Empowering Children To Rapidly Author Games and Animations Without Writing Code

Rahul Banerjee[1], Jason Yip[2], Kung Jin Lee[2], and Zoran Popović[1]

[1]Center for Game Science
Computer Science & Engineering
University of Washington
{banerjee,zoran}@cs.washington.edu

[2]Information School
University of Washington
{jcyip,kjl26}@uw.edu

ABSTRACT

Prior research has established that long-term interests in programming are often shaped by formative computing experiences, especially those involving programming and graphics. Existing authoring environments for children (ages 9-14) to make 2D games and animations require them to: (a) create programs, (b) customize templates, or (c) combine rewrite rules with programs. One way to support early experiences in computing for a more diverse set of learners is to simplify such authoring systems, by removing text heavy code and minimizing cognitive load, which can allow separation of coding concepts from writing code. In this paper, we describe an exploratory system we are designing to test this idea, called BlockStudio. Using a Programming By Example paradigm, children manipulate colored blocks on the screen to specify desired behavior via concrete changes. Based on these inputs, our system synthesizes generalized rules based on color. We give a brief overview of our current prototype, then share insights gleaned from two intergenerational co-design sessions with children and discuss implications for designers of similar systems.

Author Keywords

Authoring; games; interactive content; programming by example; rule-based system; user interface;

ACM Classification Keywords

H.5.2 Information Interfaces and Presentation: User Interfaces

INTRODUCTION

Children's initial experiences with computers have the potential to shape their motivation to engage in programming. According to Hasan [12], early successes with programming and graphics are important for computer self-efficacy beliefs, while Bandura et al. [1] found children's perceived self-efficacy to be a key factor in their careers and future aspirations. Surveying informatics students at the college level, Ko [19] found that negative experiences with programming at an early age caused many participants to decide against computer science jobs as a career.

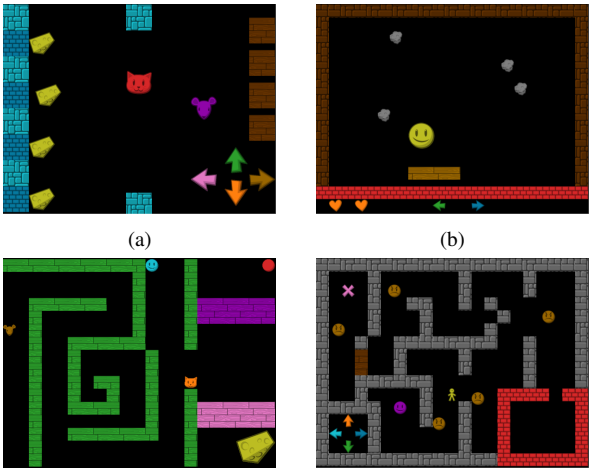


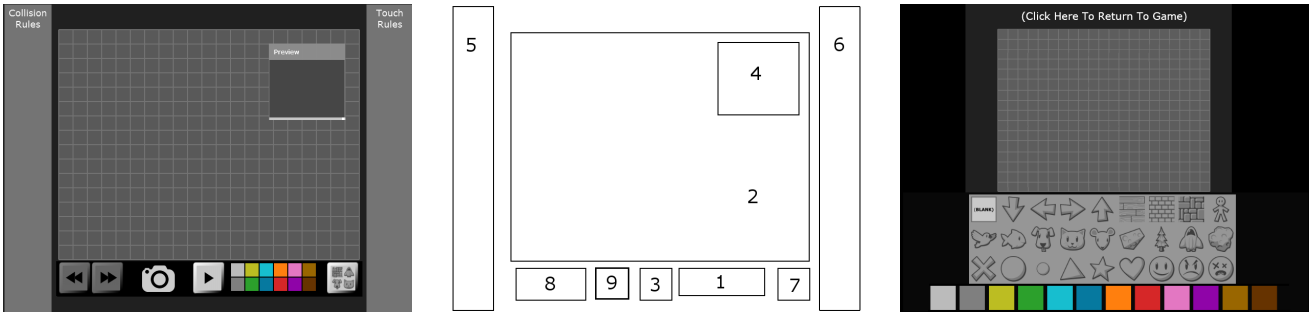
Figure 1: Games created in BlockStudio (a) Mouse Wants Food, (b) Extreme Pong, (c) Maze, (d) Grumpy Neighbors

Given children's love of video games and animations, a natural introduction to programming might be allowing students to program such artifacts themselves. Today, children ages 9-14 can choose from several authoring environments to create their own 2D games and animations ([36, 7, 33, 4, 42, 8, 14, 24, 37], also see [17, 11] for systems prior to 2002), but all of these options rely on programming (in the form of text or code blocks), on template customization via menus, or on rewrite rules combined with programming. Children have diverse backgrounds and motivations, and they may not be ready to take on one of the above systems. In particular, if they struggle too much with complex text or coding, such systems can be difficult to use.

