x` and `y` for equality, we unify their types.

The algorithm kicks in at compile-time (or graduation-time). Since all variables are global, it suffices to loop over all blocks in no particular order. The type-checking algorithm is easy, since there is no subtyping or structural types such as pair or records (we did not enable lists). For each block of the program, we look up its associated typing rule and determine types accordingly. An English, readable version of the rules may be as follows:

- when assigning to `i`, the type of `i` must be the same as the return type of the right-hand side;
- when comparing two blocks with "=" (equality), the type of the two operands should match;
- when summing two blocks with "+" (plus), both operands should be numbers;
- etc.

It may be the case that the type of a variable still remains undetermined after type inference. This is not an error; this happens, for instance, when a variable has been defined, then never been assigned to. In this situation, we arbitrarily choose "number" for the type.

The algorithm uses textbook, imperative union-find structures to perform unification. Type inference and code generation are implemented as two separate phases. The implementation is made easier by the fact that there is no lexical scope, meaning that we don't need to maintain a lexical environment and deal with shadowing issues.

Because of the relative simplicity, the only possible issue is a type mismatch between the expected type and the actual type. We generate a classic message of the form "this variable is a number, but we wanted an image". We then highlight the faulty blocks in the user interface with extra CSS classes (Figure 4).

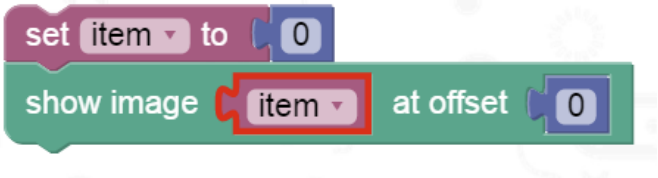


Fig. 4. Type inference flags the faulty program; in this picture, the `item` variable block is highlighted in red.

C. Run-time system

A C++ run-time system written by the University of Lancaster implements lightweight cooperative threading using fibers. In essence, multiple "threads" compete for execution. It is up to the current thread to yield control back to the runtime. This happens when: pausing, doing IO (reading the status of the pins, polling the buttons), scrolling images...

The only possible way to starve other threads is to write a `while (true)` loop and not use the run-time system within the loop body. In our context, this basically amounts to a useless loop; only bogus programs would have such an issue. Therefore, starving has not been an issue for us.

An interesting issue is the "forever" block. The block runs its body in a loop; however, implementing "forever" as `while (true)` would preclude any subsequent instructions from executing. It turns out that the intuitive semantics of the "forever" block is for the body to be executed in a loop, *on a new fiber*. Students naturally write several such "forever" blocks in the workspace and expect them to all run in "parallel". Our implementation makes sure that the fiber yields at regular intervals (Figure 5).

```
function forever (body : Action) do
  /* Repeat the code forever in the
     background. On each iteration, allows
     other fibers to run. */
  control -> in background do
    while true do
      body -> run
      basic -> pause(20)
    end while
  end
end function
```

Fig. 5. The implementation of the forever combinator

This raises the issue of forking: one could write a program that creates an unbounded number of new fibers, hence crashing the run-time system. In order to mitigate the fork-bomb issue, blocks that result in the creation of a new fiber ("forever", "on button pressed") cannot be nested within other blocks (they have no top connection). This means that the fork-bomb "forever (forever ...)" cannot be written in Blocks (it can, though, in TouchDevelop).

Fig. 6. The forever block is crafted in such a way that it cannot be nested

Having several fibers running at the same time is actually fairly useful: one may want to poll some output pin in the background (Makey makey-style project), while blinking an LED at a regular interval. Such a situation is easily expressed with two cooperative fibers.

D. Semantic matches and mismatches

Writing "when button A is pressed do..." essentially amounts to writing a closure, where the event handler may capture to variables in scope. The good thing is, in Blockly, all variables are global. Translated Blockly programs thus never refer to local variables, which completely eliminates the issue of proper scoping of captured variables, and of their subsequent compilation using a garbage-collected, heap-allocated block. This is an area where Blockly's model of global variables proved beneficial for us.

A difficulty was the compilation of for blocks into TouchDevelop for-loops. TouchDevelop only features a specific form of for-loop, where i is immutable, and one writes `for 0 ≤ i < ...` and only specifies the upper bound. This means that we had to replace the original Blockly for-loop (remove the "step" and "lower bound" parameters), in order to better match the TouchDevelop for-loop. Still, one cannot always translate a Blockly for-block into a TouchDevelop for-loop: Blockly still allows one to assign to the index (anywhere) or read it (outside the loop). We thus had to implement a series of checks (and reconstruct a notion of lexical scope in Blockly) to determine whether a Blockly for-block would result in a TouchDevelop for-loop or while-loop. This is partly due to our own constraints (we insisted on targeting TouchDevelop), party due to the very lenient model of Blockly.

IV. Looking forward

Arduino-like, hardware-based programming projects are gaining momentum. This is, after all, well deserved: unlike deploying a website to the cloud, one only needs to learn one language, the result is immediate, and the student can easily relate their program to the observed output.

One key feature in the **micro:bit** project is that it requires no special software: programming happens in the web browser, compilation happens in the cloud. Therefore, we believe that there is great potential in marrying blocks- and web-based programming environments with hardware targets.

In that context, blocks-based programming environments would benefit from a stricter, opt-in discipline that makes the rest of the compilation easier. One could, for instance, integrate the type-checking discipline within the Blockly codebase, and provide visual feedback based on the type of variables ("pop-out" on type mismatch, use colors...).

From the developer's perspective, facilities such as testing whether a variable is lexically scoped *within* a block would also be beneficial, as we could allocate some variables on the stack in the resulting, translated program.

REFERENCES

[1] N. Fraser *et al.*, "Blockly: A visual programming editor," 2013. [Online]. Available: https://developers.google.com/blockly/

[2] R. Hindley, "The principal type-scheme of an object in combinatory logic," *Transactions of the american mathematical society*, pp. 29–60, 1969.

[3] F. Lin, "BlocklyDuino, a web-based editor for Arduino," 2012. [Online]. Available: http://gasolin.github.io/BlocklyDuino/

[4] D. George, "Micro python," 2013. [Online]. Available: https://micropython.org/

[5] B. Vaugon, P. Wang, and E. Chailloux, "Programming microcontrollers in OCaml: The OCaPIC project," in *Practical Aspects of Declarative Languages*. Springer, 2015, pp. 132–148.

Block-based programming with Scratch community data: A position paper

Sayamindu Dasgupta
MIT Media Lab
Cambridge, MA 02142
Email: sayamindu@media.mit.edu

***Abstract*—In this position paper, I describe the rationale behind, and the early design of Scratch Data Blocks – a Scratch-based block-programming toolkit that allows Scratch users to program with public data from the Scratch online community.**

I. Introduction

A dominant trend in online communities is the collection and analysis of data from end-users. Though this data is then used for research, analytics, and marketing by the organization running the online-community, the users themselves rarely get to see this data. In many cases, users are not even aware that data is being collected about their activities in a given network – leading to what some scholars have termed as an "epistemological gap" [1] in privacy awareness. There is a similar trend in online learning communities and tools – though some recent developments show a positive change where designers of these communities and tools are actively engaging with the idea of presenting at least some data to community members and users through mechanisms such as dashboards, etc [2]–[4]. With the Scratch Data Blocks project, we aim to go one step further. Rather than having access to pre-programmed dashboards, with Scratch Data Blocks, young Scratch users will be able to create Scratch projects that can access, visualize, and analyze data about their own learning and participation (as well as their peers') in the Scratch community. Scratch Data Blocks will allow programmatic access to publicly available metadata about users and shared projects in the Scratch community. This would include not just social data (e.g. number of comments of a project, or the list of followers of a given user), but also, metadata about project code (e.g. number of blocks being used, or whether a certain type of block has been used or not).

With Scratch Data Blocks, we hope that Scratch users will find an engaging data-set (one that has been created by them and their peers), and utilize it to study and reflect upon not only their social participation, but also on their evolution as budding programmers.

II. Preliminary Design

In the initial design of Scratch Data Blocks, queries are expressed through "C-shaped" loop blocks. For example, projects can be retrieved through "foreach" queries of the form `for all shared projects by <<username>>` or `for all favorited projects by <<username>>` (Figure 1). Within these foreach loops or "C" blocks, context-sensitive reporter or predicate blocks can be used for retrieving metadata on the currently selected object (e.g. `title of project` or `country of user`). The available properties

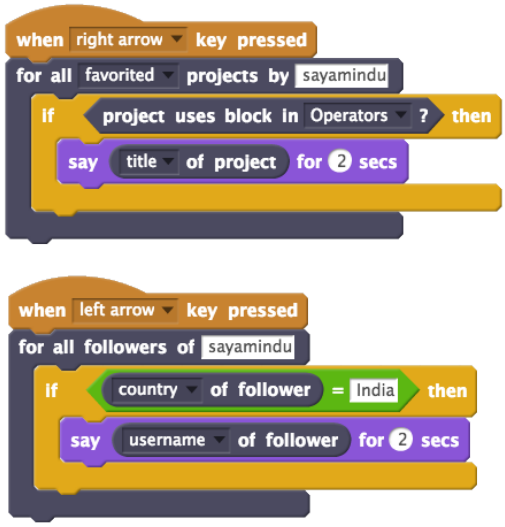


Figure 1. Scratch Code using Scratch Data Blocks

(e.g. country) for a given object-type (e.g. user) is listed in Table I. This approach, however, does represent a drawback of the current design, as Scratch has traditionally stayed away from context-sensitive blocks (with one notable exception of the `answer` reporter block to get user input). These context-sensitive blocks return a blank string, or false (in case of predicate blocks) when run outside of context (i.e. outside of the foreach loop).

Table I. Object-types and their properties

Object-type	Properties
User	username, bio, country
Project [a]	title, description, number of love-its, number of favorites, number of views

[a]The project object-type also has a predicate block for testing if a project uses a particular category of block, and another reporter block that returns the number of occurrences of a given block in a project's code.

In this design, though the loop or "C" blocks are tied to individual users (e.g. all projects shared by a *specific user*), it is possible to branch out in the Scratch community social graph through nested "C" blocks. For example, it is possible to enumerate projects shared by all the followers of a user by nesting a `for all shared projects by <<username>>` block inside of a `for all followers of <<username>>` block (Figure 2). This can be extended

978-1-4673-8367-7/15/$31.00 ©2015 IEEE

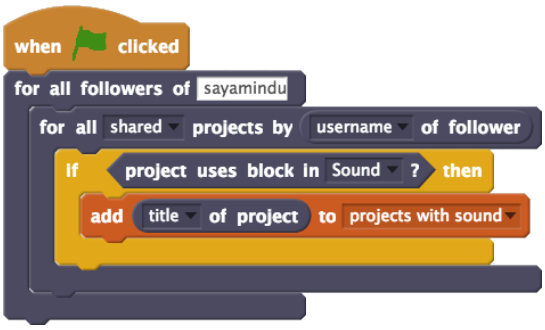


Figure 2. Nested loops to enumerate projects by all followers of a user

further to enumerate projects by friends-of-friends, though this process will quickly become slow on the client-side, and resource-consuming on the server-side, and there may be a need to impose a limit on the level of nesting that can be done for these kind of usage-scenarios.

It is worthwhile to note here that only public data is reported by the blocks, i.e. objects can be referred to or retrieved only if they are accessible publicly on the Scratch website. If a project gets unshared, or deleted, or if an user closes down his or her account, the relevant data will not be accessible via the blocks any more.

III. Future Directions

Over the next few months, I plan to start working with a selected group of Scratch community members to refine and iterate on the design described above. However, there are a few additions we hope to make to the toolkit before user-testing begins. A common phenomena within the Scratch online community are projects that celebrate community milestones (such as the sharing of the 10-millionth project, etc.). These projects are currently created manually, and there have been requests from the community in the past for access to community-wide statistics (e.g. number of projects currently shared, or the number of registered user). I plan to add reporter blocks for these global community statistics in the near-term future.

More global "queries" are also being considered, not for the initial launch, but for a subsequent release – for example, queries that enumerate projects currently in the homepage rows of the website, or users who have contributed to a given Scratch studio. A major open question here is that of performance – while it is possible in theory to have a block that allows the enumeration of all users in the community, or all projects in the community, it would entail addressing considerable architectural challenges on the server-side.

Longer term plans include exploring the possibility of write-access to the website – so that, for example, a Scratch project can generate an automatic "thank you" comment for anyone who clicks on "love-it" for the project. However, this sort of affordance, though extremely powerful, poses significant technical, as well as moderation challenges, and needs careful planning and thought before implementation.

IV. Acknowledgements

The author would like to thank Mitchel Resnick, Natalie Rusk, and other members of the Lifelong Kindergarten group at MIT, as well as Benjamin Mako Hill from the University of Washington for their suggestions and feedback during the conceptualization and prototype-design of this project. Financial support for this work came from the National Science Foundation (grant number 1417952).

References

[1] L. Floridi, "Big Data and Their Epistemological Challenge," *Philosophy & Technology*, vol. 25, no. 4, pp. 435–437, Dec. 2012. [Online]. Available: http://link.springer.com/article/10.1007/s13347-012-0093-4

[2] E. Duval, "Attention Please!: Learning Analytics for Visualization and Recommendation," in *Proceedings of the 1st International Conference on Learning Analytics and Knowledge*, ser. LAK '11. New York, NY, USA: ACM, 2011, pp. 9–17. [Online]. Available: http://doi.acm.org/10.1145/2090116.2090118

[3] J. L. Santos, S. Govaerts, K. Verbert, and E. Duval, "Goal-oriented Visualizations of Activity Tracking: A Case Study with Engineering Students," in *Proceedings of the 2Nd International Conference on Learning Analytics and Knowledge*, ser. LAK '12. New York, NY, USA: ACM, 2012, pp. 143–152. [Online]. Available: http://doi.acm.org/10.1145/2330601.2330639

[4] J. L. Santos, K. Verbert, S. Govaerts, and E. Duval, "Addressing Learner Issues with StepUp!: An Evaluation," in *Proceedings of the Third International Conference on Learning Analytics and Knowledge*, ser. LAK '13. New York, NY, USA: ACM, 2013, pp. 14–22. [Online]. Available: http://doi.acm.org/10.1145/2460296.2460301

Using Blocks to Get More Blocks: Exploring Linked Data through Integration of Queries and Result Sets in Block Programming

Paolo Bottoni
Department of Computer Science
Sapienza, University of Rome
Rome, Italy
Email: bottoni@di.uniroma1.it

Miguel Ceriani
Department of Computer Science
Sapienza, University of Rome
Rome, Italy
Email: ceriani@di.uniroma1.it

***Abstract*—While many Linked Data sources are available, there is a lack of effective non-expert user interfaces to build structured queries on them. The block programming paradigm promotes a gradual and modular approach. If—in an integrated user interface—both queries and results are represented as blocks, modularity can be effectively used to support an exploratory way of designing Linked Data queries.**

I. Introduction

The Linked Data [1]—the structured data available online—are increasing both in quantity and diversity [2]. A key advantage of the Linked Data model is its support for serendipitous exploration and reuse of existing data. In practice, though, exploring and querying Linked Data is not trivial and usually requires knowledge of a structured textual query language like SPARQL [3], the standard query language for Linked Data. Existing experimental tools for non-experts (see for example [4], [5]) while being effective for some cases, do not support the user very much in the incremental process of designing queries, because reusing intermediate queries and results for new queries is not easy.

In this paper, we discuss the use of the block programming paradigm to design queries on Linked Data sources. We specifically address the need for exploratory queries, through the integration of queries and their results in a uniform user interface (in the rest of the paper, UI). The discussion will be based on a concrete UI we recently proposed [6] to build queries on Linked Data.

II. Background

The Resource Description Framework (RDF) [7] is the data model proposed by W3C for Linked Data. In this model knowledge is represented via *RDF statements* about *resources*—where a resource can be anything in the "universe of discourse". An RDF statement is represented by an *RDF triple*, composed of *subject* (a resource), *predicate* (specified by a resource as well), and *object* (a resource or a literal, i.e. a value from a basic type). An *RDF graph* is a set of RDF triples. Resources are uniquely identified by an internationalized resource identifier (IRI) [8]. The resources used to specify predicates are called *properties*. *Prefixes* can be used in place of the initial part of an IRI, which represents specific namespaces for vocabularies or sets of resources. For example, the IRI namespace for standard RDF concepts[1] is often abbreviated as `rdf:`, as in `rdf:type`–the property used to associate a resource with its type(s).

III. A User Interface Based on SPARQL

The tool is based on Blockly, an open source library for block programming [9]. The UI (see Fig. 1) mostly sticks to the standard Blockly UI, consisting of a *workspace* for building block programs and a menu on the left—called *toolbox*—to pick the blocks from. Clicking on an item of the toolbox—a *category*—produces the visualization of a particular subset of blocks from which one can be dragged to the workspace. Following common practice, the blocks having the same role in the language also have the same color and belong to the same category in the toolbox. In order to maximize consistency with other block programming environments, the color and organization of blocks has been aligned to the standard Blockly blocks. The following subsections describe the different types of blocks provided, together with the language structure and the motivations behind the main choices.

A. Graph Queries as Blocks

The language mimics the structure of SPARQL, but will be described without explicit references to SPARQL syntax. In Fig. 2 a block representing a *select* query is shown. It corresponds to the natural language question "Give me the two highest Greek mountains." The *where* sub-block (the one to the right of the word "where") is a *graph pattern* —a block or set of blocks used to match parts of the given RDF graph—that "looks for" resources (the variable `mount` in the example) of a certain type (the class `dbo:Mountain`), connected through directed properties (`dbo:elevation`) to other resources or values (the variable `elevation`). Generally speaking, inside graph patterns different types of sub-blocks can be used:

[1] `http://www.w3.org/1999/02/22-rdf-syntax-ns#`

978-1-4673-8367-7/15/$31.00 ©2015 IEEE

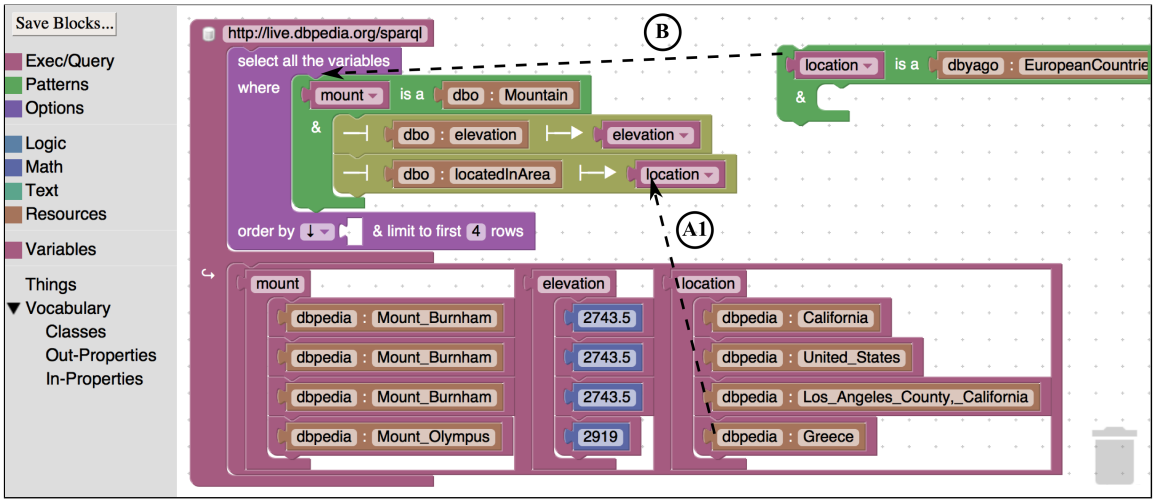


Fig. 1. User Interface after the execution of a query to get four mountains with their locations and heights

resources, *literals*, and *variables*. While resources and literals are constants that must match the same term in the given RDF graph, variables are the placeholders of the patterns that get bound to the terms of the given RDF graph for which the pattern matches. Graph patterns can be combined either by simply joining them together—in that case all must match at same time—, by using the *optional* block—the patterns under the optional may or may not be matched—, or by using the *union* block—at least one of branches of the block must match. The output of a select query is a list of tuples, each corresponding to a valid set of bindings of the used variables.