Hence, it is important to explore systems employing alternative approaches that can help such children gain early success with computer programming. If successful, this could help them progress to more complex systems as the next step, whereas the alternative is for such individuals to decide that programming is not for them, which hurts the field of computing. Turkle and Papert [44, p. 4] observed "On an individual level, talent is wasted, self image eroded. On the social level, the computer culture is narrowed". A key part of programming is figuring out the simplest rules such that the system

IDC'16, June 21–24, 2016, Manchester, United Kingdom.
ACM 978-1-4503-4313-8/16/06…$15.00

DOI: http://dx.doi.org/10.1145/2930674.2930688



(a) (b) (c)

Figure 2: The BlockStudio Interface: (a) Main Screen, (b) UI Elements, and (c) Sprite Library. Names of UI Elements in (b): 1-Palette, 2-Grid, 3-Mode Button, 4-Preview Window, 5,6-Rule Panels, 7-Sprite Button, 8-Undo/Redo, and 9-Snapshot Button.

behaves the right way globally. This has traditionally been bound up with the ability to write code, but could be isolated and given to students separately (and earlier).

This paper reports on an exploratory prototype called *BlockStudio*, which is an authoring environment aimed at empowering children to create their own games and animations, while using no code blocks, very little text, and no knowledge of programming constructs like loops or branches.

We now describe related systems, followed by a brief overview of the current prototype. We then share formative assessments from two participatory sessions with an intergenerational co-design group, showing the potential of our approach. We conclude with a discussion of what we have learned from these early sessions, our future plans and implications for designers of similar systems.

RELATED WORK

Several authoring environments exist that allow children to create 2D games and animations. While more nuanced taxonomies of such systems exist (for an excellent survey of end-user programming systems, see [17] and [11]), from our perspective, previous systems use one of three basic approaches, mainly: (a) writing programs, (b) modifying templates, or (c) combining rewrite rules with programming. We now describe each of these categories.

Writing programs

One of the earliest environments explicitly aimed at teaching programming was Alice [33], where users coded in a scripting language to build 3D worlds. HANDS [32] combined a personified computing agent with a simplified programming language. Systems like Scratch [36] and a later version of Alice [4] used drag-and-drop assembly of code blocks to create games, animations, and stories, thus sparing novices the frustration of syntax errors. Storytelling Alice [18] was designed to motivate more girls to learn programming by leveraging 3D animated stories. Recent systems employing code blocks include Kodu [24], Blockly [9], PencilCode [10], and AppInventor [48].

From a Cognitive Load Theory [43] perspective, computer programming has a high intrinsic cognitive load, meaning that the task itself has a certain complexity which cannot be reduced. The program specification method (by writing text or by assembling code blocks) adds additional cognitive load, and we can attempt to minimize this. Mason and Cooper [25] found that using a subset of the normally available code blocks when programming Lego Mindstorms robots led to better learning by novice programmers, improved self-efficacy, and a lowered perception of difficulty. When designing ScratchJr [7] (aimed at children ages 4-6) its creators decided to use one third fewer programming blocks than Scratch. Thus, prior work shows that reducing code blocks can make such systems more approachable. With BlockStudio, we are exploring the idea of removing code blocks altogether, which differentiates our approach from these systems.

We are also minimizing the use of text in the interface, which sets our system apart from prior work. Wilson's report [46] describes interviews with dyslexic computer programmers, revealing their challenges when dealing with text. Although that work dealt with adults, we believe that reducing our dependence on text will make the interface more approachable for individuals of all ages who struggle with text.

Howland et al. [13] recommend system designers to present a task at the appropriate level of abstraction for drawing out the higher-level concepts, without necessarily connecting it back to programming. BlockStudio allows construction of branches and loops using only colored blocks, providing the possibility of expressing programming concepts, while bypassing the requirement to have textual code, or code blocks labeled "if then", "forever", "repeat", "repeat until", etc.

Modifying templates

Some systems support game creation via templates (e.g., side-scroller, top-down maze, etc.). After instantiating a template, its appearance can be modified using drag-and-drop assembly of tiles or characters, while other properties like motion parameters may be modified by editing appropriate property sheets. Systems like Klik & Play [23], GameMaker [42] and GameSalad [14] offer multiple templates, trading off the absence of coding with the memorization of a more text-heavy UI. The *split-attention effect* [3] predicts that when users have

to integrate information from disparate locations, they experience a higher cognitive load. We believe that time spent learning such an interface increases the time required for a user to build their first artifact, hence we chose not to use multiple templates.

LittleBigPlanet [41], Gamestar Mechanic [37] and Super Mario Maker [30] restrict users to a single template, leveraging learners' prior knowledge of specific games in their respective environments. BlockStudio has a "lowered floor" [36], but we also wish to have sufficiently "wide walls", hence we decided not to design the environment around a single template. We wish to enable children to create games, animations, stories, quizzes, and other digital artifacts that may not neatly fit into a single category.

Combining rewrite rules with programming

An alternate technique called Programming By Example (or Programming By Demonstration) allows users to demonstrate examples in order to generate code. One flavor of PBE uses rewrite rules, where users select a rectangular region of the screen and specify before-after appearances for that portion. Systems like ChemTrains [2], Vampire [27], Agentsheets [34], AgentCubes [35], KidSim [39] and StageCast [40] use rewrite rules combined with programming, thus reducing the burden of authoring. AgentSheets (its successor AgentCubes) and KidSim (later named StageCast) are based on a grid and use visual rewrite rules to specify changes in the grid occupancy pattern. In StageCast, children edit rules inside a separate dialog using code blocks, instead of via direct manipulation. AgentSheets is agent-based, and uses a built-in language called AgenTalk, which allows for the construction of games and simulations.

Our approach to authoring dynamic behavior is inspired by the idea behind visual rewrite rules — i.e., specification of concrete changes to objects, forming a before-after pair. However, we do not ask the user to demarcate a region for expressing a visual rewrite rule. Instead, we let them use direct manipulation to modify any block's state in response to a stimulus. BlockStudio eschews code blocks and thereby, rule editors (which depict rules using code blocks). A rule is edited by deleting and creating it. This trades off the time penalty of re-creating rules with the simplicity of no code.

ToonTalk [16] maps concurrent constraint programming onto a set of metaphors from the physical world, (e.g.: "birds carry things to their nests") allowing children to contruct programs using these graphical analogies. In ToonTalk, generalized behavior is programmed by demonstrating a concrete activity to a robot (thus "training it"), then opening up the robot's thought bubble and editing the program therein. Though training is closely related to visual changes on-screen, the connection from robot actions to these graphical objects is via symbolic manipulation of object properties.

BlockStudio differs from ToonTalk in two important regards: First, generalized rules are inferred entirely from concrete changes performed on graphical objects. This lets children avoid inspection and editing of symbolic programs. Second, objects are not imbued with special significance (like birds, robots, boxes, hammers) – thus, one can start creating artifacts without having to learn these metaphors.

Other PBE Systems

PBE has a rich history in the broader domain of end-user programming systems (see [5], [22], [28], and [29]). Early systems like Chimera [20], Mondrian [21], and Lemming [31] allowed the user to create graphical editing macros from examples, but did not have any notion of time, which precluded creation of games or animations. Pavlov [47] combined the idea of PBE with a timeline-based interface (like Macromedia Director) to allow creation of games. On average, users of the system (Master's-level students of Computer Science) required 80 minutes to learn it, before they could start building a game. Gamut [26] allowed construction of entire applications using PBE, but users required several hours to do this.

Our goals are distinct from such systems in two respects. First, we specifically focus on children ages 9-14 as our target users. Second, we wish to empower these users to prototype their creative ideas within a short time, ideally completing their first BlockStudio game within an hour.