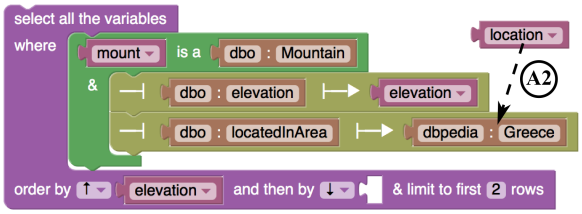


Fig. 2. Query to get the two highest Greek mountains

The blocks are visualized and organized (in the toolbox) according to their role: the query block is provided under the category *Exec/Query*, along the execution block that will be described in III-B; the different blocks used to build graph patterns are grouped under the category *Patterns*; the optional block and the union block are under the category *Options*; variable blocks—available in the category *Variables*—are managed and visualized in the standard way provided by Blockly; literal blocks have a basic type that is associated with one of the standard basic types used in Blockly, *boolean*, *numeric*, and *string* and are grouped under the categories *Logic*, *Math*, and *Text*; resource blocks can be found under the category *Resources*.

B. Execution and Results as Blocks

A specific *execution block* is designed for query execution: as soon as a query is attached to this block the SPARQL query is generated and run on the underlying dataset. Fig. 3 shows an execution block connected to a query and "waiting for the results". When the results are ready (see for example Fig. 1), they appear in tabular format attached to the lower connection of the execution block. The single data items are represented as resource blocks and literal blocks that can be dragged from the result set to create other queries (or even modify the one that generated them).

The existence of two distinct blocks for query and execution allows the user to distinguish the design of the query from its execution if he or she wants so: by building the query separately, the user gets the results only when the query is connected to an execution block. Conversely, by modifying a query already connected to an execution block, the query is executed immediately and the results shown as soon as they are available[2].

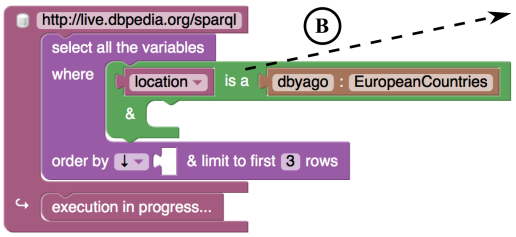


Fig. 3. Execution of a query to get three European countries

IV. Affordances

To argument in favor of the proposed paradigm, we will show some of the affordances—related to the process of designing a query—of such UI.

[2]Query execution is in any case non-blocking, i.e. the UI is reactive and operational even if one or more queries are being executed.

A. Generalization/Specialization

The design of a query may proceed from a generic version, with a minimal number of constraints, designed to start getting some data. Then, by adding constraints the query will gradually become more selective. This process, that we may call *query specialization*, is supported by reducing the free variables (through replacement with constant values or with already used variables), by adding *filter* blocks (see the "provided that" blocks in Fig. 4), by adding more graph patterns to be fulfilled, and by moving graph patterns out from an *optional* or *union* block The usage of blocks for the query output allow the user to get easily the specific blocks corresponding to resources or literals needed to replace variables or to create filter expressions. An example of *query specialization* is the transformation from the query in Fig. 1—that asks for some mountains, their locations and their heights—to the query in Fig. 2—where only Greek mountains are selected–by dragging the resource `dbpedia:Greece` from the results in the place of the variable `location` (see arrow A1 in Fig. 1).

The design of a query may also start from a specific query on known data and then proceed by lifting some constraints to include a greater set of results. We may call this process *query generalization* and it is supported by replacing constant values with variables, by removing filter blocks, by removing graph patterns (or part of), and by moving graph patterns under an *optional* or *union* block. These actions corresponds directly and naturally in our environment to the removal of blocks or the creation of new ones for the variables. An example of generalization can be the reversal of the specialization example, i.e. from the query in Figure 2 to the one in Fig. 1, by dragging on the workspace a new variable block to replace the resource `dbpedia:Greece` (see arrow A2 in Figure 2).

B. Composition/Decomposition

A complex query may be created by composing different queries together, so that, for example, the output of separate components of the query can be checked before composing them. As the proposed UI permits multiple queries to be built and executed in the same workspace, composition is directly executed by joining blocks from different queries[3]. For example, to build a query for European mountains the user may first design a query to get mountains and their locations (Fig. 1) and another query to get the European countries (Fig. 3). The complete query can be composed by dragging the graph pattern of the latter query to add it inside the former one (see arrows B in both figures).

The reverse operation is to decompose a query in smaller ones. From the point of view of the UI the actions are similar to the ones needed for composition: dragging blocks and creating some new ones. Reversing the example of composition gives an example of decomposition.

C. Stepwise Querying

Sometimes a query design comes after some exploratory steps which identify some relevant resources, classes or properties. This approach is supported through permitting values dragged out of the result set of a query to be used by other queries. The old query may then be removed or simply kept aside for further use. For example, the query in Fig. 4 may be used to get some candidate classes to represent the set of mountains, by searching among available classes using their label. As soon as the resource `dbo:Mountain` is identified as the relevant class, the corresponding resource block may be dragged away (see arrow C in Fig. 4) to be used for follow up queries, like for example the one in Fig. 1 (see the "is a" part of the graph pattern).

[3]Blocks may also be duplicated to preserve the original queries.

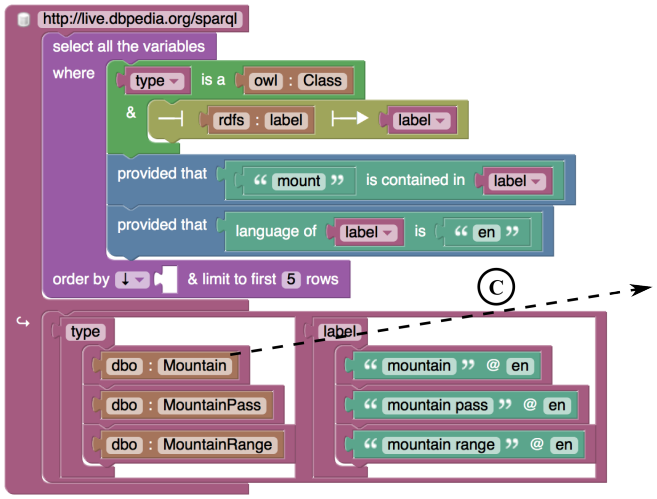


Fig. 4. Execution of a query to get classes with a label containing "mount"

V. Conclusions and Future Work

We propose a new paradigm for querying Linked Data, based on a novel take on block programming: using blocks not only to program but also to show results, which can be incorporated in incremental design of queries. The proposed incremental approach and UI could be possibly extended to a wider set of query languages and data models. We plan to start an extensive user evaluation of usability, both for Linked Data and in a more general setting.

References

[1] T. Berners-Lee, "Linked data," Design Issues, 2006.

[2] C. Bizer, T. Heath, and T. Berners-Lee, "Linked data-the story so far," *Int. J. on Semantic Web and Information Systems*, vol. 5, no. 3, pp. 1–22, 2009.

[3] S. Harris *et al.*, "SPARQL 1.1 Query Language," W3C REC 21 March 2013.

[4] S. Ferré, "Sparklis: a SPARQL Endpoint Explorer for Expressive Question Answering," in *ISWC 2014 Posters & Demonstrations Track*.

[5] A. Russell, P. R. Smart, D. Braines, and N. R. Shadbolt, "Nitelight: A graphical tool for semantic query construction," 2008.

[6] P. Bottoni and M. Ceriani, "Linked Data Queries as Jigsaw Puzzles: a Visual Interface for SPARQL Based on Blockly Library," in *Proc. of CHItaly 2015*. ACM, 2015, p. [To Appear].

[7] R. Cyganiak, D. Wood, and M. Lanthaler, "RDF 1.1 Concepts and Abstract Syntax," W3C REC 25 February 2014.

[8] M. Duerst and M. Suignard, "Internationalized Resource Identifiers (IRIs)," RFC 3987 (Proposed Standard), Internet Engineering Task Force, Jan. 2005. [Online]. Available: http://www.ietf.org/rfc/rfc3987.txt

[9] N. Fraser *et al.*, "Blockly: A visual programming editor," 2013.

Incorporating real world non-coding features into block IDEs

Mikala Streeter

Georgia Institute of Technology
Atlanta, GA

***Abstract*—We often see block-based coding environments as toy environments that let novice programmers have fun as they learn the basics of programming. While these environments do have an engaging low floor, they are missing out on other aspects of introductory programming that could further engage students and better replicate the real work of software developers. In this paper, I describe real world non-coding features (e.g. collaboration, code reviews, version control, aesthetic appeal, galleries/community) that should be incorporated into block IDEs. There are likely other important non-coding features that are not included.**

Keywords—blocks; programming; collaboration; code reviews; version control; aesthetic appeal; galleries; novice

I. Introduction

We often see block-based coding environments as toy environments that let novice programmers have fun as they learn the basics of programming. While these environments do have an engaging low floor, they are missing out on other aspects of introductory programming that could further engage students and better replicate the real work of software developers. In this paper, I describe real world non-coding features (e.g. collaboration, code reviews, version control, aesthetic appeal, galleries/community) that should be incorporated into block IDEs. There are likely other important non-coding features that are not included.

II. Collaboration

In introductory courses, we often enlist pair programming as a low risk way for students to create projects. This approach gives each student a buddy with whom they can discuss ideas, write code and fix bugs. However, an issue that often arises is when students want to continue working on their projects at home or when their partner is sick. The project was likely created under the account of one student assuming that she'd be there each day for class, but when she's not, if the teacher hasn't written down the student's login information, her partner will either lose a day of work or will have to rush to re-create all of their work. If block IDEs allowed students to add collaborators to a project like professional software developers can do through tools like Github, then this issue would never occur. Both students would have easy access to their shared code because they are both collaborators on the project. Students would also be able to work in larger groups of 3 to 4 people on multiple computers, creating more complex code and experiencing what it's like to work on a real team of software developers building something amazing together.

III. Code Reviews

Another key aspect of collaboration and learning at the introductory level is getting feedback on the quality of your code. In industry, professionals do this through code reviews, which can occur both synchronously and asynchronously. By focusing primarily on the changes made from the previous version, teams can give one another in-line feedback. While computer programs give students immediate feedback on how their code works, a real person needs to intervene to give feedback on the elegance and efficiency of the code. Adding the ability for reviewers and collaborators to do code reviews of the latest changes by tying comments to the new blocks or scripts would speed up grading for educators, decrease wait time for students wanting feedback, and enable asynchronous mentoring of students by industry professionals.

IV. Version Control

In professional work, programmers are able to take deep dives into rabbit holes with their code because they know that their versioning software will easily allow them to backtrack. For them, this flexibility is useful for intentional experimentation with coding approaches. For introductory students, versioning would also be useful albeit for unintentional experimentation, where they thought they had enough time or understanding to try a complex approach, only to later realize that it wasn't the best course of action, but now have gone too far down this path to Undo their work. If block IDEs were

978-1-4673-8367-7/15/$31.00 ©2015 IEEE

built with version control, preferably through auto-save, students could much more safely experiment knowing that they can go back to an earlier working version with ease. Version control would also enhance code reviews in the event that students are given feedback that the latest version of code is quite poorly written and should be reverted. Version control would additionally support collaboration to simplify merging code from multiple students.

V. Aesthetic Appeal

While some introductory students try computer science because they're interested in the technical challenge or the puzzles, some were intrigued by the potential to make the beautiful mobile apps and video games they use everyday. They're excited about incorporating their own creativity and design eye to their coding work, only to find that this isn't a priority in many block IDEs. These students may feel that their love of design isn't valued in the technical world and be turned off from the field altogether or at a minimum, forgo the possibility that their interests in art and technology can intersect, possibly putting them on an outbound trajectory in the long run. If block IDEs were designed, perhaps like Twitter Bootstrap that makes even the simplest websites beautiful, to ensure aesthetic appeal as students learn to code, beginner students would see how even as novices they can create work that is both complex and beautiful.

VI. Galleries/Community

Scratch and OpenProcessing have shown us the power of worked examples in programming communities. There is a rare day that goes by in my introductory classes that I don't encourage a student to search for an example in either community. These galleries inspire new students about what's possible as they develop their skills and also validate their work as they receive comments and tweaks/remixes from others in the community. And yet, new block IDEs are being built with galleries and community as secondary or even tertiary features. If our focus is on broadening participation, we need to make every student feel as though as they are a part of something much larger than themselves, and that there are lots of people also learning who are willing to help and encourage them along the way. Communities and galleries go a long way for sending this positive message.

VII. Conclusion

As we teach students to code, we need to keep in mind the importance of enculturating them into a computer science community of practice. This community is about much more than being able to write code. It's about being able to write quality code often with other people over many iterations to build something that works well and looks beautiful. In order for students to determine whether they want to stick with computer science, they have to see the full picture and incorporating these non-coding features into the blocks-based programming environments will go a long way in service of this goal.

Online Community Members as Mentors for Novice Programmers Position Statement

Michelle Ichinco and Caitlin Kelleher
Department of Computer Science and Engineering
Washington University in St. Louis
St. Louis, USA
{michelle.ichinco, ckelleher}@wustl.edu

***Abstract*— While online communities exist surrounding blocks programming environments, they do not support the type of feedback a novice programmer might receive in a classroom setting. We propose a feedback system in which experienced programmers author feedback and programmatic rules to distribute feedback to novice programmers. Finally, we outline three main considerations for designing systems for online community feedback for novice programmers: community members, the format of the feedback, and how the feedback is distributed at a large scale.**

Keywords—novice programming; online communities; block programming

I. INTRODUCTION

Current online communities for block programming languages allow users to share their content, provide basic feedback such as "likes" and comments, and reuse programs. Many programmers involved in these communities learn to program outside of a classroom, so they only receive these basic forms of feedback on their programs. However, feedback is critical to improving a new skill, especially programming. Furthermore, in some online communities, content rarely even receives likes or comments, yet feedback is directly linked to participation. Feedback, especially when it is specific and personalized, can be critical in boosting motivation and learning for novices.

We believe that online communities for block programming languages can be leveraged to provide more specific and useful feedback to programmers. One such system involves experienced programmers who view content contributed by friends or family members. These experienced programmers may have valuable feedback that they can provide to a novice programmer they know, which can also extend to other novice programmers. This work describes a system for leveraging experienced programmers as mentors for novices. We then present open questions about the possible types of community members, types of feedback, and ways to scale online community feedback.

II. EXISTING ONLINE COMMUNITIES

A variety of current online communities exist to support users of blocks based programming languages. However, none support detailed feedback aimed at motivating novices or helping them improve their skills. For example, Scratch has a community of users who share projects, collaborate, comment, and reuse [1]. Looking Glass also has an online community where users can share, reuse, and comment on projects [2]. Similarly, TouchDevelop [3] has a showcase where users can post their projects and comment on other users' projects. Kodu [4] and App Inventor [5] have forums where users can ask questions or discuss a variety of topics. However, these communities do not provide support for members to give in-depth feedback. In order to harness the power of online communities for blocks programming environments, we need community members to be able to act as mentors for novice programmers and also ensure that feedback can be extended to a large population of novice programmers.

III. EXPERIENCED PROGRAMMER MENTORS

We propose one way to utilize an online community: to have experienced programmers act as "mentors", contributing feedback for novices. Some novice programmers will likely have a friend or family member who has programming experience and motivation to help the novice improve. If we can leverage the feedback of a small number of mentors from an online community by having them author rule scripts that distribute the feedback to other novices, we can provide personalized valuable feedback to novice programmers at a large scale. In this system, experienced programmers have three main roles: A) feedback creation, B) rule authoring, and C) feedback and rule vetting.

A. Feedback creation

We propose that experienced programmers can identify issues in programs and improve novice code with more complex concepts. An exploratory study showed that experienced programmers often made valuable technical suggestions, as well as suggestions based on program content [6]. We hypothesized that a system could present example code for a novice programmer to use in improving their program, but many challenges still exist in novice programmer use of examples [7]. It is also possible that better forms of feedback exist that could be more motivating or effective than example code. Regardless, the feedback should not show the novice programmer exactly what they need to do, but should demonstrate skills that the novice programmers has not yet mastered.

978-1-4673-8367-7/15/$31.00 ©2015 IEEE

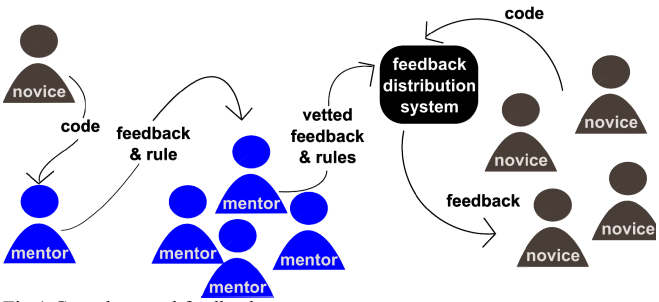


Fig 1 Crowdsourced feedback system

Once the mentor improves a part of a program, the system can then present feedback to novices to improve their code the same way, or to learn that skill. The system can determine which novices should receive which feedback using rule scripts authored by the mentors.

B. Rule Authoring

Many experienced programmers can author a short script, or a rule, that can check whether a novice program contains a certain issue. We ran an exploratory study that showed that experienced programmers could often write a pseudocode rule to check a program, like the example shown in Fig. 2 [6]. However, early work on the design of a system for rule authoring showed that only highly skilled programmers can actually author working rules. One reason for this might be that experienced programmers do not often work with block programming languages, so even with API support, authoring a rule to check the contents of a block program was challenging. However, another issue may be varying "experience" levels in the community of mentors. Some programmers noticed right away by reading the instructions for rule authoring that it was beyond their skills. Others attempted to author rules, but they were not all successful. Due to the differing skill levels and possible quality issues related to crowd work, this system needs a way to ensure that feedback and rules actually provide valuable information to novice programmers.

C. Feedback and Rule Vetting

We propose that having experienced programmers create feedback and then vet or improve each others' feedback can produce high quality feedback. While experienced programmers likely are not trained in creating educational content, we hypothesize that a community of mentors can create valuable educational material through an iterative process of improving and filtering the feedback. Furthermore, some experienced programmers who do not necessarily have the skills to author rules or even create feedback may have enough knowledge to determine the quality of the feedback. Thus, this system can leverage a variety of different mentor skill levels to produce valuable feedback for novices at a large scale.

```
for each doTogether in program:
        if doTogether.size()<2:
                return True
```

Fig 2 An example of a pseudocode rule. In this rule, a program receives feedback if it has a parallel execution block with only one statement inside, since that might indicate a misunderstanding of the parallel execution concept.