SYSTEM DESCRIPTION

BlockStudio lets users arrange colored rectangles (*blocks*) on screen to determine the appearance of their game, following which they can specify dynamic behavior for these blocks by demonstrating *responses* to runtime *stimuli*. All blocks have attributes (color, position, size, and velocity), and any combination of these can be modified in response to stimuli.

Stimulus-Response Rules

A stimulus-response pair is called a *rule*. Stimuli include user inputs (clicks on blocks or keypresses), called *touches*, and overlap between blocks, called *collisions*. Rules are characterized by the type of stimulus (keypress, click or collision), and when blocks are involved (click and collision), parameterized by the color of the blocks involved in the stimulus. Blocks can be assigned appearances other than a solid color (like a person or a spaceship), by choosing from a built-in library of images (Figure 2c).

No Code Blocks and Little Text

BlockStudio users never encounter code blocks. Rules are applied solely based on color, meaning that a rule defined using one yellow block applies to all yellow blocks. Thus, in BlockStudio, **color determines behavior**.

There are only a few places in the interface that use text: the two rule panels (which say "Touch Rules" and "Collision Rules"), the mode button (which says "Done" in Demonstration mode), the prompt "What Happens When?" (also shown in Demonstration mode), and the label instructing users to click the game preview in order to exit the Sprite Library.

Design Interface

The BlockStudio interface (Figure 2b) consists of the *palette*, from where blocks are dragged and dropped onto the *grid*, which is a rectangular space to organize blocks. The *mode button* switches the system between various modes (we describe modes in more detail in the next subsection).

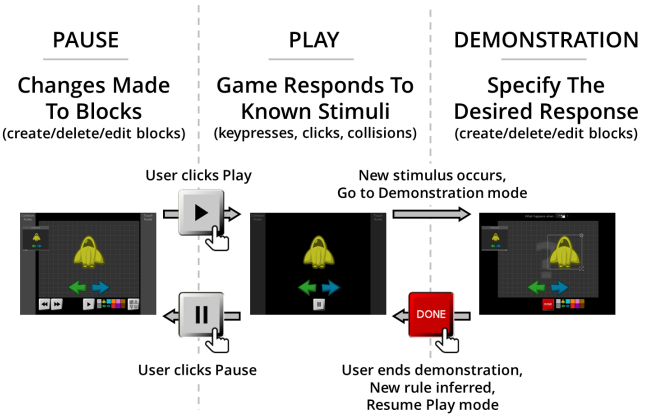


Figure 3: BlockStudio's three modes and overall workflow.

The *preview window* is always live, showing the result of playing the game/animation one second into the future. It is intended to help fine-tune block positions and velocities.

As rules are created, the *rule panels* reflect their existence via *rule widgets* (Figure 4) representing the corresponding stimuli. There are two panels, one for touch rules (keypresses and clicks) and one for collision rules. Scroll buttons are added to a rule panel when there are more rules than can be shown at once.

Clicking the *sprite button* opens the *sprite library* (Figure 2c), which lets the user assign appearances to blocks from a built-in library of images. We compute convex hulls from these images using the Gift-Wrapping method [15] and use the resulting polygons for overlap testing. If no sprite is assigned, a rectangle is used for this purpose.

The arrangement of blocks on the grid is regularly saved and accumulated in a linear history, accessible via the *Undo/Redo* buttons. There is also a *camera button*, which lets users manually save the current grid appearance to a built-in clipboard area.

Modes

The BlockStudio interface can be in *Pause*, *Play*, or *Demonstration* mode (Figure 3). In *Pause mode*, time does not advance. Here, children can arrange blocks and modify their attributes so as to achieve a desired static appearance, like for a game level or an animation. Changes made to blocks in this mode do not affect any rules. In Pause mode, clicking the mode button causes a switch to Play mode.

In *Play mode*, the game or animation is playing (i.e. time is advancing) and a stimulus causes the corresponding response to be executed, provided a rule exists for that stimulus. In Play mode, the blocks and the mode button are visible (clicking the mode button causes a switch to Pause mode). The rule panels are visible, but greyed out (to indicate that rules cannot be deleted in Play mode). All other UI elements, like

(a) (b)

Figure 4: Rule widgets in BlockStudio for various stimuli: (a) Mouse clicks on yellow, blue, and green blocks (b) Collisions between yellow-blue, green-blue, and yellow-green blocks.

the grid, the palette, and the preview window are hidden. The mode button looks like the symbol for "Pause".

While in Play mode, after encountering a stimulus for which no rule exists, BlockStudio transitions automatically to *Demonstration mode*, where the user must specify how the blocks' arrangement should be affected in response to this stimulus. Possible changes include creation/deletion of blocks and modification of their attributes, like color, position, size, and velocity. In this mode, the grid is shown, because this helps to align blocks and demonstrate precise horizontal/vertical motions. The rule panels are hidden, because one cannot delete a rule as part of a demonstration. The palette is visible, because one can create blocks as part of a demonstration. Also, the mode button turns red and displays the word "Done", indicating that BlockStudio is waiting for the user to finish demonstrating their desired response.

Clicking "Done" ends the demonstration, at which point the altered game state is examined by the inference system to create a new stimulus-response rule. The system then returns to Play mode.

The transition from Demonstration to Play mode must be initiated by the user, because it is impossible to reliably detect when the user has finished modifying the state.

Inference of rules

After the user ends their demonstration (by clicking "Done"), we synthesize generalized rules based on color from their specified changes, via a process we call *inference*. A detailed description of our inference mechanism is beyond the scope of this paper, but it relies on pattern matching, looking for differences between the blocks on the grid before and after the demonstration. For instance, a difference in position is inferred to mean "move the selected block by the specified amount". A difference in color is inferred to mean "set the selected block's color to be X". A block's presence before and absence after the demonstration implies deletion of that block, whereas the reverse scenario implies creation of a new block. Comparing block velocities before and after a collision, we can infer "bounce", "stop", etc. An inferred rule is added to the list of known rules, with a rule widget added to the appropriate Rule Panel (depending on whether it is a touch or collision rule) indicating its presence. Figure 4a shows three rule widgets corresponding to mouse clicks on blocks, while Figure 4b depicts rule widgets for collisions

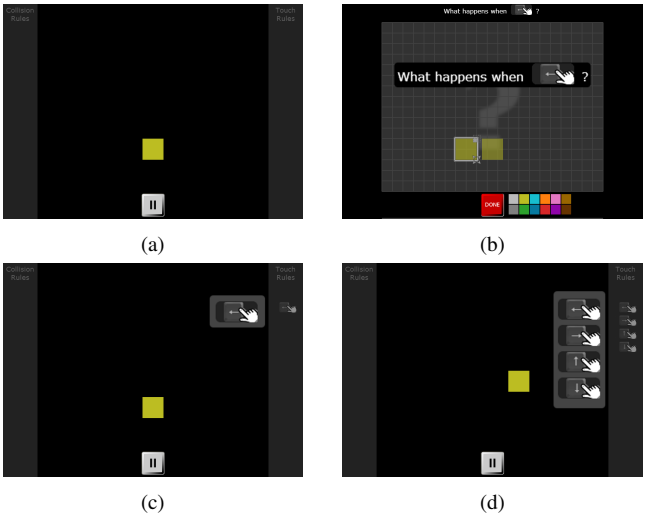


Figure 5: Specifying keypress rules in BlockStudio: (a) Create block, click Play (b) Press left arrow key, drag block left (c) Click Done (d) Repeat for other arrow keys. Now the yellow block can be moved using cursor keys. (Prompt and rule widgets in (b), (c), and (d) magnified for readability)

among blocks. Figure 5d shows rule widgets for various arrow keypresses.

While in Demonstration mode, after every mouse click, the preview window shows the effect of the inferred rule (based on the demonstration so far). Thus, users can try out various "what-if scenarios" during their demonstrations, before committing to a particular response for that stimulus.

Workflow

The authoring workflow in BlockStudio is *incremental*, *visual*, and based on *concrete examples*.

Incremental means that the artifact is built up piecemeal, and the overall design does not need to be clear at the start. Thus, movement patterns for a player's character can be specified first, or background objects can be laid out in detail before doing other things. This can be more accommodating of working styles like Turkle and Papert's "bricolage" [44].

Visual means that all input uses graphics, not text. This is in contrast to text-heavy interfaces or code blocks (annotated using text), which require users to read this text in order to understand their function.