IV. Design Features of Online Community Feedback

Based on the findings and issues with having experienced programmers provide feedback to novices, three important questions are: which community members can and will provide valuable feedback, what format is best for providing valuable and motivating feedback, and what is the most accurate and efficient way to provide feedback at a large scale?

A. Community Mentors

There are a variety of different types of online community members who could fulfill the role of a mentor, such as experienced programmers and novice programmers who have started to gain skills.

1) Experienced programmers

Experienced programmers will likely program as a part of their career and have a friend or family member using a blocks based programming language. Their motivation would stem from their relationship with a specific novice programmer who they would like to help. The benefit of involving these experienced programmers is they may be motivated to continue providing feedback as long as the novice programmer they know continues programming. We have also found that some experienced programmers generally want to "give back", by sharing their programming skills. However, these "experienced" programmers may have a variety of different skill levels and often do not have training in education. Thus, having experienced programmers provide feedback requires that the feedback be checked for quality and generalizability.

2) Novice programmers with skills

"Novice programmers" who have learned certain skills may also be able to provide feedback to newer programmers. These experienced novice programmers have the potential to generate highly motivating content, as they have more experience with the blocks based programming language and features. They also might generate feedback that is closer to the level of the novice programmer, since they just learned those skills. Like the experienced programmer feedback, this feedback would likely also need vetting.

B. Feedback format

The feedback novice programmers receive should motivate them to continue programming, in addition to helping them improve their programming skills. In the context of blocks based programming, users often choose to program a specific project and may not want to be interrupted with suggestions. Instead, the feedback could be an alternative programming activity that grabs their attention and gives them new ideas and direction. Ideally, the feedback novice programmers receive can both motivate them and educate them, by presenting new skills that novices can use to create exciting new content. Two possible ways to do this are to provide extra motivation to use feedback, such as awards for completing certain tasks [8], or by presenting feedback in a highly motivating way, such as a game or puzzle. Likely, other motivating and effective feedback forms also exist.

C. Feedback distribution

Since it is unlikely that online community members can generate feedback for each program created, we need a way to distribute feedback to a large number of programmers who could also benefit from it, such as by programmatic rules, machine learning, or a curriculum. However, mentor rules may not always correctly generalize an issue, while machine learning can also misclassify a programmer's skill level and knowledge gaps. With a curriculum, the system would also need to check where a programmer is within the curriculum, and it may not be personalized to the programmer's needs. Regardless of the method, we also need to be able to capture whether the distribution of feedback accurately captures the needs of the programmers.

V. Conclusion

In this position paper, we argue that online community members can help to provide detailed feedback to novice programmers at a large scale. We have shown that experienced programmers have the capabilities to fulfill this role, by creating feedback and authoring rules to distribute the feedback. However, another possible direction is to have novice programmers create feedback for others. We believe there is still more work to be done to discover the best way to present feedback to novices in blocks programming environments, as well as in how to best distribute feedback at a large scale.

VI. References

[1] "Scratch | Home | imagine, program, share." [Online]. Available: http://scratch.mit.edu/. [Accessed: 19-Sep-2012].

[2] "Looking Glass Community." [Online]. Available: https://lookingglass.wustl.edu/. [Accessed: 24-Feb-2013].

[3] "TouchDevelop - create apps everywhere, on all your devices!," *TouchDevelop*. [Online]. Available: https://www.touchdevelop.com/. [Accessed: 21-Jul-2015].

[4] "Kodu | Home." [Online]. Available: http://www.kodugamelab.com/. [Accessed: 21-Jul-2015].

[5] "MIT App Inventor | Explore MIT App Inventor." [Online]. Available: http://appinventor.mit.edu/explore/front.html. [Accessed: 21-Jul-2015].

[6] M. Ichinco, A. Zemach, and C. Kelleher, "Towards generalizing expert programmers' suggestions for novice programmers," in *Visual Languages and Human-Centric Computing (VL/HCC), 2013 IEEE Symposium on*, 2013, pp. 143–150.

[7] M. Ichinco and C. Kelleher, "Exploring Novice Programmer Example Use," *IEEE Symp. Vis. Lang. Hum.-Centric Comput.*, To Appear 2015.

[8] A. Zemach, "The Effects of Gamifying Optional Lessons on Motivation," Master's Thesis, Washington University in St. Louis, 2014.

Programming Environments for Blocks Need First-Class Software Refactoring Support

A Position Paper

Peeratham Techapalokul and Eli Tilevich
Software Innovations Lab
Virginia Tech
Email: {tpeera4, tilevich}@cs.vt.edu

***Abstract*—Block-based programming languages and their development environments have become a widely used educational platform for novices to learn how to program. In addition, these languages and environments have been increasingly embraced by domain experts to develop end-user software. Though popular for having a "low floor" (easy to get started), programs written in block-based languages often become unwieldy as projects grow progressively more complex. Software refactoring—improving the design quality of a codebase while preserving its external functionality—has been shown highly effective as a means of improving the quality of software written in text-based languages. Unfortunately, programming environments for blocks lack systematic software refactoring support. In this position paper, we argue that first-class software refactoring support must become an essential feature in programming environments for blocks ; we present our research vision and concrete research directions, including program analysis to detect "code smells," automated transformations for block-based programs to support common refactoring techniques, and integration of refactoring into introductory computing curricula.**

***Index Terms*—refactoring, metrics, code smells, block-based programming languages, end-user software engineering, computer science curriculum, introductory programming.**

I. Introduction

As a software project grows, so does the complexity of its source code. Without targeted and timely programming efforts to counter this complexity, it can quickly overwhelm the average programmer. *Software refactoring*—changing a program's structure to improve the program's design while preserving its external functionality—has become an indispensable part of the modern software development process. Modern text-based programming environments support refactoring via *refactoring browsers*, sophisticated program analysis and automated transformation engines; they enable the programmer to *refactor with confidence*, with assurance of refactoring transformations not changing the program in unexpected ways. This first-class refactoring support, which has become an inextricable part of modern integrated development environments (IDEs), serves as one of the pillars of the iterative programming process, in which development is intermingled with a continuous stream of improvement and enhancement tasks[1], [2].

Block-based programming languages, including Scratch and Blockly, have become highly popular by providing both a "low floor" for novices to get started and a "high ceiling" for more experienced programmers to create increasingly complex projects over time. Despite their proven educational utility, programming environments for blocks are still in infancy compared with text-based languages when it comes to supporting the software development process with state-of-the-art software development tools, based on program analysis and automated transformation.

Nevertheless, software engineered in block-based languages suffers from the same problems of design rot and code quality deterioration, as programs become increasingly complex. The problem is real. Scratch projects authored by novice programmers have been found to contain a large quantities of message passing and scripts scattering around[3]. Other prevalent design problems included duplicate code, uncommunicative name, excessive use of concurrent scripts, etc. These manifestations of unnecessary design complexity and code quality degradation make software written in block-based programs hard to comprehend, maintain, and evolve.

We argue that the best practices of systematic refactoring support for text-based languages can and should be applied to block-based languages and environments, so that block-based software can reap similar software engineering benefits. To that end, programming environments for blocks should be enhanced to provide support for program analysis and transformation. State-of-the-art support is required to enable block-based language programmers to become aware of design problems as they arise by ascertaining standard software metrics used to measure code quality, to discover "code smells" to uncover the symptom of poor design, and finally, to transform programs automatically with a refactoring tool to safely and efficiently improve their design.

The expected benefits will be immediately tangible: block-based software that is easier to understand, modify, and evolve. By systematically growing the refactoring support for block-based languages, one can effectively raise their "high ceiling" property; first-class refactoring support can equip the average programmer with powerful automated tools for managing

978-1-4673-8367-7/15/$31.00 ©2015 IEEE

program complexity. Systematic refactoring support can also advance the role of block-based programming for end-user software development. In the following discussion, we present the research ideas and directions that will form the basis of this effort as well as discuss some possible paths for integrating refactoring into introductory computing curricula.

II. Capturing Design Quality

Without a way to measure the design quality of a program, one cannot be certain about when and where the program needs improvement. Thus, essential to modern software development process are the tools to evaluate the design quality and to detect bad design. These can be achieved by using software metrics, and identifying "code smells", respectively.

A. Software Metrics

Software metrics aids programmers in understanding applications, getting an overview of a large system, and identifying potential design problems [4]. Most of the metrics proposed in the literature focus exclusively on text-based languages [4], [5], [6]. For block-based languages, one promising research direction would be to focus on the set of metrics to capture both structural quality and visual organization.

Since the visual aspect is prominent for block-based languages, research questions should seek to answer how metrics can be formulated to capture the software quality concerning with the visual organization of the code. As suggested by Conversy [7], the representation of code should rely on the capability of the human visual system. "Programs must be written for people to read, and only incidentally for machines to execute[8]." Aside from the spatial layout enforced by languages' syntax, it would be interesting to explore how one can measure the design quality (e.g., how the readability of block-based programs is affected by visual grouping of scripts.)

B. Code Smells

The concept of "code smells" is commonly used to identify various structural characteristics of software indicative of design problems. The most notable work in this area is the refactoring book by Fowler [9], which documents code smells and the corresponding refactoring steps to remove them. Evidence from both educators and researchers [10], [3], [11] points out both structural (e.g., spaghetti code, duplicate code) and visual organization shortcomings (e.g., the scattering of small scenario-based scripts that cause "a confusing visual effect"), which are prevalent in the Scratch language family.

For block-based languages, language-independent code smells (e.g., *Long Script, Duplicate Code, etc.*) can be adapted from the existing ones and more specific smells can be formulated to capture the design shortcomings unique to block-based language features for both structural and visual organization aspects. A set of software metrics described previously can be composed to capture high-level smells similar to [4]. To automate the task of discovering code smells or refactoring opportunities, a targeted research effort would need to focus on formalizing the code smells and analyzing statistically the thresholds of the metrics used. Realizing these research ideas as a general framework for building automated code smell detection tools for programming environments for blocks can provide a practical tool for a wide programming audience.

III. Software Refactoring Support for Block-Based Languages

As discussed above, software refactoring[12] has become an intrinsic component of any non-trivial software development effort. The quality of all software tend to degrade over time, unless special efforts are put in place to actively counteract the process. Software engineered in block-based languages is subject to the same quality pressures [3], [11]. Practical refactoring hinges on automated program transformation tools, as refactoring by hand is often tedious and error-prone.

Figure 1 shows an example of a manual refactoring that extracts a custom block (a.k.a. *Extract Method*), one of the most commonly used refactoring transformations in text-based development environments[1]. The simple Snap! program segment in Figure 1 makes a character jump differently depending on which keyboard (space bar or up arrow) is pressed. Though seemingly simple, the code in the *before* version is not easy to read—a fragment of code represents jumping, yet it may not be so obvious to the programmer trying to understand it. Adding comments to the source code to clarify its meaning is sometimes used as a simple fix, often a good indication of poor code readability. What we have here is a unit of potentially reusable functionality that can be used in multiple places in the program. Unfortunately, when not encapsulated within a method, a code fragment can only be reused by means of copy-and-paste, thus replicating a code fragment in several places within a project. This repeated sequence of code is a common "code smell" known as *Duplicate Code*. As a result, not only the code is hard to read, but it is also hard to change, as the changes need be made everywhere in all duplicates (e.g., consider the changes one would need to make to add a gravity effect to the jump). These issues would be further exacerbated in the context of a large, real-world program.

In contrast, the code in the *after* version is improved with respect to readability and reusability. In particular, when a custom block is given a descriptive name, the improved code is self-documenting, with a custom block name explaining the high-level intent of the code fragment. The resulting program is easier to read, with custom block names serving as code comments. Extracting a code fragment into a custom block increases the chance of it getting reused elsewhere in the project.

One can see how simple refactoring transformations can improve program comprehension and reusability, reducing both code duplication and complexity. Unfortunately, modern programming environments for blocks have limited support for automated program transformation, required to support refactoring. As a result, programmers can only refactor their block-based programs by hand.

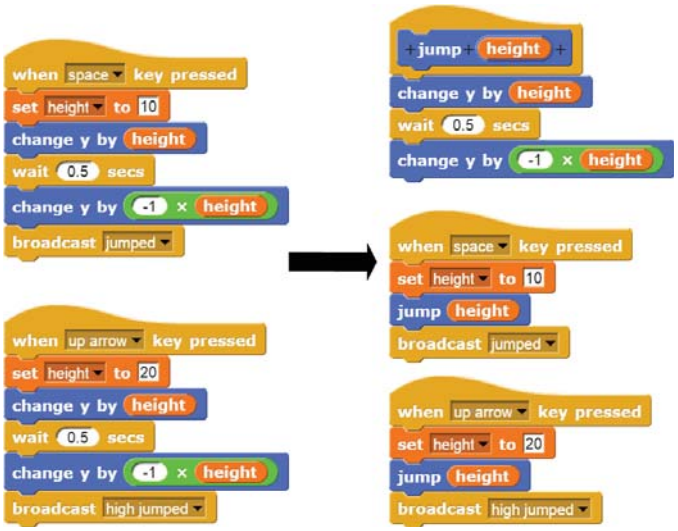


Fig. 1. An example of the code *before* and *after* performing (by hand) the *Extract Custom Block* refactoring in Snap!

What constitutes a conceptual challenge is formulating a set of systematic guidelines to determine which refactoring techniques are necessary for block-based programming and should be exposed as automated refactoring transformations within the development environments. In other words, which design problems plague block-based software most commonly, and which of these problems are amenable to a refactoring treatment. Because the refactoring research has almost exclusively focused on text-based languages, the designers of programming environments for blocks find themselves in need of guidelines tailored specifically for them.

Another important research effort in that realm should focus on the creation of internal program representations and analysis tools in support of refactoring. Although some block-based languages can be represented as text, analyzing the textual representation may not be sufficient for the objectives at hand. What is needed are internal representations that reflect not only the semantic, but also the visual organization of a block-based program. In other words, the refactorings should be mindful of how blocks are displayed on the screen, so as to increase program readability. A promising direction is to use the refactoring techniques developed for text-based languages as a starting point, and then modify and enhance them as necessary for the needs of block-based programming.

IV. Refinement-Centric Approach to Block-Based Software Development

The pedagogical effectiveness of block-based languages and their development environments is evident. These languages, including Scratch, Snap!, and Blockly, have witnessed increasing adoption as a means of teaching introductory programming. Unfortunately, with the current lack of powerful refactoring support as discussed above, programming environments for blocks can inadvertently condition novice programmers to view programming as a single-step linear process, unaligned with the iterative process of enhancement and improvement, an intrinsic part of modern software development.

Although programmers can always refactor their block-based programs by hand, the process can be tedious, error-prone, and in some cases equivalent to rewriting from scratch in complexity. As programs grow in complexity, programmers are likely to forgo even simple refinements, thus adversely impacting software quality with respect to modularity, cohesion, coupling, etc. Forgoing refinements is an ill-conceived programming habit that gives rise to potentially insurmountable difficulties in comprehending, maintaining, and reusing software.

Making refactoring tools available for block-based languages will highlight the importance of continuous code improvement in the minds of novice programmers, thereby improving the pedagogical effectiveness of block-based programming. Thus, the outlined research directions are concerned with instantiating the theoretical bases of refactoring in block-based languages and systematically evaluating the developed technologies with diverse student audiences.

References

[1] E. Murphy-Hill, C. Parnin, and A. P. Black, "How we refactor, and how we know it," *IEEE 31st International Conference on Software Engineering*, pp. 287–297, 2009. [Online]. Available: http://ieeexplore.ieee.org/lpdocs/epic03/wrapper.htm?arnumber=5070529

[2] K. Beck, *Extreme programming explained: embrace change*. Addison-Wesley Professional, 2000.

[3] O. Meerbaum-salant, R. Israel, and M. Ben-ari, "Habits of Programming in Scratch," 2011.

[4] M. Lanza, R. Marinescu, and S. Ducasse, *Object-Oriented Metrics in Practice*. Secaucus, NJ, USA: Springer-Verlag New York, Inc., 2005.

[5] K. K. Chahal and H. Singh, "Metrics to study symptoms of bad software designs," *SIGSOFT Softw. Eng. Notes*, vol. 34, no. 1, pp. 1–4, Jan. 2009. [Online]. Available: http://doi.acm.org/10.1145/1457516.1457522

[6] K. A. M. Ferreira, M. A. S. Bigonha, R. S. Bigonha, L. F. O. Mendes, and H. C. Almeida, "Identifying thresholds for object-oriented software metrics," *J. Syst. Softw.*, vol. 85, no. 2, pp. 244–257, Feb. 2012. [Online]. Available: http://dx.doi.org/10.1016/j.jss.2011.05.044

[7] S. Conversy, "Unifying textual and visual: A theoretical account of the visual perception of programming languages," in *Proceedings of the 2014 ACM International Symposium on New Ideas, New Paradigms, and Reflections on Programming & Software*, ser. Onward! 2014. New York, NY, USA: ACM, 2014, pp. 201–212. [Online]. Available: http://doi.acm.org/10.1145/2661136.2661138

[8] G. Sussman, H. Abelson, and J. Sussman, "Structure and interpretation of computer programs," 1983.

[9] M. Fowler, K. Beck, J. Brant, and W. Opdyke, "Refactoring: Improving the design of existing code."

[10] "Scratch project forum discussion," http://scratched.gse.harvard.edu/discussions/computer-science-education/what-are-worst-scratch-programming-practices, accessed:2015-07-20.

[11] M. Gordon, A. Marron, and O. Meerbaum-Salant, "Spaghetti for the main course?: Observations on the naturalness of scenario-based programming," in *Proceedings of the 17th ACM Annual Conference on Innovation and Technology in Computer Science Education*, ser. ITiCSE '12. New York, NY, USA: ACM, 2012, pp. 198–203. [Online]. Available: http://doi.acm.org/10.1145/2325296.2325346

[12] T. Mens and T. Tourwe, "A survey of software refactoring," *IEEE Transactions on Software Engineering*, vol. 30, no. 2, pp. 126–139, Feb. 2004. [Online]. Available: http://ieeexplore.ieee.org/lpdocs/epic03/wrapper.htm?arnumber=1265817

Transparency and Liveness in Visual Programming Environments for Novices

Steven L. Tanimoto
Dept. of Computer Science and Engineering
University of Washington
Seattle, WA, 98195, USA
tanimoto@cs.washington.edu

***Abstract*—Blocks-based programming environments offer an alternative program representation to textual source code that simplifies the considerations of syntax for novices. This position paper raises the question of whether additional affordances related to transparency of data, transparency of semantics, and liveness of execution can be consistently added to these environments to obtain some significant advantages for these novice programmers.**

Keywords—programming environment, learning to code, novice programming, blocks-based, transparency, liveness, visual programming, data factory, live coding.

I. Introduction

Blocks-based programming languages represent the computer program with a diagram that shows juxtaposed rectangular or prismoidal blocks in an assemblage [1, 4]. Legal program syntax is hinted and/or enforced through mechanisms such as jigsaw-puzzle-style interlocks, color coding, icon coding, and the shapes of nested containers. These clues and constraints provide pedagogical scaffolding that permits novice programmers to shoulder a reduced cognitive load when building and editing programs.

Additional affordances, especially transparency of data and semantics, and livenes, can further reduce the cognitive load. Blocks-based languages support these less consistently than they do program syntax. Better support for them could further facilitate programming by novices.