Concrete examples means that behavior is specified via actual occurrences, instead of via abstract specifications (in the form of code). David C. Smith [38] refers to such representations as "analogical", in contrast to symbolic or "Fregean" representations. Thus, to specify the outcome for a particular situation requires the user to first set up this situation, which ties in well with an incremental approach to authoring.

For example, to program a rule of the form "when the left arrow key is pressed, all yellow blocks should move left", the user creates a yellow block, clicks Play, presses the left arrow key, drags the yellow block left and clicks "Done" (Figure 5). Thus, the user provides a concrete example of what needs to happen (moving the block), as opposed to an abstract specification thereof (like "block.x = block.x - 10"). Besides avoiding code in the form of text or blocks, this also avoids negative numbers, coordinate systems, pixels, etc.

CO-DESIGN SESSIONS

The BlockStudio interface is based on the idea of avoiding code, and minimizing text for children ages 9-14. Our long-term goal is to maximize its usability by children who struggle with text and/or find coding a challenge. As an initial step in our designs, we organized two participatory design sessions with an intergenerational co-design group of adults and children, to gauge what works well, what does not, and discover elements that could be added to BlockStudio to improve its usability. We chose to engage in Druin's Cooperative Inquiry [6] to work with children as design partners. At a broader level, we hoped that these sessions would inform our ideas for future work. We now briefly describe the co-design sessions, and continue our discussion of its results in the next section.

Each session lasted for 90 minutes. We worked with 10 child participants (four boys, six girls, ages 7-11) and half of them had some programming knowledge (i.e., Scratch), while the rest had little or none. Each session had seven adult design researchers working with the children. For each session, we began with the "Question of the Day" and presented a 10-minute demonstration on a large screen on how to use BlockStudio and then allowed them 40 minutes with the BlockStudio interface to create a game (session 1) and remix a game (session 2). In both sessions, we used a co-design technique called Stickies [45], in which we collected the children's likes, dislikes and design ideas for BlockStudio onto notes with adhesive backing. Using these notes, we created a frequency analysis of the common design and interaction themes around how to improve BlockStudio.

For the first session, we wanted to focus on the usability of BlockStudio, so we primed them with the question "what is your favorite game?", before letting them explore the system for 40 minutes for game creation. The second co-design session focused on modification of existing creations. The second co-design session focused on modification of existing creations, which is a longer-term goal. Therefore, we primed the children with the question "if you could change something about a game, what would it be?". For this session, we gave them three existing BlockStudio artifacts as a "starting point" and gave them around 10 minutes to modify a copy of each of them. The artifacts were as follows:

An animation of a mouse avoiding a cat and eating some cheese.

A puzzle game, where a block had to be moved through a maze (using the cursor keys) to a star. Certain parts of the maze had to be visited in a certain order to unblock obstacles.

A space game, where falling asteroids had to be aimed at using the left/right arrow keys, and shot by pressing the spacebar to fire a pellet.

In both sessions, we arranged the participants into dyads (five dyads total), with an adult observer per dyad. We encouraged children to think aloud, while the observers coded their remarks as "Likes", "Dislikes", or "Design Ideas" using the Stickies technique.

Findings from first session

The overall feedback from the children was that they enjoyed using the system to author games. Their Likes showed a positive outcome for usability ("easier than Scratch", "easy and I have control over it", "deleting rules") and variety ("spawning new things on collision", "collision possibilities", "world creation control"). Dislikes included block creation ("having to click-drag to create things"), elements being too small ("when square too small, hard to control"), or not small enough ("blocks not thin enough"), and certain system-wide choices, like automatic block deletion ("blocks going offscreen should not be deleted, but instead just stop there"). Noteworthy Design Ideas included expanding existing possibilities ("more colors", "a color picker to select your own color"), introducing new possibilities ("make/upload own artwork", "backgrounds", "sounds", "rotate blocks"), and shortcuts ("button to duplicate block", "button to reset all colors to default rectangle appearance", "button to clear grid").

Findings from second session

The Likes gathered from the second session showed a positive perception of the variety possible ("making a story completely different", "creating a teleporter", "like that you can modify the game").

Many Dislikes and Design Ideas were similar to those from the first session, like asking for more customization ("more designs", "background music", "sound effects").