II. Transparency of Data and Semantics

Transparency of data in a programming environment refers to the visibility of program state. It traditionally has been minimal, and debugging tools have been needed to see any value that was not explicitly printed by the program or reflected in the graphical display as a result of extra coding for this purpose. Transparency of data is helpful in understanding what a program is doing. Another level of transparency can be called semantic, and it refers to the functionality that is or is intended to be implemented in a program. Affordances for data transparency are one means towards semantic transparency, because semantics deals with change in state. But other approaches include the use of suggestive icons for program constructs, and program animation.

The lack of transparency of data was long a challenge for novice programmers, and they were encouraged to sprinkle print statements throughout their code in order to find out what the values of their variables would be during execution. With the use of microworlds in novice programming environments such as Logo, Agentsheets, and many newer systems, the values of variable have often been directly reflected by the state of the microworld display -- the turtle's position, for example. Still, most systems hide variable values by default, which typically imposes another cognitive load on novices. The issue is, therefore, the question of how to lower that cognitive load by altering the environment. In my "Data Factory" system of 2003, I took transparency to an extreme: all values are always visible. This doesn't scale to large programs, but it gives a consistent affordance at small scales. Finding the right balance for various learning scenarios is the issue.

III. Liveness

Liveness of execution, as a characteristic of a programming environment, relates to the latency between editing operations (such as changing a programming construct in a program) and the visible change in the program's behavior as a result of the edit. An example of a novice programming environment that incorporates a form of liveness is Sonic Pi, developed by Sam Aaron for the Raspberry Pi and in use by many young, novice coders in the U.K. The ability of Sonic Pi to support live coding (programming of music synthesis in front of an audience) is a major selling point contributing to the environment's wide adoption. My position is that the best programming environments for novices will tend to provide multiple forms of scaffolding. Blocks-oriented environments do particularly well on scaffolding for syntax. However, they incorporate these other aspects with varying degrees of success.

Liveness is another feature that can potentially reduce the cognitive load of programming for novices. In traditional environments, there is a significant time delay between programming act (e.g., adding a statement, changing a constant's value) and seeing its effect on execution. The delay could be arbitrarily long -- so long that many edits might be made before seeing the effect of any one of them. This means that the set of possible causes for a bug, in the mind of the

978-1-4673-8367-7/15/$31.00 ©2015 IEEE

programmer, could be quite large. Lowering that latency not only caters to short attention spans but reduces the cognitive load of debugging -- typically lowering the set of seemingly plausible causes for the bug. Not only that, liveness helps the programmer to quickly understand the effect of a program change on execution. How can novice programming environments such as blocks-based ones be made more "live?" A variety of techniques are possible. Some of these were covered in my recent keynote at the International Conference on Live Coding held during July 2015 at Leeds, UK. Techniques include using implicit loops around the code in focus, secondary executions, predictive execution, and automatic checkpointing of execution to responsively support an evolving program semantics.

REFERENCES

[1] E. P. Glinert, "Towards "second generation" interactive, graphical programming environments," in 2nd IEEE Computer Society Workshop on Visual Languages, 1986, pp. 61–70.

[2] S. L. Tanimoto, "Programming in a Data Factory," Proc. IEEE Symposium on Visual Languages and Human-Centric Computing (VL/HCC'03), Oct 2003.

[3] S. L. Tanimoto, "A perspective on the evolution of live programming." Proc. LIVE 2013, workshop on Live Programming, San Francisco.

[4] F. Turbak, S. Sandu, O. Kotsopoulous, E. Erdman, E. Davis, and K. Chadha, "Blocks languages for creating tangible artifacts," in IEEE Symposium on Visual Languages and Human-Centric Computing (VL/HCC '12), Oct. 2012, pp. 137–144.

Visual Debugging Technology with Pencil Code: Position Paper

Amanda Boss
Harvard College
aboss@college.harvard.edu

Cali Stenson
Wellesley College
cstenson@wellesley.edu

Jeremy Ruten
University of Saskatchewan
jeremy.ruten@gmail.com

ABSTRACT

Pencil Code, a web-based program, utilizes blocks to aid students in creating code. This paper presents work creating a debugging tool to further improve Pencil Code and create an environment where students can better learn to understand code by having a way to visually trace its operations.

1. INTRODUCTION

First-time programmers do not naturally understand how to debug programs, so we propose a tool which helps beginners trace and debug the execution of their code without requiring manual debugging with breaks and print statements. Often times, when students are debugging their programs for the first time, they have the most difficulty with tracing through the program. When Fitzgerald et al. conducted a study, they found that a majority (57%) of the students had the most difficulty with finding the problem with the code [1]. In the same study, structured interviews with the students showed that the most commonly mentioned strategy for debugging code was tracing followed by testing and isolating the problem [1]. Even when first-time programmers do fix one of the bugs in their programs, they often have difficulty applying those same debugging principles for future bugs [2]. This stems from an inability to uniformly isolate bugs and understand code flow [2]. Our tool automates the process of isolating problems in the code and tracing code to offer a dynamic learning experience for the user.

2. OVERVIEW

Our debugging tool is a slider element that depends on a tracing transpiler and works in tandem with code annotations. Figure 3 shows the interaction between the three components.

2.1 Pencil Code

Our debugging tool is integrated into Pencil Code [3], a block-based online programming tool. Pencil Code allows users to convert the blocks into text and enter an open sandbox where they can create any program that is possible with CoffeeScript, JavaScript, and HTML. The environment is ideal for learning and is geared towards first-time programmers of a wide range of ages.

2.2 Tracing Transpiler

To help students see how their code is connected to what is happening when it runs we trace each line as it is animated on the canvas and allow students to highlight a given line to show what the animation looked like when that line ran. Tracing is a subtle, but useful, visual that allows students to see how their program works.To get a complete trace of a student's program as it runs, we pass the student's program through a code instrumenter before running it. Each statement in the program is instrumented with two function calls, one before the statement and one after, that send the location of that statement, as well as values of variables and function calls that are being tracked, to a listener that collects these events into an array containing the full trace of the program.

For example, if the following program is instrumented,

```
var x = 2;
var y = x + x;
```

then the code instrumenter will output a program that looks something like this:

```
ide.trace({type: 'before', line: 1,
vars: [{name: 'x', value: x}]});
var x = 2;
ide.trace({type: 'after', line: 1,
vars: [{name: 'x', value: x}]});
ide.trace({type: 'before', line: 2,
vars: [{name: 'y', value: y}, {name: 'x', value: x}]});
var y = x + x;
ide.trace({type: 'after', line: 2,
vars: [{name: 'y', value: y}, {name: 'x', value: x}]});
```

The code instrumenter works by parsing the input program into an abstract syntax tree (AST), and then adding the instrumented code before and after each node in the AST that is recognized as a statement. The AST is also used to find all variables and function calls within a statement, so that they can be tracked. Variables are tracked by passing

978-1-4673-8367-7/15/$31.00 ©2015 IEEE

their name and value directly into each event. Return values of function calls are tracked by assigning each function call to a temporary variable, and passing the value of the temporary variable into the event. Once the program has been fully instrumented in AST form, it is compiled and run in the Pencil Code environment. As trace events are raised by the instrumented program, we use the information in each event to visualize what the program is currently doing, and allow the student to go "back in time" to any event to see what state their program was in at that time.

2.3 Visualization of Events

One of the primary features for the visual debugger is an interactive slider element that allows users to have an in-depth understanding of code flow. When the code in the editor panel is compiled and there exists more than two lines of code, the slider element appears. The line number requirement is to ensure sufficient complexity that warrants a debugging slider element. Each "tick" on the slider element represents an event that was traced by the compiler as a result of the inputted code. As users drag and click through the slider they are able to see the corresponding line of code highlighted as well as see the canvas reflect what the turtle animations looked like at that line of code. As users are scrolling through the slider, they are also able to see the arrows and variable tracking that occurred at that point in the code. The goal of this feature is to not only understand the sequence of events that occurred in the users' code, but to allow the users to grasp the fundamentals of important programming concepts. These concepts include, but are not limited to, looping and recursion. As users update their code and re-run their program, the slider element is also updated to reflect their changes. The slider works by keeping track of the state of events as code is run and storing those events so that they can be replayed after the code has run. The events from the transpiler are given to the slider and the slider indexes the events as a new value is selected on the slider and shows a visual of which line was run and what was happening on the screen at that line.

2.4 Code Annotations

Another feature we developed are arrows that help students understand code flow. The code annotations that show arrows were implemented to help students understand how lines of code are being processed. Arrows are deployed by the debugger when it traces an event on a line that does not immediately follow the line before it, as seen in Figure 1.The arrows were created with the idea that students would better understand loops and function calls if an arrow was drawn to direct their attention to the jump in lines since one would normally expect lines to run in order when new to programming. Arrows point out the repetitive nature of loops as well as the way code inside of functions is run on a function call. The arrows are designed to flash on the screen as code runs and to show when the slider highlights a line that an arrow is drawn from. Arrows are assigned to pairs of events from the transpiler that are nonlinear in code lines. When the slider shows the arrows those events that have had an arrow assigned to it will call the arrow function and re-deploy the arrows the user saw during runtime. Variable tracking allows us to show the values of variables as they are updated and passed into functions. As seen in figure 2, we show the variables as a pop-up on the right-hand side of a line of code

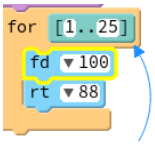


Figure 1: Arrows in Block Mode

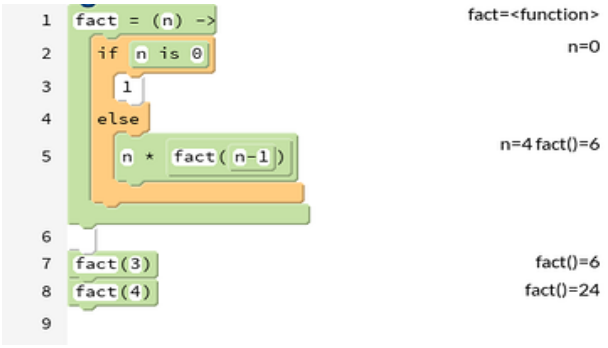


Figure 2: Variable tracking with a factorial function in Pencil Code

so that users can see what the value of their variable was at a given point in runtime. Every variable used in an expression is displayed for the user alongside that expression. The return values of function calls used in the expression are also displayed. These return values allow the user to be aware of every value going into each expression, helping them track down where bad values might be coming from when they have a bug in their code. In the case of assignment statements, the old value of the variable is displayed before the statement executes, and the displayed value updates with the new value when the statement finishes (the debugger may have stepped into multiple functions in the meantime).

3. RELATED WORK

There are a variety of online programs to help users with the debugging process. We discuss those that influenced our work.

3.1 Python Tutor

One of the most prevalent programs is Python Tutor, a visualization web-embedded tool that allows users to step through the code at execution and view elements currently in the stack and heap [4]. This concept of tracing through the code at execution has proven to be successful in the debugging process as it mimics what a teacher would explain to students on a whiteboard. However, no version of Python Tutor for other languages such as JavaScript,which gives first-time programmers more flexibility to create web-based programs, currently exists. Furthermore, the audience of Python Tutor is geared towards students in an introductory computer science class [4]. Pencil Code is designed to be a self-learning tool for all ages; thus, we have created a more accessible debugger tool than that offered by Python Tutor.

3.2 Gidget

Another debugging tool, primarily geared towards a younger audience, is a web-based game called Gidget [5]. The game

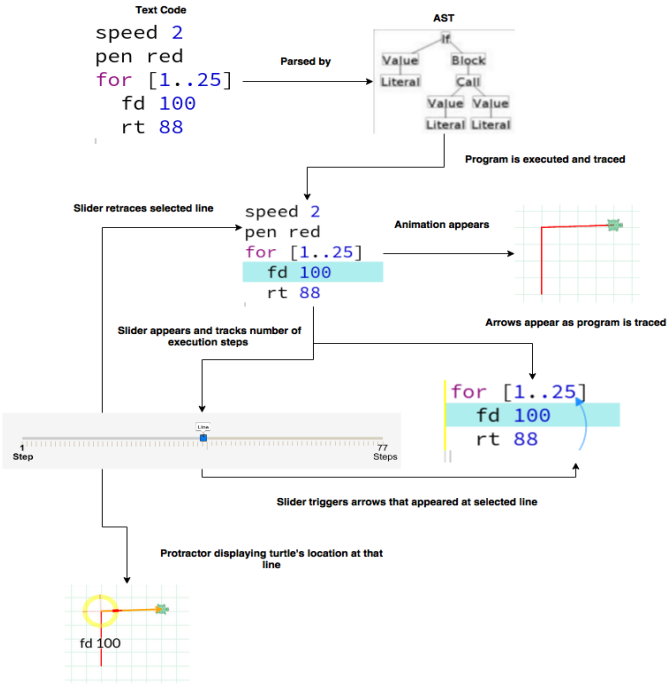


Figure 3: Debugging Flow in Pencil Code

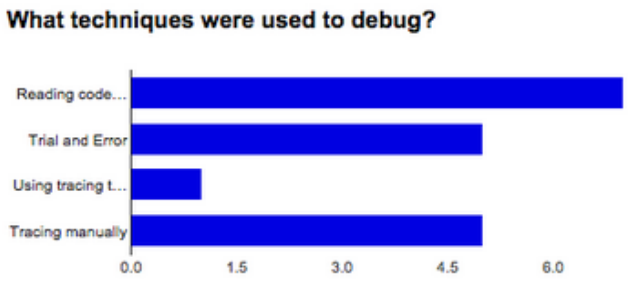


Figure 4: Results from Survey

consists of a variety of levels centered around helping a character called Gidget. In order to pass each level, users must fix the bug in the code. Users are equipped with a variety of tools such as tracing through the program and going step by step. By gamifying the debugging experience as well as making it more interactive, Gidget is able to appeal to younger first-time programmers [6]. However, Gidget does not support an open sandbox environment for users like Pencil Code.

4. CONCLUSION

Pencil Code is an open sandbox and our debugger encourages students to fearlessly create programs and see how they work. Our debugger will allow students to improve their programs without the learning curve of understanding breakpoints and logging to help figure out programming mistakes. With the support of our debugger, students can watch as their variables change, see which line of their code is running, and re-watch how their code works. These features create an environment that teaches you how to understand how your code behaves. Knowing how their code behaves will help students figure out what they should change if it does not work in the way they expect. With the debugger, users of Pencil Code can learn to write more complicated programs and become experienced programmers who program easily without the support of a blocks structure.

5. FUTURE WORK

To fully understand the effect our tools have on students we would like to get more user feedback on our debugger tools. This feedback will come from future studies and user responses to our features when they are in production on pencilcode.net. In addition to understanding how people use and appreciate current features, there are myriad of features that we hope to add to our debugging tool. One of the observations that we made during our user study was that participants were more likely to use buttons rather than a slider UI. Similar to Python Tutor, we plan on adding buttons that will perform the same task as the slider and will allow users to go forward and back one step in the program.

6. REFERENCES

[1] Fitzgerald, S., McCauley, R., Hanks, B., Murphy, L., Simon, B., and Zander, C. Debugging From the Student Perspective. *IEEE Transactions On Education, 53* (3). 390-396.

[2] Ahmadzadeh, M., Elliman, D., and Higgins, C. An Analysis of Patterns of Debugging Among Novice Computer Science Students. in *Innovation and technology in computer science education: Proceedings of the 10th annual SIGCSE conference,* (Libson, Portugal, 2005), 84-88.

[3] Bau, D., Bau, D.A., Dawson, M., and Pickens, S.C. Pencil code: block code for a text world. in *Interaction Design and Children: Proceedings of the 14th International Conference,* (Massachusetts, USA 2015), 445-448.

[4] Guo, Philip. Online Python Tutor: Embeddable Web-Based Program Visualization for CS Education, March 2013. Retrieved July 22, 2015,from Python Tutor: http://pythontutor.com.

[5] Lee, M.J. Gidget: An online debugging game for learning and engagement in computing education. in *IEEE Symposium on Visual Languages and Human-Centric Computing,* (Melbourne, Australia 2014), 193-194.

[6] Lee, M.J. and Ko, A.J. Personifying programming tool feedback improves novice programmers' learning in *Proceedings of the seventh international workshop on computing education research,* (Rhode Island, USA 2011), 109-116.

Position Statement: App Inventor Instructional Resources for Creating Tangible Apps

Krishnendu Roy
Department of Mathematics and Computer Science
Valdosta State University
1500 N. Patterson St., Valdosta, GA 31698
Email: kroy@valdosta.edu

***Abstract*—App Inventor is one of the most popular block-based programming environments. Currently, there are limited instructional resources that guide students to create tangible apps using App Inventor. In this positional statement we make a case for the need for more App Inventor instructional resources related to tangible apps. We also present our proposal to address this need.**

I. Introduction

App Inventor is currently one of the most popular block-based programming environments. The main goal of App Inventor is to teach computing and programming to students with limited prior programming knowledge and to democratize app creation by providing an easy-to-learn environment. App Inventor has 3 million registered users with close to $100,000$ unique weekly active users. To date, students have created more than 8 million apps using App Inventor.

App Inventor has experienced broad adoption in diverse venues. Researchers have used App Inventor in summer camps and other outreach activities for K12 students for several years now [1], [2], [3], [4], [5]. App Inventor has also been effectively used for professional development workshops for K12 teachers [6], [7], as well as introductory computing courses at the college level [8], [9], [10]. Extremely popular app design competitions like the Verizon Innovative App Challenge and Technovation Challenge also use App Inventor.

Most apps designed using App Inventor run on an Android phone or tablet. However, it is possible to create apps that can go beyond just the virtual realm and somehow interact with the physical world. Examples of such apps that others have already created using App Inventor are Lego Mindstorms robot controller apps, apps that interface with an Arduino board etc. We define such apps as *tangible apps*.

There are many instructional materials available for learning computing and programming using App Inventor, and they range from books to video tutorials. Video tutorials with written instructions are ideal for fast-paced outreach activities like after-school/weekend workshops and summer camps. However, there are very few App Inventor instructional resources that guide a student to create a tangible app. Most of the information available in this area is scattered over multiple posts in the App Inventor Forum Google group and there in no easily-accessible centralized list of resources.

In this project we attempt to address this void by:

1) Creating a master list of instructional resources related to tangible apps created using App Inventor.
2) Modifying existing instructional resources to make them more suitable for K12 outreach and creating new resources to augment the existing ones.

II. Motivation for Tangible Apps

We have organized computing summer camps at our university using App Inventor for the past five summers. From our anecdotal experience, one of the most popular and enjoyable App Inventor app that the students created was the Lego Mindstorms NXT robot controller app. We believe the "coolness" factor of an app interacting with the physical world cannot be matched by non-tangible apps.

Summer camps organized at Clemson University also reported similar experience in [2]. The most popular App Inventor app in their camp was a treasure-hunt game app that involved tangible interaction using QR codes.

Other recent research projects have also developed block-based programming environments that generate tangible artifacts and interactions. Turbak et al. in [11] reported about a block-based programming environment that allows users to create tangible laser/vinyl cut artifacts. Ardublocks[1] is another project that lets users program Arduino microcontrollers using a block-based interface. Deitrick et al. in [12] reported about BlockyTalky, another block-based programming environment supporting tangible interaction which is also focuses on parallel and distributed programming. Scratch, one of the other most popular block-based programming environments has supported tangible interactions through PicoBoards for a while now. Tickle[2] is another block-based programming environment offering tangible output. Users can use Tickle to program Arduino boards, robots, and even drones and smart home devices. Spherly[3] is a block-based programming environment to program Sphero robots. When first introduced in 2014, Google's *Made with Code* project[4] allowed users to create personalized bracelets using a block-based programming environments. These bracelets were then 3-D printed and shipped to the users for free. Another current block-based tool of that same project allows users to create t-shirt designs with embedded LED lights.