A new Dislike we saw was due to personal preference ("dislike storytelling – needs more thinking and editing"). Most of the remaining Dislikes and Design Ideas were from children struggling to figure out the existing rules in these artifacts ("hover text explanation of collision rules that exist", "use text to describe an item", "don't know what brown blue boxes are doing").

DISCUSSION

The first session provided positive feedback that the system was fairly usable by children with ten minutes of instruction, providing a fast ramp-up for these first-time users. Though some of our young co-designers had prior programming experience and the rest did not, BlockStudio held the interest of both sets of children, which allows us to recognize the potential of environments that are free of code and text-heavy elements. Some children wanted to automatically create more objects, at which point they were taught how to construct a loop-like design pattern (bouncing a block repeatedly between two other blocks). Once they knew the pattern, they were able to use it for their own purposes. This has multiple implications for systems that don't use code blocks.

First, these findings show that children (some of whom had never written programs), were able to understand the idea underlying loops, and then implement it in their own games, implying that the next step would be to explain to them what they did and how it relates to coding. A separate study will be necessary to establish how well this sort of understanding transfers to actual programming tasks, but we consider this a favorable outcome for our early design prototype. This should be of interest to CS education researchers, because this could allow individuals with learning disabilities to learn programming, using non-textual methods.

In addition, we have observed other design patterns in prior informal sessions, including "lives" (respawning a character), "score" (counting up/down), "progress bar" (filling up a space), etc. Thus, even systems devoid of code blocks have a vocabulary of design patterns, which novices cannot be expected to know. Therefore, designers of code-free systems need to teach these patterns in a scaffolded way, paying attention towards eventually transitioning learners to code. Template-based systems often have a library of such patterns ready to drag-and-drop into one's project. Thus it might be worth studying how to let advanced users save and reuse patterns that they have created.

During the second session, the children were now comfortable with the interface and knew how to construct games using the system. Focusing their attention on modifying existing artifacts revealed that children tended to erase all existing blocks and rules before authoring their game. This is unsurprising, since there is no mechanism in our current prototype for describing the response corresponding to a particular rule. This suggests that future work should focus on finding ways to visually convey what the game's code "does". We believe this visualization challenge to be a rich space for exploration by UI designers. Even for textual or block code based systems, it might be worth quantifying how often children read and then modify programs in systems using code blocks, compared to a system like BlockStudio.

One participant was interested in seeing the generated code that controls the blocks. He was the most advanced user in the group and had experience creating games in Scratch. While contrary to our goal of eliminating code from the interface, this suggests that we could let novices create programs using PBE, but after a certain level of proficiency, reveal the inner workings via code blocks. One possibility is that BlockStudio can be designed to help lead into more complex block coding environments (e.g., Scratch).

Though BlockStudio is primarily aimed at older children (ages 9-14), we are curious to investigate how well adult novices can use this system, especially if they have struggled to create programs using conventional methods. In particular, we would like to conduct further studies investigating how BlockStudio might allow families to learn, create, and code together. Our overarching goal is to enable groups of diverse individuals to collaborate, combining their strengths to realize creative ideas. This early version of BlockStudio is our first step towards this objective.

CONCLUSION

We have presented an exploratory prototype system called BlockStudio, aimed at empowering young novices to create

2D games and animations without writing code. We have summarized existing authoring systems and contrasted their code and/or text-based approaches with ours. A brief description of our system was provided, followed by future directions for BlockStudio and other similar systems, gleaned from two participatory co-design sessions with children. Our initial findings contribute ideas to CS education researchers and UI designers by showing the possibility of separating the concepts underlying coding from the text aspects thereof.

SELECTION AND PARTICIPATION OF CHILDREN

We found ten children (4 boys and 6 girls) using snowball sampling, and their ages ranged from 7 through 11. We had IRB consent and assent throughout the process. Ethnically, there were 4 White, 2 Native, 2 mixed heritage (Asian/Black, White/Asian), and 2 Asian participants, and they were a mix from public schools and homeschooling (N = 1).

ACKNOWLEDGEMENTS

The first author would like to thank artists Brian Britigan, Barbara Krug, and Marianne Lee for creating BlockStudio artwork, Nova Barlow for coordinating early pilot studies, and Dan Butler, Kathleen Tuite, and Adam Smith for thoughtful feedback on early prototypes of the system. We also wish to thank our KidsTeam UW co-designers, without whose help this work would not have been possible.