[1] http://blog.ardublock.com
[2] https://tickleapp.com/en-us/
[3] http://outreach.cs.ua.edu/spherly/
[4] https://www.madewithcode.com/projects

978-1-4673-8367-7/15/$31.00 ©2015 IEEE

Based on the success of these projects, we think block-based programming environments generating tangible artifacts and interactions will become increasingly popular in near future. Hence, tangible apps created using App Inventor are an important educational resource and more work needs to be done in streamlining the existing instructional resources and developing new ones.

III. Proposed Work

We plan to achieve two main goals in this project. Our first goal is to create an easily-accessible master list of all the existing instructional resources related to tangible App Inventor apps. To come up with the master list, we mainly plan to look in to the App Inventor website, the App Inventor Forum and the App Inventor Gallery.

Our second goal is to a) make sure that the instructional resources included in the master list are suitable for fast-paced K12 outreach activities like a summer camp, and b) create new instructional resources. Summer camps are usually one week long. In this limited time, there is not much scope for learning details of all aspects of programming. Hence, one of the goals of summer camps is to make the students interested in computing and enhance their positive attitude towards computing and programming so that they continue to explore computing in future. Since many of the students in a summer camp will be learning programming for the first time, confusing instructional resources can negatively impact their attitude towards computing in general. Additionally, many camps are organized with the help of K12 teachers who might be inexperienced in App Inventor themselves. Hence, easy-to-understand instructional resources with clear concise directions are very important for success of an outreach activity like a summer camp.

To achieve this second goal, we plan to modify some of the existing resources. Some of the existing resources were designed using App Inventor Classic. We plan to re-implement them using the latest version of App Inventor. Some other resources lacks detailed description. In those cases we plan to create detailed descriptions.

Additionally, we plan to create some new tangible apps and associated instructional resources. We have some preliminary ideas about what tangible apps we want to create:

- Lego Mindstorms robots are one of the most popular robotics kits and are widely used in K12 classrooms. App Inventor had blocks to interface with the NXT (previous) version of the Lego robots. But, App Inventor does not support the latest version, Lego EV3 robots. We want to see if we can build that support in App Inventor or not.
- NFC tags have become relatively inexpensive now. So we plan to create some apps leveraging App Inventor's NFC blocks.
- Finally, we plan to create some more apps that can control an Arduino. For example, one of the app that we plan to develop will enable an user to open and close window blinds using a smartphone. A second app will use an Arduino to light up an LED with different colors to denote how hot/cold a day's temperature will be.

We plan to create written as well as video tutorials for all these apps. We also plan to create a list of possible variations for each app so that students can further explore their creations.

IV. Conclusion

In this position statement we make a case for increasing popularity of tangible apps and the need to develop more instructional resources that demonstrates how to create tangible apps using App Inventor. We believe tangible apps are going to be more common in future. Hence, instructional resources related to tangible apps will be an exciting augmentation to the current set of App Inventor instructional resources. Successful completion of this project will open the door to new opportunities like organizing tangible-apps related summer camps and organizing teachers' workshops similar to Google's CS4HS workshops, with the main focus being tangible apps.

References

[1] S. AlHumoud, H. S. Al-Khalifa, M. Al-Razgan, and A. Alfaries, "Using app inventor and lego mindstorm nxt in a summer camp to attract high school girls to computing fields," in *Global Engineering Education Conference (EDUCON), 2014 IEEE*. IEEE, 2014, pp. 173–177.

[2] M. H. Dabney, B. C. Dean, and T. Rogers, "No sensor left behind: enriching computing education with mobile devices," in *Proceeding of the 44th ACM technical symposium on Computer science education*. ACM, 2013, pp. 627–632.

[3] B. Ericson and T. McKlin, "Effective and sustainable computing summer camps," in *Proceedings of the 43rd ACM technical symposium on Computer Science Education*. ACM, 2012, pp. 289–294.

[4] K. Roy, "App inventor for android: report from a summer camp," in *Proceedings of the 43rd ACM technical symposium on Computer Science Education*. ACM, 2012, pp. 283–288.

[5] 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*. ACM, 2013, pp. 621–626.

[6] J. Gray, H. Abelson, D. Wolber, and M. Friend, "Teaching cs principles with app inventor," in *Proceedings of the 50th Annual Southeast Regional Conference*. ACM, 2012, pp. 405–406.

[7] J. Liu, C.-H. Lin, P. Potter, E. P. Hasson, Z. D. Barnett, and M. Singleton, "Going mobile with app inventor for android: a one-week computing workshop for k-12 teachers," in *Proceeding of the 44th ACM technical symposium on Computer science education*. ACM, 2013, pp. 433–438.

[8] H. Abelson, R. Morelli, S. Kakavouli, E. Mustafaraj, and F. Turbak, "Teaching cs0 with mobile apps using app inventor for android," *Journal of Computing Sciences in Colleges*, vol. 27, no. 6, pp. 16–18, 2012.

[9] E. Spertus, M. L. Chang, P. Gestwicki, and D. Wolber, "Novel approaches to cs 0 with app inventor for android," in *Proceedings of the 41st ACM technical symposium on Computer science education*. ACM, 2010, pp. 325–326.

[10] D. Wolber, "App inventor and real-world motivation," in *Proceedings of the 42nd ACM technical symposium on Computer science education*. ACM, 2011, pp. 601–606.

[11] F. Turbak, S. Sandu, O. Kotsopoulos, E. Erdman, E. Davis, and K. Chadha, "Blocks languages for creating tangible artifacts," in *Visual Languages and Human-Centric Computing (VL/HCC), 2012 IEEE Symposium on*. IEEE, 2012, pp. 137–144.

[12] E. Deitrick, J. Sanford, and R. B. Shapiro, "Blockytalky: A low-cost, extensible, open source, programmable, networked toolkit for tangible creation," in *Proceedings of Conference on Interaction Design for Children, Aarhus, Denmark*, 2014.

Approaches for Teaching Computational Thinking Strategies in an Educational Game: A Position Paper

Aaron Bauer, Eric Butler, Zoran Popović
Center for Game Science
Computer Science & Engineering
University of Washington
Seattle, WA 98195
Email:{awb, edbutler, zoran}@cs.washington.edu

***Abstract*—Computer science is expanding into K12 education and numerous educational games and systems have been created to teach programming skills, including many block-based programming environments. Teaching computational thinking has received particular attention, and more research is needed on using educational games to directly teach computational thinking skills. We propose to investigate this using *Dragon Architect*, an educational block-based programming game we are developing. Specifically, we wish to study ways of directly teaching computational thinking strategies such as divide and conquer in an educational game, as well as ways to evaluate our approaches.**

***Keywords*—block-based programming, game-based learning, computational thinking, CS education**

I. Introduction

Teaching *computational thinking* has been a focus of recent efforts to broaden the reach of computer science education, including those using block-based programming environments. In their review of recent literature on teaching computational thinking, Lye and Koh [1] use Brennan and Resnick's definition of computational thinking as consisting of *concepts*, *practices*, and *perspectives* [2]. Concepts are basic programming ideas (such as variables, conditionals, and loops), practices are the problem-solving strategies used while programming (such as "being incremental and iterative" or "using abstraction and modularization"), and perspectives are the relationships with the wider technological world. Lye and Koh conclude that more research is needed on teaching computational thinking *practices* in particular. How to *directly* teach such skills in is an open problem, including in the context of a block-based programming environment. Teaching skills directly involves providing appropriate guidance and scaffolding to help students acquire those skills, rather than just exposing them to environments where those skills are necessary.

In this paper we propose investigating directly teaching computational practices in an educational block-based programming game called *Dragon Architect*[1]. We briefly discuss existing work on teaching computational thinking and the direct teaching of problem-solving strategies. We describe ways this might be attempted in *Dragon Architect*, as well as ways we might evaluate such work.

[1]Available at http://centerforgamescience.org/portfolio/dragon-architect

II. Game Description

Before we discuss approaches for teaching computational thinking, we describe basic information about *Dragon Architect* to provide context for the discussion. In *Dragon Architect*, players write code to control a dragon that builds 3D structures in a cube world. Our game, in development since spring 2014, is played in a web browser. Similar to other programming environments, the user interface is separated into two parts: an area where the player can assemble their code and a visualization of the 3D environment their code affects (see Figure 1). The game uses the block-based programming library Blockly [3] for inputing code.

The player can write programs to move the dragon in three dimensions and have the dragon place and remove cubes of various colors. In addition to blocks that control the dragon directly, players can use definite loops and procedures (see Figure 2). As players progress through the game, they alternate between short sequences of puzzles with a specific goal and a constrained set of available code blocks and an open-ended sandbox. The game begins with puzzles that introduce the idea of assembling and running code, as well as the code blocks for moving the dragon and placing cubes. After that, the player can creatively experiment and build in the sandbox and complete other puzzle sequences to make more code blocks available, switching between sandbox and puzzles at any time. In this way, the language the player uses to write instructions for the dragon gradually expands as the player advances.

The popularity and broad appeal of *Minecraft* [4] motivated our use of a 3D grid world in which the player's programs could place cubes. This choice also makes it natural to extend our game in the future with exploration, more complex interaction with the environment, or players working together in a shared world. Our playtests with *Dragon Architect* have shown the premise of programming a dragon in a *Minecraft*-like world appeals to younger players of all genders. Common sandbox activities have included making the dragon travel very long distances, building big and impressive towers, and spelling one's name out of cubes.

III. Teaching Computational Thinking Strategies

Many have studied how to increase the presence and effectiveness of computational thinking in computer science educa-

978-1-4673-8367-7/15/$31.00 ©2015 IEEE

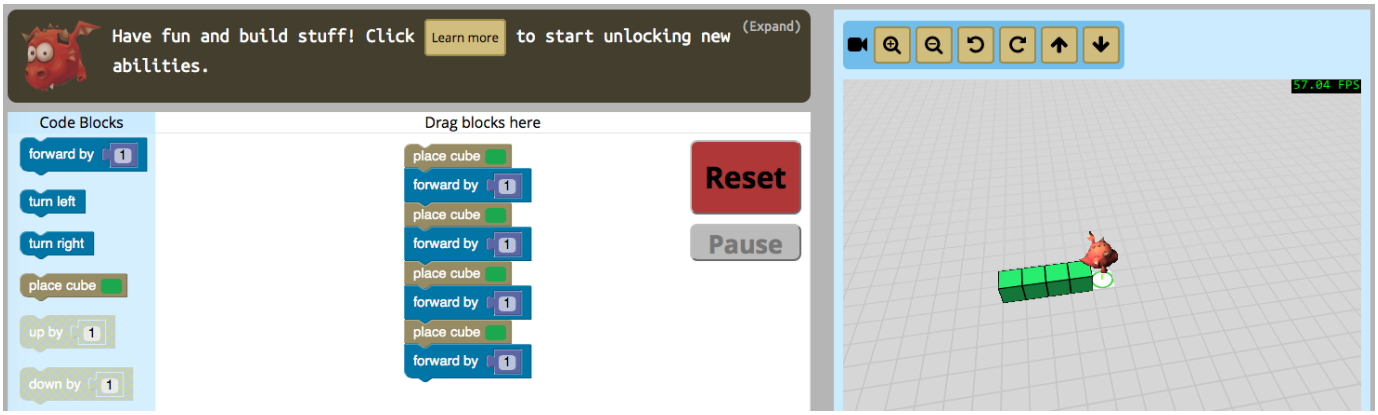


Fig. 1. The player assembles code to control the dragon on the left side, and the dragon and world it inhabits are visualized on the right side. Only a few different code blocks are available to the player initially, and more are unlocked by completing guided puzzles.

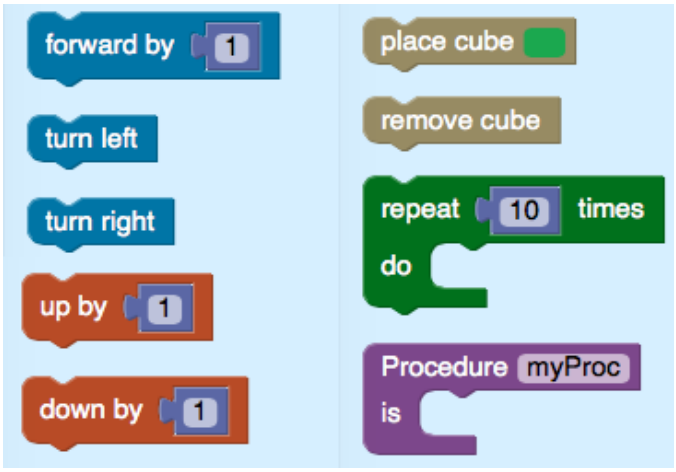


Fig. 2. The programming elements available in *Dragon Architect*, which include moving the dragon, placing blocks, definite loops, and procedure definitions.

tion and education in general (e.g., Barr and Stephenson [5], Grover and Pea [6]). Furthermore, educational games and other systems often have teaching computational thinking as an explicit goal (e.g., Weintrop and Wilensky [7], Kazimoglue et al. [8]) or have a computational thinking framework built around them (e.g., Gouws et al. [9], Computational Thinking with Scratch [10]). A recent review of the literature on teaching computational thinking found that additional empirical research is needed, especially in the case of computational thinking practices [1]. We believe the use of educational games in particular to teach computational thinking skills deserves to be the focus of more empirical work.

Specifically, we propose to investigate directly teaching computational thinking strategies in *Dragon Architect*. Simply playing in a computational environment where these strategies are necessary is unlikely to teach students such complex skills [11]. Instead, we must address how to directly teach computational thinking skills by investigating which guidance is effective and how it is best deployed in an educational game.

One computatinal thinking skill of interest is the identification and application of problem-solving strategies. A great deal of recent education research suggests that "curricula can model such strategies for students" and that appropriate guidance can "enable students to learn to use these strategies independently" [12]. Mayer and Wittrock call attention to the substantial evidence in the education literature for teaching what they call *domain-specific thinking skills* and *metacognitive skills* [13]. The former would include the ability to use a strategy like divide and conquer, and the latter would include knowing when and where to employ that strategy. In both cases, Mayer and Wittrock describe studies (for non-computer science domains) that have shown teaching these skills directly can improve learning and performance. It is an open question whether this can be applied to teaching computational thinking in a game.

One computational thinking strategy we intend to focus on in *Dragon Architect* is *divide and conquer*. One potential approach is to lead the player though a top-down deconstruction of building a castle in order to model iteratively subdividing a large problem into more manageable subproblems. The player is presented with a single code block that builds an entire castle, but discovers the construction has a number of flaws. The next several puzzles each decompose some part of the flawed program in order to give the player a chance to repair it. For example, to enable the player to give the castle the correct number of walls and towers, the castle code block is split into a tower block and a wall block that the player uses to write a corrected castle procedure, as shown Figure 3.

A companion approach is a gradual bottom-up progression modeling combining the blocks currently available to the player into more sophisticated constructs. For example, the player is tasked with writing a program to place a line of cubes. When this is completed, the player is awarded a new kind of block that by itself places a line of cubes (i.e., a block encapsulating the player's previous program). Subsequent puzzles ask the player to use the line block to construct other, more complicated structures, each time granting the player a single, encapsulating block.

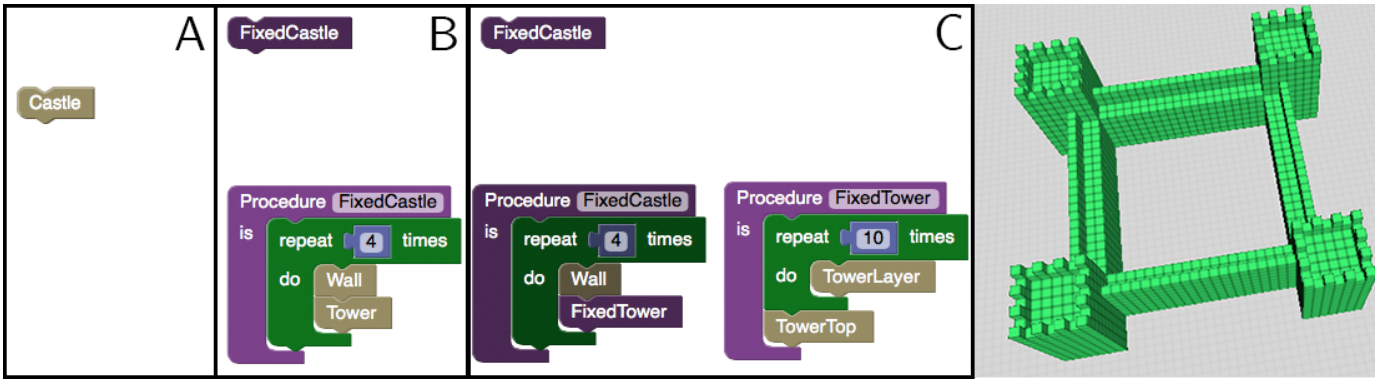


Fig. 3. The code required by a progression of levels demonstrating the strategy of divide and conquer. In A, the player uses a single code block to build an entire castle. Then, in B, the player is given an empty `FixedCastle` procedure, which they must fill with the appropriate number of wall and tower blocks. Finally, in C, the player is given a completed `FixedCastle` procedure and must fill in the `FixedTower` procedure as shown. The final completed castle is shown on the right.

Evaluating learning outcomes and other effects of these or similar approaches is a tremendous challenge. Computational Thinking with Scratch proposes three kinds of assessment: (1) artifact-based interviews, (2) design scenarios, and (3) learner documentation [10]. The interviews are intended to engage the learner in a conversation about the artifacts they have created and the practices they used. Design scenarios are a sequence of projects that challenge the learner to critique, extend, debug, and remix existing code. Finally, learner documentation focuses on engaging learners in reflection about their learning, and examples include keeping a journal, commenting code, and creating a visual walk-through of a project using screen capture software. We believe these approaches to be promising, and worthy of further study.

In addition to design scenarios, we propose that more general kinds of in-game assessments could be useful. Like Scratch, *Dragon Architect* provides users a place for unstructured creative exploration. The effectiveness of an attempt to teach computational thinking strategies could be assessed by comparing the programs written by those who completed the relevant puzzles to those who did not (after controlling for time played and differences in programs prior to completing the puzzles). A variety of other in-game metrics could contribute to an assessment including a player's solutions to specific challenges, time taken to complete puzzles, etc.

On-paper assessments (given as pre-test and post-test) could also serve as an evaluation. Computational thinking skills might be assessed through language-independent assessments of computer science knowledge [14] or general problem-solving assessments such as those developed by the Program for International Student Assessment [15]. Though neither of these assessments are explicitely targeted at computational thinking, they both contain items involving computational thinking skills. Finally, Grover and Pea suggest "academic talk" (i.e., student development and use of computational language) could be leveraged as an additional assessment of computational thinking [6].

IV. Acknowledgments

This work was supported by the Office of Naval Research grant N00014-12-C-0158, the Bill and Melinda Gates Foundation grant OPP1031488, the Hewlett Foundation grant 2012-8161, Adobe, and Microsoft.

References

[1] S. Y. Lye and J. H. L. Koh, "Review on teaching and learning of computational thinking through programming: What is next for k-12?" *Computers in Human Behavior*, vol. 41, pp. 51–61, 2014.

[2] K. Brennan and M. Resnick, "New frameworks for studying and assessing the development of computational thinking," in *Proceedings of the American Educational Research Association*, 2012.

[3] "Blockly," https://developers.google.com/blockly/, accessed: 2015-02-18.

[4] Mojang AB, "Minecraft," 2011.

[5] V. Barr and C. Stephenson, "Bringing computational thinking to k-12: what is involved and what is the role of the computer science education community?" *ACM Inroads*, vol. 2, no. 1, pp. 48–54, 2011.

[6] S. Grover and R. Pea, "Computational thinking in k–12 a review of the state of the field," *Educational Researcher*, vol. 42, no. 1, pp. 38–43, 2013.

[7] D. Weintrop and U. Wilensky, "Robobuilder: a computational thinking game," in *SIGCSE*, 2013, p. 736.

[8] C. Kazimoglu, M. Kiernan, L. Bacon, and L. Mackinnon, "A serious game for developing computational thinking and learning introductory computer programming," *Procedia-Social and Behavioral Sciences*, vol. 47, pp. 1991–1999, 2012.

[9] L. A. Gouws, K. Bradshaw, and P. Wentworth, "Computational thinking in educational activities: An evaluation of the educational game lightbot," in *Proceedings of the 18th ACM Conference on Innovation and Technology in Computer Science Education*, ser. ITiCSE '13. ACM, 2013, pp. 10–15.

[10] "Computational thinking with scratch," http://scratched.gse.harvard.edu/ct/index.html, accessed: 2015-07-23.

[11] R. E. Mayer, "Should there be a three-strikes rule against pure discovery learning?" *American Psychologist*, vol. 59, no. 1, p. 14, 2004.

[12] National Research Council, *Report of a Workshop on the Scope and Nature of Computational Thinking*. National Academies Press, 2010.

[13] R. E. Mayer and M. C. Wittrock, "Problem solving transfer," in *Handbook of educational psychology*, D. C. Berliner and R. C. Calfee, Eds. Routledge, 1996.

[14] A. E. Tew and M. Guzdial, "The fcs1: a language independent assessment of cs1 knowledge," in *Proceedings of the 42nd ACM technical symposium on Computer science education*. ACM, 2011, pp. 111–116.

[15] "Program for international student assessment," http://www.oecd.org/pisa/, accessed: 2015-07-23.

Blocks Versus Text: Ongoing Lessons from Bootstrap

Emmanuel Schanzer, Shriram Krishnamurthi, Kathi Fisler

Bootstrap
www.bootstrapworld.org

Abstract

Block languages need to be studied from many perspectives. One important viewpoint that we do not believe has been explored in the literature is the effect on teachers during professional development workshops. We have acquiring experience comparing the use of block languages against (parenthetical) textual ones in the process of training teachers for the Bootstrap curriculum. We believe our observations are not only interesting in context, but might also lead to discovering interesting new subtleties about the use of blocks.

1. Context: the Curriculum

Bootstrap is a middle-school and high-school program that combines algebra and programming. Students with no prior programming experience begin by designing a simple videogame (given a set of constraints), and then learn programming and algebra concepts to implement it. In particular, the underlying programming language used in Bootstrap is purely functional, mirroring the semantics of algebra.

In the past, Bootstrap has exclusively used functional, student-friendly subsets of Racket (comparable to a functional core of Scheme). Some institutions used the DrRacket programming environment, which is desktop-based, but most use WeScheme, a browser- and cloud-based environment. It's noteworthy that Racket and Scheme have a parenthetical syntax.

Two years ago, Code.org—a non-profit running a large national effort to popularize computing education—chose Bootstrap's curriculum as its Middle School Mathematics (MSM) module. For the first year they used the Bootstrap curriculum verbatim. However, being strong believers in block-based languages, Code.org created a block language for MSM. They based their version on a Blockly-based prototype that the authors built with a student, Spencer Gordon.

This block language is intended to go live in classes in Fall 2015. During summer 2015, we have been training teachers to prepare for this curriculum. Thanks to our extensive experience training Bootstrap teachers with textual languages and the similar activities used in both curricula, we have been able to compare the experiences with these two languages in an apples-to-apples fashion. This document presents some of our preliminary observations, which we believe would be interesting to discuss at the workshop. (We believe the issue of *training teachers with blocks* has not been studied previously in the literature.)

2. Observations

- Blocks are more of an all-or-nothing pedagogical approach than we anticipated. The pedagogy of Bootstrap makes extensive use of on-paper exercises (designed to complement and extend math teachers' existing practices, and also useful in school settings with limited computing equipment). However, blocks do not lend themselves well to paper, and teachers end up wanting a "paper syntax" for writing programs. Our own experience with Bootstrap has found that the on-paper exercises are an essential ingredient for learning, and Code.org has also decided to provide an (optional) paper workbook. However, this imposes two syntactic constraints for the language used on paper:
  - The paper syntax must be as close to the block design as possible. Otherwise, the translation steps from problem to paper to code become a point of friction.

    For example: Math notation includes ternary expressions ($50 < x < 100$), whereas a particular programming language may require students to translate that expression into three binary operators ($50 < xandx < 100$). This transition has been problematic with our block language. In contrast, it has not been problematic in Bootstrap, which also uses binary comparison and logical operators (i.e., we do not exploit the Lispy ability to have variable arity operators, such as `(< 50 x 100)`). What might explain this discrepancy?

978-1-4673-8367-7/15/$31.00 ©2015 IEEE

When using a text-based language, this translation must happen at the textual level. When using a block-based language, this translation can happen at the moment that teachers write expressions on paper (text) or at the moment when they move from handwritten expressions into blocks. In our experience, the text-to-block transition was much more difficult for teachers than the text-to-text transition. This implies that a block language designed to be used alongside written materials may actually increase the need for rigorous textual syntax, rather than reducing it.

Of course, there is another possible approach. One could eliminate the restriction that these operators be binary, and allow a more direct translation. However, designing a block language that allows arbitrary arity may involve subtle user interface effects, which we have not thought through.

- The syntax must be well-specified and unambiguous in order for teachers to perform these translations accurately, and to establish a shared vocabulary for class- and group-discussion. This means confronting the very syntax issues that blocks are designed to gently avoid!

We conjecture that a similar need will be felt for writing code on the board in class, too. We assume this issue has been studied in the other literature on teaching with block-based languages.

In short, pedagogical techniques that are especially common in mathematics appear to require a formal, textual pseudocode—routing around the problem using blocks on the computer only shifts this burden to what happens on paper (and on the board).

- Properly modeling the abstraction provided by functions is not always a good idea! The Code.org language uses modal windows for definitions, which nicely reflects the function definition versus function use distinction. (Named constants—a.k.a., "variables"—are also defined modally.) In practice, this has proven to be a surprisingly bad idea. When writing one function should make use of another, teachers strongly resist the use of abstraction, because they are unable to see what is inside the callee block! This encapsulation trades visual space for working memory, and it is a much more expensive trade than we had anticipated. On the other hand, not having a modal mechanism means it is much harder for learners to track what it is they are doing on the screen at a given moment.
- The goal of our prototype—subsequently adopted by Code.org—was to explore the use of *types represented by colors*. Naturally, there are only so many color gradations that the eye can discern; our view was that once students advance to more sophisticated types, they should move beyond block-based programming anyway. Therefore, the language intentionally offered only a fixed set of types, represented by distinguishable colors.

  That said, some amount of type sophistication appears surprisingly early. Consider a conditional construct: what "color" is it? In fact, it's polymorphic: it takes on a type—and hence a color—based on the content of its branches. There appear to be three ways of handling this:

  - Our prototype gave such blocks a neutral color initially, and used Hindley-Milner inference to determine their type as the program evolved.
  - Code.org's version instead uses a neutral color and, eschewing an inference process, leaves the blocks in that color. This unfortunately grossly violates the precept of block languages, confusing the meaning of colors, thereby creating considerable confusion for learners. This is not to suggest that the inference-based solution is more *usable*; it is certainly more *accurate*, and its usability needs to be studied further.
  - A third possibility, which we are now exploring in our collaboration with Code.org, is to simply have each of these constructs be duplicated across the different colors. This simple expedient appears to gracefully address both of the above problems, but it creates its own issues: (a) the user must pre-select which colored conditional to use, and pay a high switching cost if they chose incorrectly; and (b) it suggests that there are N different notions of the conditional construct, which is at least unsavory (and arguably highly inaccurate) from a programming linguistic perspective.
- One of the strengths of a block-based language for students is that dragging a block may simply be faster than typing an expression. However, teacher workshops themselves rarely suffer from this constraint. We have found that for the teachers in our professional development, writing a program using blocks takes far more time than typing. As facilitators, our goal is often to spend most of our time discussing pedagogy and content, and expect the actual typing to be quite brief. The use of a block language interferes with this, requiring much longer workshops or less discussion of content and pedagogy.

Note that these problems largely disappear in most textual representations. Some of these are specific to professional development of teachers, but some are much more general. We therefore believe each of these represents a line of research necessary into the relative strengths—and designs—of textual and block languages.

Acknowledgements This work is partially supported by the US National Science Foundation, Code.org, Google, CSNYC, and TripAdvisor. We thank Rosanna Sobota and Emma Youndtsmith for their collaboration and insights.

MUzECS: Embedded Blocks for Exploring Computer Science

Matthew Bajzek, Heather Bort, Omokolade Hunpatin, Luke Mivshek, Tyler Much, Casey O'Hare, Dennis Brylow
Marquette University
firstname.lastname@marquette.edu

***Abstract*—We build on the success of the Exploring Computer Science (ECS) curriculum, which has improved the accessibility of a Computer Science education to students of all backgrounds. While ECS has been well-received by its students and successful in reaching many students of diverse backgrounds, it currently suffers from a final module which is expensive and offers no easy transition to text-based programming.**

This module, which teaches robotics as an application of computing, rests on the LEGO Mindstorms[1] platform. The Mindstorms have a block-based programming language that abstracts too much code to be of any use for making a transition from block-based programming to text-based programming. They are also more expensive than most schools can afford. This paper reports on one aspect of our successful effort to create an alternative curriculum which meets the same curriculum objectives as the current sixth module, but for a reduced price.

Keywords*—*Computer Science, ECS, block-based programming, paper, Arduino.

I. Introduction

Despite its critical and growing importance, Computer Science is taught in only a small minority of United States high schools.[1] The MUzECS project has designed the hardware, software, and curriculum for a new, low-cost module to replace the expensive existing final module for Exploring Computer Science[2] , an introductory high school curriculum. ECS is aimed at increasing the accessibility of Computer Science education for women and minorities in order to improve the diversity and student opportunity within computing courses.

The curriculum has proven effective in its goal, and have improved upon it by both lowering cost and creating a dialect of Ardublock[3] upon, thus making the ECS curriculum a better and more widely avalible learning tool. This dialect is what our entire module relies upon to be the interface students have with learning basic programming through our module.

A. Project Scope

In order to offer a cheaper alternative for low-income schools to teach Computer Science courses, we have created a new module based on an entirely new platform. Our design utilizes the Arduino Leonardo, a streamlined set of simple peripherals, a rewritten class curriculum, and a custom block-based programming language.

Students write code in our dialect of Ardublock: an open-source hobbyist language already available for enthusiast use and modification which sends its blocks as strings of code to the Arduino IDE to deploy it directly to the Arduino board. The board receives input and provides output using the set of peripherals included in the Arduino shield, a board containing peripherals that can be plugged directly into the top of the Arduino, we provide. The small embedded system created by this combination of parts will closely follow the content of the previous modules during which students learned to program using Scratch[5], a block-based programming language akin to Ardublock.

The curriculum we deliver has well-defined lesson plans, assignments, and projects which meet specific student learning needs. This curriculum has been piloted at half-a-dozen schools totaling seventy individual hardware units being tested and used with our software. The student data from this pilot study last spring are currently being analyzed to see if our curriculum meets the same objectives as the current module. The data that has been processed points towards a successful integration into the curriculum.

The curriculum is backed by a custom circuit board

[1] http://www.exploringcs.org/resources/cs-teaching-resources

Fig. 1. The shield and its peripherals attach to the Arduino GPIO headers

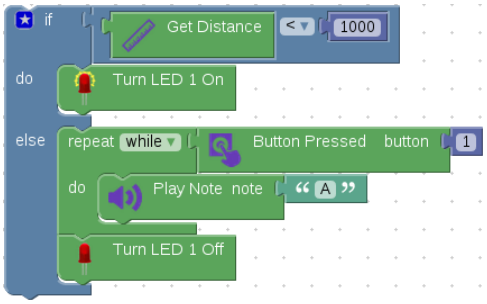


Fig. 2. Example use of control structures similar to high level programming

shield, as seen in figure 1, which mounts directly onto the General Purpose Input/Output (GPIO) headers of the Arduino Leonardo in order to power its peripherals, which the students will control via our final deliverable, the custom block-based language of MUzECS. This language, created by extension of Ardublock is to be used by students via a simple interface familiar to them from their Scratch training. Code will be deployed from this interface using a single button that prompts the Arduino software to compile to the board, ultimately executing the students' programs.

We support the natural progression to an adept understanding to text-based languages from block-based by supplying control structures as blocks and making all other blocks compatible with these. The control structures in use are if-block, if-else, if-else-if, repeat-until, repeat-while, and for-loop. A classroom could use these to create a variety of applications. For example, a block program where when an object is close it turns LED one on and when it is far away turns off and while button one is pressed a note plays, as seen in figure 2.

II. Previous Research

There are many widely used visual block based programming languages and dialects in computer science education. These visual programming languages serve the needs of high school teachers with varying degrees of success. We built MUzECS to fill some of the needs of a high school computer science curriculum that some visual block based programming languages don't completely satisfy.

For instance, Scratch, is a visual programming language that makes it easy to create your own interactive stories, animations, games, music, and art. In addition, Scratch is very tinkerable, social, and diverse in the projects one can make with it. This is ideal for high school and middle school students who are just beginning to learn how to program. Scratch is easy to use and makes block programming interesting for students. Unfortunately, we have found that Scratch doesn't originally work with any open hardware and doesn't facilitate the progression to text based programming languages.

Furthermore, ECS found a block-based robot set, which supplied a visual block based programming environment and interactive hardware, that was currently being deployed in many schools' computer science curricula called Lego Mindstorms NXT[7]. There is a programmable robot set, which comes with several types of sensors, motor, and programmable brick, named Mindstorms NXT(NXT) made by the Lego Company.

MUzECS is designed to cost a fraction of Lego Mindstorms, while still supporting equivalent curriculum goals. Our MUzECS set is sixty dollars compared to six-hundred dollars for the LEGO Mindstorms NXT 2.0 by the Lego Company on Lego.com[2]. According to Barry Fagins, who is the developer of Ada for Mindstorms and has used Mindstorms in courses[6] there is no proof that Lego Mindstorms improves learning or enhances retention in computer science education.

In making our visual programming dialect MUzECS usable on all operating systems that can run the Chrome browser, we built it off of Google's open source visual block based programming environment called Blockly [4].

A. MUzECS Ensemble

Natural progression is further facilitated by requiring the user to click the 'Arduino' tab showing the Arduino code before they can upload to be compiled. This ensures the user sees the Arduino code after they use the blocks

[2]http://shop.lego.com/en-US/LEGO-MINDSTORMS-EV3-31313?p=31313&track=checkprice

and can make connections between the conversion. This connection will hopefully ease the transition to a text-based language.

Within MUzECS there is also an 'Advanced (Pins)' section where more complicated blocks are available. We have chosen to implement this to give the user more freedom and thus creativity. The labeling 'Advanced' signifies a user with knowledge of the pin numbers on the Arduino which correspond to different hardware peripherals on the shield can use these blocks. This is also where users who would like to use the Arduino pins labeled 'advanced pins' on the shield can come to use their own peripherals with the Arduino. If a user would like to use MUzECS without the shield designed for it, this section allows for direct pin reference on the Arduino and can be used to set up other peripherals on the Arduino.

III. Blocks

Throughout the creation of the software, hardware, and curriculum in the MUzECS project the hardest obstacle to overcome in the whole project was creating the design for what blocks are available for student use and how they are presented in the MUzECS dialect of Ardublock. Although it is built to be usable by both beginners and those familiar with programming, the project is geared towards helping teach a ninth grade audience who are learning to program. The design decisions for how MUzECS blocks were all made with this goal in mind.

The block interface allows MUzECS to show users multiple versions of simple ideas with differing complexity while allowing more difficult programming to be abstracted behind simpler blocks. The entirety of the design attempts to make programming not only easy for a ninth grade beginner, but to provide them stepping stones towards the comprehension of more complicated ideas in programming and ease the transition from our block-based programming dialect to more standard text-based languages.

A. Overlapping Block Objective

In many high level languages there are many ways to accomplish a single objective which is why we implemented a similar ability in MUzECS. Some sets of blocks in the MUzECS dialect accomplish this in order to provide the tools for students to learn how the blocks behave in written code.

Fig. 3. Example setup of how to play musical notes with the different methods provided by MUzECS

For example, the blocks 'play note time' and the combination of 'play note' or 'play note frequency' with 'no tone,' and 'delay,' as shown in Figure 3, all hold the same function: playing a musical note. 'Play note' time is the easiest to use for a beginner and holds the most abstraction of the three choices as it automatically takes a standard musical note and converts it to that note's frequency then sets a 'delay' until the note should be terminated at which point it sets a 'no tone'.

As students become more familiar with how the 'play note time' block works and the code behind it they are able to move forward and understand the building blocks of 'play note time' as they can access all of its components in block form. 'Play note' is an intermediary step that not only allows simple use of notes without having to look up frequencies for each note, but can be used to understand the underlying code to 'play note time' without comprehension of what a frequency is and how that can be used to make musical notes.

The above layering of block operation offers user friendliness and intricate control in how a note is played. It allows a user to create a song from strict notes or for sounds via frequency to depend on a changing variable or to play a desired pitch between standard notes.

B. Block Abstraction

Using blocks to abstract more complex code can be used in ways other than layering their use with other blocks. This method of abstraction is helpful to hide code that is much too complex for a beginner to comprehend.

The 'get distance' block is an example of this abstraction. It may seem to be a simple data return as all it gives is the distance of an object, but it has complicated programming behind it. '[G]et distance' returns refined data that uses the time it takes an ultrasonic ping to bounce off of an object and return to the sensor. That data is then filtered and smoothed prior to being sent to the user's computer.

The block-based nature of the MUzECS dialect of Ardublock is perfect to put this code behind a simple

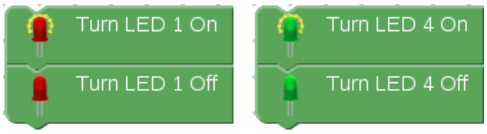


Fig. 5. Visualization queue to quickly assist beginners in understanding what their code will do

block in the interface. We have also put it into its own function within the text-based code. This means that a beginning programmer can just as easily look at the text-based code created by MUzECS and see their program with a simple function call to getDistance(). A user may then look at both the text-based and block-based code and see a direct one-to-one correspondence between their blocks and the Arduino code as shown in 4.

C. Visual Aids

We have found it is important to visualize the one-to-one relationship between code and output. It saves time and aids in a friendly user interface. To add this visualization, many blocks in MUzECS have associated images with them. An example of such is 'turn LED # on' with a bright colored LED image, 'turn LED # off,' with a dark LED image, as seen in Figure 5. Similarly, 'play note' has a speaker and sound waves, and 'set up keyboard' with a keyboard image.

These encourage the user to visualize what their code will do prior to compilation. When a user is able to imagine what their code will do without compiling it and seeing the output is first, they are able to make less mistakes and save time. This is also necessary for our user's natural progression to adept programmers.

D. Block Expansion

We bolster the transition to text-based programming languages from block-based ones through use of setting up resources for input and output devices. The keyboard runs via the three blocks: 'setup keyboard,' 'update keyboard,' and 'key pressed' all of which in conjunction allow a student to read input from a keyboard just as they would a button on the shield. However, unlike a button the method to poll for a keyboard key is much more complicated than polling for a button press.

This means that while 'button pressed' is a single block the keyboard uses three. This complication is due to the button being directly on the shield, making it fairly easy to interact with while the keyboard needs

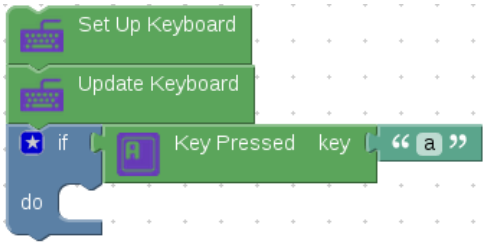


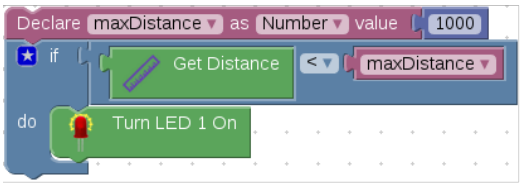
Fig. 6. Setup of I/O to assist in transition to text-based languages

to be found on the user's local system and interacted with via a queue. In order to use the keyboard the student must use the blocks in the following order: 'setup keyboard,' 'update keyboard', then 'key pressed,' as shown in figure 6. While all three of the blocks could be combined into one, keeping them separate allows the students using MUzECS to learn the basic differences between communicating with a button on the shield. The 'setup keyboard' block defines the variables being used by the other two blocks while 'update keyboard' sets up communication with the local computer to talk to the keyboard and checks the buffer for recently pressed keys and flags them. Finally, 'key pressed' checks for the flag of a specific key and returns true or false.

This separation allows a student to comprehend what steps are going into the process to communicate with the keyboard and its complexity compared to communicating with a button on the shield even if they cannot grasp the specific code elements of each block. This introduces the student to the idea of communicating with different I/O devices on the computer and helps take a step towards a better understanding how the Arduino works and takes the student closer to being able to use and interact with the 'Advanced (Pins)' or 'Advanced (Code)' sections of MUzECS.

E. Error Handling

Similar to standard IDEs for text based coding which attempt to point out as many possible errors prior to compilation, MUzECS uses a system of alerts. For example, the 'button pressed' block takes any integer as input for a button, but only an integer containing only a valid button number will get passed on to be compiled. When an invalid button number is passed and the code is attempted to be sent to the Arduino IDE for compilation, MUzECS alerts the student prior to the code being sent and the blocks are converted to code. The alert message in this case consists of "Invalid button used in block. Hint: Valid button numbers are 1-4."




```
void setup()
{
  maxDistance = 1000;

  Serial.begin(9600);

  pinMode(3, OUTPUT);

  pinMode(2, OUTPUT);

  pinMode(5, OUTPUT);

}


void loop()
{
  if (smoothDistance() < maxDistance) {
    digitalWrite(5, HIGH);

  }

}
```

Fig. 4. Example of the code mapping in both text-based and block-based of a sample program that uses a variable and lights an LED if an object is close to the shield

Additionally, 'play note' works in a similar fashion. This allows students to learn the basics of handling poor argument passing and other coding practices to be avoided prior to learning about the compilation process and how to interact with a more complex IDE in order to find the bugs in their code. Similar alerts are thrown whenever a user attempts to compile code that has errors in it such as attempting to use an undefined variable (Arduino uses on its own flavor of C) or improperly setting up the overarching loop structure embedded systems like Arduinos run on.

IV. Future Work

We will be collecting block usage statistics from high school students and teachers, who are already using MUzECS, through the use of surveys. We are eager to see if block usage statistics from MUzECS indicate a natural progression towards text based programming languages and stimulates a penchant for programming among our users. The kind of conclusions we hope to draw from this data collection is whether our users liked using MUzECS; whether they found using MUzECS difficult in any way; what bugs they encountered with MUzECS; and what features they want added to MUzECS. Our analysis will include a coding of student final projects with both a MUzECS experimental group, and a Mindstorms control group to assess whether students are achieving similar or improved outcomes with MUzECS.

Our next step is to develop more additions to the shield in order to foster more creativity among our users and expand curriculum. We have plans to add more sensors in addition to the distance sensor on the shield, such as a temperature sensor, a light sensor, a pressure sensor, and an accelerometer.

After speaking with a significant part of our user base, we have noticed that some schools are shifting to using Chromebooks in their computer science curriculum. This is understandable as Chromebooks are cheap compared to standard laptop prices, and fit most of high school computer science's needs. Unfortunately, Chromebooks can't install Arduino software. Due to this, we have developed a Chrome browser based web-page and web app to eliminate limited platforms. Anything that can run a Chrome browser can use our software and hardware in tandem. This is planned to be rolled-out to schools in Wisconsin this coming school year for feedback.

V. Conclusion

As a final point, our main goal was to design a low-cost alternative to the sixth module of the Exploring Computer Science curriculum that we piloted in schools during the Spring 2015 semester. While future work does exist and analysis of survey data will no doubt provide useful insight into the strengths and shortcomings of our designs, our team is confident that we have successfully produced hardware, software, curriculum, and resources which meet our customers needs and wants and fulfills our main objective.

As has been mentioned several times throughout this document, our projects ability to meet the subjective customer needs put forward during the initial design phase of this project are to be assessed in the future after the collection of teacher feedback and survey data post-piloting of our module. Once these data are collected,

there will be extensive work to be done in improving our designs to meet the feedback. In addition, further testing work will need to be performed as new features are implemented in order to guarantee the continued reliability of the MUzECS platform. Our team has already made plans with Dr. Brylow to continue these efforts into the fall and beyond to ensure the longevity of the project.

VI. Acknowledgment

Some of the students on the MUzECS team were supported by the National Science Foundation, grants CNS-1339392, and ACI 1461264. We would like to thank our pilot teachers and students. We owe special thanks to curriculum consultants Robert Juranitch and Gail Chapman. The pilot curriculum testers both teachers and students alike, software and hardware included the work of previous student researchers Alex Calfo, Farzeen Harunani, Jonathan Puccetti, and John Casey. The MUzECS project was built on top of the existing Ardublock system which in turn was built on the Blockly architecture.

References

[1] Code, https://code.org/promote

[2] ECS, www.exploringcs.org

[3] http://blog.ardublock.com

[4] Google Blockly, developers.google.com/blockly/

[5] Resnick, Mitchel, et al. "Scratch: programming for all." Communications of the ACM 52.11 (2009): 60-67.

[6] McNally, Myles, et al. "Do lego mindstorms robots have a future in CS education?." SIGCSE. Vol. 6. 2006.

[7] Kim, Seung Han, and Jae Wook Jeon. "Programming LEGO Mindstorms NXT with visual programming." Control, Automation and Systems, 2007. ICCAS'07. International Conference on. IEEE, 2007.

Middle School Experience with Visual Programming Environments

Barbara Walters and Vicki Jones
vanbara, Inc
Apex, NC USA
Email: {barbara.walters, vicki.jones}@vanbara.com

***Abstract*—Middle school students often avoid programming endeavors, thinking they are boring and non-productive. We have been successful engaging and maintaining interest in programming through the use of App Inventor 2 and a curriculum of our design. This paper details the features in App Inventor 2's visual programming environment that we believe are significant contributors for engaging students and facilitating their understanding of abstract concepts.**

Keywords—visual programming; middle school; teaching programming;

I. INTRODUCTION

As computing professionals who love programming, we have fully embraced visual programming environments to entice the next generation of programmers and technologists. As volunteers at local schools, we have developed curriculum and activities to teach programming using App Inventor 2. In this paper we discussour experience and major contributors to our success.

We have engaged in outreach activities to attract students to engineering for most of our professional careers. More recently we have tutored middle school students in math and science. We found their lack of exposure to programming troubling and decided to become more proactive.

In the 2014-2015 school year we started using App Inventor 2. We worked with nine groups of students, in year-long weekly after-school clubs, as enrichment in an existing class and as week long summer camps. There were over 150 students involved in these activities.

Teaching middle school students to program is wholly dependent on keeping them interested long enough to succeed. We have discovered that middle school students are interested and remain engaged in programming activities that produce an outcome they can produce with ease, creatively customize, and share with pride. Our program allows them to do that using the App Inventor 2 visual programming environment to develop customizable apps.

Teaching hundreds of middle school student to program is wholly dependent on enabling their teachers to succeed. App Inventor 2 is easy to use by anyone, regardless of their experience with technology. Our experience in North Carolina is that little is required in the way of infrastructure not already available in many school and community locations. App Inventor 2 and our curriculum provide ample opportunity for enriching the middle school curriculum.

II. VISUAL PROGRAMMING IS REQUIRED

Our goal is to teach middle school students how to program. These students are mature enough to follow multi-step instructions, understand math concepts such as variables and the coordinate plane, and can read and understand words such as "orientation." In order to engage students, we found a visual programming environment was a requirement. Many of our students have been previously exposed to programming activities that left them with a negative impression of programming. However, because we offer to teach students how to make apps, we have no difficulty attracting students to our program and overcoming their initial hesitation. Producing apps is the hook for engaging our students. The ease of the visual programming environment keeps them, and their teachers, coming back.

Many of our students have parent(s) who program professionally for companies like Cisco, SAS Institute, Red Hat, Epic Games and others. The parents, including the authors, love programming and tried to share the experience with their children. They used a variety of text based languages such as Python, Java and Javascript. In general, these attempts were unsuccessful because the students simply did not like using these languages. They disliked the programming environments, they were impatient with the syntax requirements and they struggled to create something they wanted to show off.

Searching for more engaging programming environments, we found Scratch. In the fall of 2013 we began using Scratch at several middle schools as part of a year-long enrichment program. Acquiring the necessary computer facilities and internet access were no problem so this made Scratch very attractive. Since we did not find many materials to use with the class, we created activities that are very similar to those included in Google's CS First.

Through that experience we discovered that one very important factor in motivating students is the ability to customize a program and make it their own. We had difficulty supporting those desires with Scratch. After students completed several Scratch activities and understood the basics

978-1-4673-8367-7/15/$31.00 ©2015 IEEE

of programming, we let them choose one of three types of projects: a program that demonstrated a science concept, told a story or related a set of facts on some topic. Initially the students were excited, but for many of them the enthusiasm faded. Rather than programming, we spent many hours searching and manipulating images for their projects.

It was our experience that the functionality provided by Scratch was not sufficient to engage our target age group and encourage them to continue programming.

III. Our Experience with App Inventor 2

After using Scratch, we looked for alternatives. We experimented with Snap! and Alice before finding App Inventor 2. App Inventor 2 provides a rich programming environment with a variety of components: traditional UI components (button and label), animation components (canvas and sprite), sensors (accelerometer and location), social media, storage, connectivity and so on. From the beginning, our students recognized that they were not restricted by the environment. Importantly, to them this was not merely a vehicle for teaching programming, but an environment where they could create fully functional apps.

The schools in North Carolina are well equipped with computers and wireless networks. Bring Your Own Device programs are becoming common. However, App Inventor produces Android apps and we found no schools had those devices; Apple devices were common. Before our program could be deployed, we helped teachers obtain devices through programs like Donors Choose and donations from corporations and individuals.

App Inventor provides an emulator. There were numerous issues with the emulator, from long start up time, lack of functionality (you can not shake an emulator) and frequent updates. We chose not to use the emulator and instead used the AI2 Companion app. This also requires frequent updates but was a better choice for our program.

From the very beginning of our program, students have the opportunity to customize their work. We were struck by how important customization has been to our students: we attribute much of our more recent success keeping students engaged to providing opportunities for them to exercise their creativity and make their product unique.

Initially our students were eager to make their own apps (not ours), but they have no idea what skills they need to be successful. Our approach is to have students create 5 apps of our design, then guide them in creating an app of their design. By requiring them to make our apps, we can build skills and introduce new concepts gradually. Because App Inventor 2 provides such a rich set of components, it was easy for us to create apps that have a variety of functionality and afforded the opportunity to gradually introduce computer science and math concepts. The depth and richness of App Inventor 2 is very beneficial in keeping our student's interest as well as convincing them that this is a "real" programming environment.

Each of our five apps teaches new concepts and introduces new components. They include several games, apps that perform text manipulation used in Mad Libs or Hang Man and utility apps that use API's of other apps. The apps access data that may be stored in the app, on the device or on the cloud. Some apps use social media. In addition, each app allows the student to exercise their creativity by customizing the app. We had very few students that were satisfied by merely producing our app. Initially, due to having minimal skills, the students modified the appearance of the app. This led to lively conversations about intellectual property rights and copyrights.

As they gained skills they began to change the behavior of the apps as well. They added or modified blocks to change the logic of a program. They added components such as social media and storage components. They also started thinking in more concrete terms about the apps they would like to make. They started thinking about what was available in App Inventor 2 and how they wanted their apps to look and behave.

We attribute our success using App Inventor 2 to engage our students to several factors such as creating something concrete and shareable and facilitating creativity. Another contributor to this success is the App Inventor 2 user interface. For the students it is important that it is easy to use. As educators, we appreciate the support it provides in reinforcing computer science concepts. We volunteered with several teachers who teach programming in high school and they were surprised and pleased at how quickly students grasped the basics of computer science using App Inventor 2. The teachers commented on how difficult it was to teach some of these concepts using their traditional methodology and with this environment, much younger students understood and applied important computer science concepts.

IV. Usability and App Inventor 2

In our first app, we explain fairly complex concepts regarding user interface, components, hierarchies, property sheets, event driven programming and methods. Our students have no problem grasping these concepts because App Inventor's user interface supports these concepts.

In the design view, students are introduced to the concepts of user interface design, components and properties. Once a component is added to the project, it can be selected and its property sheet made visible. At a glance, the students see what properties they can control. In follow-on apps, we discuss application life-cycle and initial conditions. The property sheets are ideal for this discussion.

In the design view they are also introduced to hierarchies based on the way components are organized. A screen is a parent to a canvas and a canvas is a parent to a sprite. Setting the height of a component to the value "Fill Parent" makes this relationship easy to understand.

When they switch to the blocks view and select a component, students easily identify the getters and setters because the names (usually) match the names in the property sheet. Getters and setters are similar colors and appear distinct from other methods.

The blocks view makes it is easy to understand the behavior of a component. When selected, each component displays the events it can detect as well as its methods. Together, the properties, methods and event handlers define the behavior of the component.

Because methods are organized with a component, it is easy to explain that methods are named programs associated with a type of component. Later, when students create procedures, they are already familiar with the concept of a named program. We explain methods are named programs associated with components and procedures are named programs that are part of an application.

Event-based programming is quite natural as well. We start by having students create apps that respond to user events: a finger dragged across the screen, shaking the device, pushing a button. Because they understand user initiated events, it becomes easier to understand other types of events that were not caused by a user action or a sensor. Discussing application life-cycle in terms of events (Create, Start, Resume, Pause, Stop, Destroy) is easily understood, even though all of these events are not surfaced in App Inventor 2.

The operators in App Inventor 2 make it very easy to explain data types. Unlike programming environments where data type is expressed by a shape, in App Inventor 2 data types are color coded and match the color of their operators. In the blocks view, creating a value of a particular type and the operations performed on that type are grouped together, subtly establishing important relationships in the mind of the novice programmer. Many operators will not allow a connection to a data block if it is not the correct type. This re-enforces the concept of data type and provides a natural opportunity to discuss computer science concepts regarding data. Some operators allow connections to any data type. This provides an opportunity to discuss data conversion. Many data types can be converted from one type to another, such as when a number is converted to a string so it can be displayed. This is why the text operators accept most data types. But the conversion is hidden which can be a source of confusion if not explained to the student.

Most property "getter" and "setter" methods require a specific data type, for example true or false for the property "visible". These methods do not require a data block of the required type. We have encountered a few examples of students connecting an incorrect type, but this has been relatively rare.

In addition to being engaging for middle school students, the richness of supported activities in App Inventor 2 allows additional educational advantages. For example, having colors as a data type and blocks to create new colors leads naturally to discussions on binary and hexadecimal representation. Throughout our curriculum we take advantage of other opportunities for enhancing traditional middle school subjects.

Overall we are delighted with App Inventor 2 and are reluctant to criticize because, as an open source project, we could contribute code and we have not prioritized the time to do so. The following are ideas for improvement. There is only a single area for programming blocks and organizing blocks for anything other than a trivial program is difficult. There is no way to create a library of reusable components. Image Sprites have only a single property for heading which is used both for rotating an image as well as setting the direction for movement. Independent control of these is often desirable. The most common mistake our students make is in typing the name of a media file. If there were blocks that contained the list of media files, this type of error would occur infrequently.

We personally invested hundreds of hours learning how to best use App Inventor 2 to meet our goal of teaching fairly young students how to program. There are lots of resources available on-line that are now used by our students after they completed our program. What we did not find and had to create ourselves, was an age-appropriate introduction to programming that included the computer science concepts we believe are essential.

V. App Inventor 2 Makes Math Real and Relevant

App Inventor 2 makes it easy to demonstrate math concepts in a very concrete way. Many Common Core Math concepts are reinforced in our activities. Teachers are excited that the students see the relevance of these concepts as they build apps.

For apps that use the Canvas component with either image sprites or displaying text, the student becomes accustomed to using the graphics coordinate system to position objects. They have to resize images and scale them so they maintain the correct appearance, using the ratio of the screen size and image size.

Transformations such as rotation, translation, dilation and reflection of image sprites is used in one of the gaming apps and we introduce and define those terms. Variables are used to maintain state and in calculations.

Time units are used for moving image sprites as well as clock events. Order of magnitude is used to describe human time (seconds) compared to the time units used in App Inventor 2 (milliseconds).

Boolean values and operators are used regularly in control expressions and are used to implement the rules of a game.

Probability and random numbers are used in one of the game apps and are common in student-designed apps.

Bits, bytes, binary and hexadecimal representation are easily explained through the construction of colors.

These are just a few of the math concepts used in our activities. Our students are familiar with some of the concepts, such as the coordinate system, but some are not introduced until 8th grade or later. Our students are excited to use their math skills and enjoy understanding how to use them in making apps.

VI. Student Ownership

For each of our apps, the student builds the entire app: the user interface, the program logic and finally customization. In our initial pilot we miscalculated the complexity of one of the apps and it took our students nearly eight hours to complete it. When we realized how long it would take, we offered a

partially completed project to shorten the time to completion. Our students rejected the offer, preferring to complete the app on their own, at their own pace.

In another setting, we worked with teachers who taught programming electives and asked if they had suggestions for improving our program. They suggested that we create App Inventor 2 projects that are partially completed and have the students add only the parts that are related to the concepts being taught. This reduces the amount of time the students spends on a particular app.

Through post-project surveys we found students did not like the partial project approach. In all cases where we offered a partially completed app, the students responded with much lower ratings compared to projects built entirely by the student. After a few attempts with partial projects, we discontinued this approach.

We do not know why partial projects are not as well accepted and think this is an area that may merit further study. One hypothesis is that the process of producing a correct, well-behaved and attractive program is similar to the process of creating a well-constructed piece of literature. Both engage a student's creativity, both have a set of well-established best practices and require skills attained through practice in order to produce a high-quality end-product.

The act of writing is an iterative process, just like the programming process. Rarely are students asked to contribute just a portion of written work to an existing document. Instead, they learn to write by making many attempts at creating documents with appropriate beginning, middle and end, refining their skills and their expertise over time. The premise is similar. Partial programming is equally unsatisfying as editing or completing literary works.

VII. Measures of Student Success

While we provide pre- and post-course assessments as well as quizzes for each project, these have been used only in classroom settings where our content is used for enrichment, not in the summer camp or school club setting. Our primary measure of success is whether, after creating our 5 apps, a student has learned enough that they can design and implement an app by themselves.

Each of our apps starts with a specification describing the appearance and behavior of the app. This is followed by a description of the App Inventor 2 components that are used. For this last project, the student is given only a description of the behavior of the app. In most cases we have asked them to make a magic eight-ball app. Many of our students are not familiar with the magic eight-ball, so we demonstrate how it works and give them time to play with it and discuss it.

The students then create their design which includes both the user interface and the behavior of the app. They choose the components that support the desired behavior, create the associated programs and complete the app.

This project gives them the opportunity to explore and use components that have not been used in our apps. Our students are able to complete an app of their design and we have been delighted at the variety of implementation techniques and the creativity they demonstrated. They proved they had learned a great deal and that they could expand their own knowledge independently.

We expect our second year students to continue making a few apps following our instruction in order to introduce additional concepts and teach more advanced techniques. In addition, we look forward to them exploring their own ideas for apps, making trade-offs in functionality versus complexity and increasing their expertise.

VIII. Next Steps

Our activities have been used in for a one-week summer camp, as enrichment included in another course or for a year-long after school club. For the second year, our students will continue to use App Inventor 2 to create apps of our design and implementation. In addition, the experienced students will mentor novice students as well as creating apps of their own design. We will introduce robotic programming as well as data collection through sensors.

But after the second year, we believe it will be appropriate to introduce more traditional languages and development environments. We are also concerned that at this stage, expertise within the classroom will be essential and there are not very many teachers knowledgeable in this area.

One language we are considering is Python. There are a wide variety of frameworks that work with Python, from gaming to data science. Whatever language is chosen as a follow-on, it must provide a functionally rich environment and the end product has to be as exciting and fulfilling as the apps produced by App Inventor 2.

IX. Acceptance in Schools and Teacher Engagement

Our activities were offered in a variety of environments: after-school club, in-class enrichment, summer camp. We believe this approach will also be successful as an elective course but would require acceptance at the district or state level and additional educational material. Instead, we plan to continue to offer the activities as enrichment or in informal settings.

Most schools have the infrastructure to support App Inventor 2: appropriate computer systems and browsers; and most schools allowed installation of the App Inventor emulator. Most of the schools provide Google accounts for their students. None of the schools had Android devices available; all had IPads. The private schools purchased Android devices so they could use our activities. We donated a set of Android tablets to one school and several others used Donor's Choose or 21st Century grants to pay for the devices.

A professional teacher was always involved, either in leading the activity or in partnership with us; we worked with 10 different teachers. They were certified in a variety of fields: elementary education, gifted education, math, science, Project

Lead the Way middle school programs, media specialists and computer programming specialists. It surprised us how reluctant some teachers were to learning the material. While all of them were supportive and very pleased with the how well the students were doing, only half of them tried the activities themselves.

In the classrooms where we were not involved and the teacher did not master the material, the students were still successful. Our activities are fairly prescriptive, with test and debug steps incorporated as the app is developed. This limits the scope of problems and we have encouraged the students to ask each other for help. That strategy has worked so well, that in the classrooms where we are present, we are rarely involved while the students are actively programming. We found ourselves much more in the role of coach and as a facilitator during discussions.

We believe in order to offer the opportunity to learn programming to a wider group, engaging teachers is essential. Teacher mastery of the content is not necessary for student success, but teacher buy-in is essential. The activities will not be offered unless there is a teacher willing to sponsor the activity. Anecdotally, teachers who plan to teach the material at a summer camp, in a community center or in a fee-based after school program were interested in learning the material themselves. We believe this is because they will be reimbursed for their efforts.

To encourage teachers to learn the material, we have engaged several of them to teach at community centers. We are also setting up an online course and community for teachers. The course will be offered during specific dates, not as a do-it-yourself program, with assignments and assessments. When a teacher completes the program, we provide a certificate of completion and will support them in teaching the activities in informal settings. Our hope is that once these teachers have mastered the material, these teachers will use the activities in the classroom as well.

There will continue to be an issue with the lack of Android devices, but we are also providing guidance regarding grants that will help purchase the devices and we have been astounded by the generosity of people donating via Donors Choose, especially people who are outside the school district and seem to donate based on the prospect of encouraging students to code.

X. Conclusions

In the 2014-2015 school year our activities were used in five schools and in two summer camps. Over 150 students, mostly fifth through seventh graders and about a dozen high school students participated. For 2015-2016 we are expanding where we teach and will offer the program in local community centers. Based on recognized need, we also plan to develop on-line instruction and a community for teachers.

Why are our students asking for more? Encouraging their friends to participate? We believe it is because they have made something unique, that they can show off with early and continued success made possible by App Inventor 2. We hope to share the enthusiasm by reaching more teachers, enticing them through a curriculum with obvious value for enhancing the curriculum and an environment that is not intimidating.

We are interested in pursuing research to vet the hypotheses presented here: Are students more interested in programming when creating apps, not just applications? Do students value creating their own product in its entirety and customized to their liking? Do students better understand the math concepts they have used in programming? How do we assist teachers in leading programming/computer science discussions?

Teaching and Learning through Creating Games in ScratchJr

Who needs variables anyway!

Aye Thuzar
Computer Science & Mathematics
The Pingry School
Basking Ridge, NJ, United States
athuzar@pingry.org

Aung Nay
Zatna LLC
Martinsville, NJ, United States
anay@zatna.com

***Abstract*— This paper presents an idea for teaching and learning through creating games in ScratchJr, which is an extension of the existing curriculum available for ScratchJr. It discusses our experience of running half-day camps with K-2nd graders using ScratchJr and how essential skills such as designing and planning, sequencing, problem solving, thinking about thinking (metacognition), and sharing can be developed through creating games in ScratchJr. Since ScratchJr does not have variables but rather events and messaging, we used a sprite (an object that performs actions) moving towards a goal to represent positive integers and away from that goal to represent negative integers. It is the same concept as using a number line. The sprite is at zero at the start of the game, and as the game progresses, the sprite moves either to the left or right goal triggering the winning or losing page. This way of scoring tracking is highly visual, and younger students can understand and incorporate it into their games even without understanding addition, subtraction, or negative integers properly. With this idea of tracking progress, we can create a variety of games that are very entertaining to play while still being understandable to younger students who will be able to code on their own.**

Keywords— ScratchJr; K-2; Games; No Variables; Progress Bar; Number Line

I. Introduction

We have been teaching Computer Science summer camps since 2009 and for the summer of 2015, we wanted to target the younger K-2 age group using ScratchJr in our half-day week-long camps. After reviewing the ScratchJr curricula that were available on scratchjr.org, we came to the conclusion that we would not be able to maintain the students' excitement just using the storytelling and animation elements for a week. We needed to offer something more. The natural inclination was to include games. However, creating games in ScratchJr was a challenge since ScratchJr does not have variables. Because of this, we developed the idea of a sprite moving towards a goal as an ongoing visual tracker/indicator of the player's progress, which allowed us to create a variety of games. The game creation allowed us to move beyond the interactive storytelling phase and maintain excitement throughout the week.

II. Background

ScratchJr is a graphical programming language that allows children from age five to seven to create "interactive and animated scenes and stories" [1]. It addresses the lack of programming tools that focus on "content creating or higher level thinking" [1] for kindergarten to second grade students. ScratchJr software deployment comes with the curriculum and online community, and the design goal of the ScratchJr software is to "provide young children with a powerful new educational tool as well as guidance for teachers and parents to implement it to the benefit of diverse areas of early learning, from math and literacy to interdisciplinary knowledge structures" [1].

III. Curriculum Deployment

Our ScratchJr camp curriculum is designed in such a way that a concept is revisited throughout the duration of the camp with additional concepts being added to the sum of knowledge as the camp progresses. Our planned learning experience starts with animated scenes and stories, using motion, looks, and sound blocks, and progresses to interactive animations, followed by game creation, using all block categories. Approximately 60% of the camp time was spent on creating games. We used iPad Mini2s with ScratchJr in our camps. Each student and instructor had an iPad. The instructor to student ratio was kept to 1:5 or lower. The camp day consists of modeling, project planning and creation, and social interaction (feedback and assessment). The camp week culminates in a final project, which is also an assessment.

A. Modeling

At the beginning of each day for each project, students see a model or example of what they are supposed to achieve for the day as well as for the individual phases that help lead the students to that day's overall outcome(s). For example, prior to a project, we might talk briefly about which concept(s) they need to learn and we might show them a sample project or two of what could be achieved with this new concept(s), along with what they already know. To introduce new concepts or to review concepts, we might use unplugged activities, such as coming up with a sequence of commands for someone to

978-1-4673-8367-7/15/$31.00 ©2015 IEEE

perform a particular task in real life, acting out a given sequence of commands or playing a card game with ScratchJr block images.

Fig. 1. Progress bar as a score tracker. (*Wizard & Tac*)

Fig 1 shows a sample project we modeled on the second day of camp. The game has the progress dot that starts at the middle and has the red vertical bars on the left and right sides of the screen. As time progresses, the red dot moves to the left periodically; however, if the fireball hits Tac, the little creature, the red dot will move to the right. Winning or losing is defined by whether the red dot reaches either the right or the left red bar. Here, we introduced the idea of the number lines, as well as positive and negative integers with this visual progress bar.

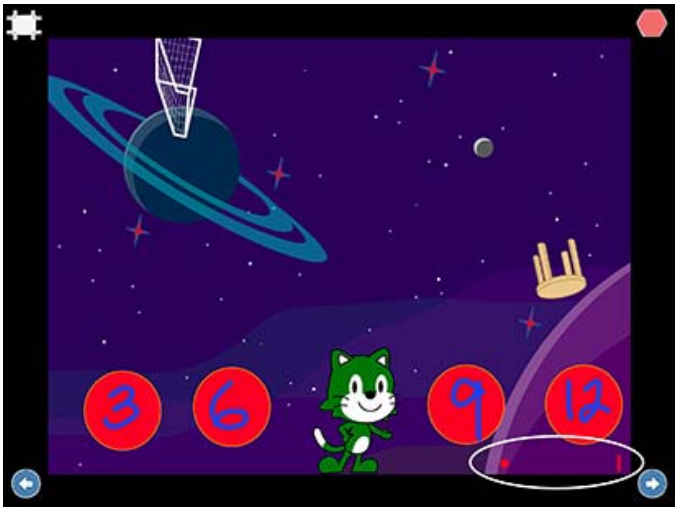


Fig. 2. Example of a progression tracker. (*Space Jump*)

Fig 2 shows a sample project we demonstrated on the fourth day of camp. Progression through the game is tracked by the red dot sprite moving towards the vertical red bar at the bottom right part of the screen that is highlighted by the white oval. As the game progresses, the red dot moves towards the red bar. When the red dot reaches the red bar, the player is greeted by a "You Win" scene.

Along with the achievement tracking, this game also tells a story about the dangers of space junk, and collecting energy stars. The storytelling aspect of the games always draws students in, motivates, and inspires them to create their own games with storyline. The integers that you see on the screen are buttons that allow the Martian Scratch Cat to jump different heights to collect the stars.

Our curriculum for ScratchJr also features the concept of "learn to code and code to learn" [4]. For example, there is always a math lesson to teach or review with every game we introduce to the students, either about numbers, total time frame, or the number line. In the game above, we had conversations about space junk and how to be responsible in space as well as why and how stereotypical Martians are green.

B. Project Planning & Creation

We make planning a regular part of the students' learning process. Before every project, students plan out different elements of it. They are required to draw out their plans and describe their plan to the instructors either in writing or orally. Below is an example of a student's final project planning sheets. This activity encourages them to think about their own thinking process(es).

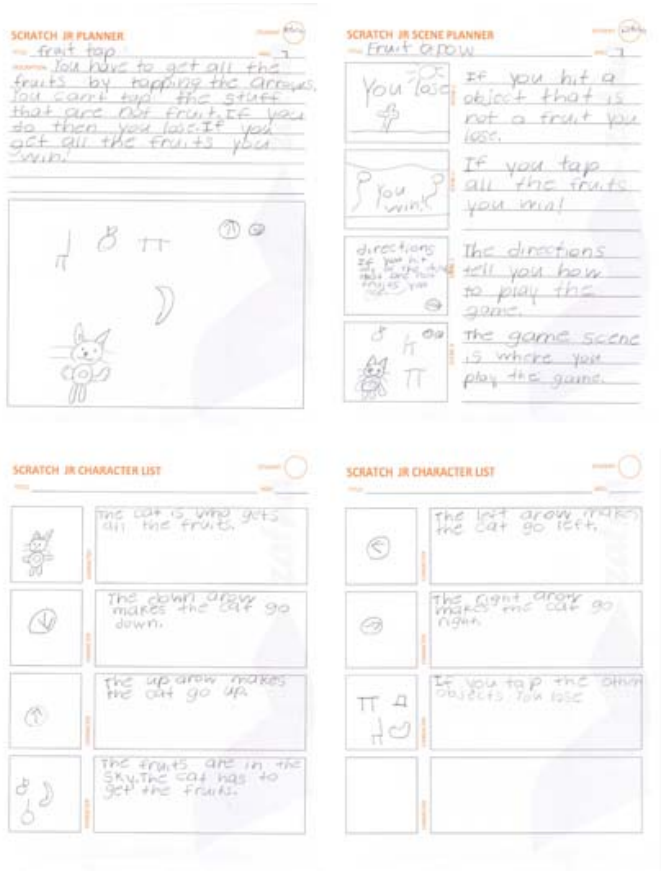


Fig. 3. Student planning sample. (*Fruit Tap*)

The students come up with a general concept, followed by a breakdown of that concept into different scenes, a list of characters and their actions. This planning process asks the students to think about the instructions/commands they need to

give to each of the characters in the game and how those characters will come together to form one cohesive story or game.

One of the biggest tasks in the planning process is the discussion that takes place between the individual student and the instructor with regards to the vision that the student has. This help students clarify their thinking processes. With the approval of an instructor, students can move on to iPads to make their plans come to life on screen.

C. Social Interaction

Our intent for social interaction among our students is three-fold: interaction at camp, interaction online and interaction at home. Interaction at camp is done through "code and tell" [2]. And the way we assess the "tell" part uses the three "levels of describing, demonstrating and imagining projects" [2]: simple, intermediate and complex. This YouTube link is an example of a student describing her game, including the story and her planning process: https://youtu.be/EFbo76jE_sk. When students complete their projects, they should spend a few minutes sharing their unique projects and stories with their campmates and instructors. The students should also give constructive criticism of each other's projects. This promotes cross-pollination of ideas between projects. Online interaction takes place at the end of the day when students vlog (video blog) which is intended as a conversation starter for friends and/or family members. Interaction at home occurs in a few ways: explaining the details of what the students did for the day to their parents (since parents already have prompts from the vlogs); teaching other siblings the great new things that students have learned at camp; and showing-off/bragging about one's achievements to friends who do not attend camp.

D. Final Project

Fridays are open project days. Students are given certain criteria such as creating a game with a storyline, having sound, and using at least four sprites etc. After the planning and creation phase for the final project, their projects are then shared with their peers and instructors at the end of the day. The final projects are assessed based on students' plan for the final projects. The instructors evaluate whether a student was able to create the project that s/he has envisioned and planned.

IV. Reflection

Having a relatively smaller instruction set and needing only a few blocks to create highly entertaining games worked out well with younger students. Individuals who had experiences in other drag and drop environments wished there were variables; however, we believed that we could do without variables, especially for younger students. In our opinion, ScratchJr is a very flexible tool with relatively "low floors" [3], low barrier to entry, and "wide walls" [3]. It supports a broad range of activities, including games. We can teach younger students how to express themselves digitally and be excited about their creations, a task not easily achievable prior to ScratchJr.

Additional benefit of creating games in ScratchJr is giving students "a new context for learning" [4]. Depending on their age, some were able to revisit and apply mathematical concepts such as positive and negative integers that they had already learned. For some younger students, it became a great way to get introduced to those concepts.

Some students feel restricted by the limitation on number of messages that could be sent and number of scenes they could use and wished there were more messages and scenes. We also experienced increased delays in our games with the increased number of sprites, regardless of the iPad model we were using.

Outside of the daily and sustained camp experience, the ephemeral use of ScratchJr might preclude the students' memory of integral aspects of the program which might require the teacher to re-expose the students to components that they have already seen. In the camp setting, this does not apply as often due to daily exposure to and use of the program.

V. Conclusion

We found that creating games in ScratchJr is a great next step for campers after they made their interactive and animated scenes and stories. Creating games with storylines ignited and maintained the interest of K-2 students throughout our half-day week-long camp. Our progress bar idea opened up opportunities to create different types of games that kept students' interest piqued. We were able to keep students excited throughout the week with more variety than what was possible with the ScratchJr cards and existing ScratchJr curricula at scratchjr.org. In addition, students were also exposed to a new context for learning other topics. Most importantly, we were able to create an environment where younger students could explore and learn about designing and planning, sequencing, problem solving, thinking about thinking, and sharing.

References

[1] Flannery, Louise P., Elizabeth R. Kazakoff, Paula Bontá, Brian Silverman, Marina Umaschi Bers, and Mitchel Resnick. "Designing ScratchJr: Support for Early Childhood Learning through Computer Programming." DevTech Research Group. Proc. of 12th International Conference on Interaction Design and Children, ACM, New York, NY. Tufts University, n.d. Web. 29 Aug. 2015

[2] Portelance, Dylan J. "Code and Tell: An Exploration of Peer Interviews and Computational Thinking With ScratchJr in the Early Childhood Classroom." Thesis. Medford/Somerville, MA/Tufts University, 2014. Code and Tell: An Exploration of Peer Interviews and Computational Thinking With ScratchJr in the Early Childhood Classroom. Tufts University, 2015. Web. 26 May 2015.

[3] Resnick, Mitchel et al. "Scratch: Programming for All." Communications of the ACM 52.11 (2009): 60-67. MIT Media Lab. Massachusetts Institute of Technology. Web. 21 July 2015.

[4] Resnick, Mitchel. "Learn to Code, Code to Learn (EdSurge News)." EdSurge. EdSurge, 08 May 2013. Web. 26 May 2015.

[5] Strawhacker, Amanda, Melissa Lee, Claire Caine, and Marina Bers. "ScratchJr Demo: A Coding Language for Kindergarten." DevTech Research Group. Proc. of 14th International Conference on Interaction Design and Children, ACM, Boston, MA. Tufts University, n.d. Web. 29 Aug. 2015.

Author Index