REFERENCES

1. Bandura, A., Barbaranelli, C., Caprara, G. V., and Pastorelli, C. Self-efficacy beliefs as shapers of children's aspirations and career trajectories. *Child development* (2001), 187–206.
2. Bell, B., and Lewis, C. Chemtrains: A language for creating behaving pictures. In *Visual Languages, 1993., Proceedings 1993 IEEE Symposium on*, IEEE (1993), 188–195.
3. Chandler, P., and Sweller, J. The split-attention effect as a factor in the design of instruction. *British Journal of Educational Psychology 62*, 2 (1992), 233–246.
4. Cooper, S., Dann, W., and Pausch, R. Alice: a 3-d tool for introductory programming concepts. In *Journal of Computing Sciences in Colleges*, vol. 15, Consortium for Computing Sciences in Colleges (2000), 107–116.
5. Cypher, E., and Halbert, D. C. *Watch what I do: programming by demonstration*. The MIT Press, 1993.
6. Druin, A. Cooperative inquiry: developing new technologies for children with children. In *Proceedings of the SIGCHI conference on Human Factors in Computing Systems*, ACM (1999), 592–599.
7. Flannery, L. P., Silverman, B., Kazakoff, E. R., Bers, M. U., Bontá, P., and Resnick, M. Designing scratchjr: Support for early childhood learning through computer programming. In *Proceedings of the 12th International Conference on Interaction Design and Children*, ACM (2013), 1–10.
8. Games, Y. Game maker. `https://www.yoyogames.com/gamemaker`, 2007. [Online; accessed 20-May-2016].
9. Google. Blockly. `https://code.google.com/p/blockly/`, 2012. [Online; accessed 20-May-2016].
10. Google. Pencil Code. `https://pencilcode.net/`, 2014. [Online; accessed 20-May-2016].
11. Guzdial, M. Programming environments for novices. *Computer science education research 2004* (2004), 127–154.
12. Hasan, B. The influence of specific computer experiences on computer self-efficacy beliefs. *Computers in Human Behavior 19*, 4 (2003), 443–450.
13. Howland, K., Good, J., and Nicholson, K. Language-based support for computational thinking. In *Visual Languages and Human-Centric Computing, 2009. VL/HCC 2009. IEEE Symposium on*, IEEE (2009), 147–150.
14. Inc., G. GameSalad. `http://www.gamesalad.com/`, 2009. [Online; accessed 20-May-2016].
15. Jarvis, R. A. On the identification of the convex hull of a finite set of points in the plane. *Information Processing Letters 2*, 1 (1973), 18–21.
16. Kahn, K. Toontalk - an animated programming environment for children. *Journal of Visual Languages & Computing 7*, 2 (1996), 197–217.
17. Kelleher, C., and Pausch, R. Lowering the barriers to programming: A taxonomy of programming environments and languages for novice programmers. *ACM Computing Surveys (CSUR) 37*, 2 (2005), 83–137.
18. Kelleher, C., Pausch, R., and Kiesler, S. Storytelling alice motivates middle school girls to learn computer programming. In *Proceedings of the SIGCHI conference on Human factors in computing systems*, ACM (2007), 1455–1464.
19. Ko, A. J. Attitudes and self-efficacy in young adults' computing autobiographies. In *Visual Languages and Human-Centric Computing, 2009. VL/HCC 2009. IEEE Symposium on*, IEEE (2009), 67–74.
20. Kurlander, D. Graphical editing by example in chimera. *Watch What I Do: Programming by Demonstration, Cypher, A.(ed.)* (1993), 270–290.
21. Lieberman, H. Mondrian: a teachable graphical editor. In *INTERCHI* (1993), 144.
22. Lieberman, H. *Your wish is my command: Programming by example*. Morgan Kaufmann, 2001.
23. Lionet, F., and Lamoureux, Y. Klik and Play. `http://knpforschools.webs.com/`, 1994. [Online; accessed 20-May-2016].
24. MacLaurin, M. B. The design of kodu: a tiny visual programming language for children on the xbox 360. In *ACM Sigplan Notices*, vol. 46, ACM (2011), 241–246.

25. Mason, R., and Cooper, G. Mindstorms robots and the application of cognitive load theory in introductory programming. *Computer Science Education 23*, 4 (2013), 296–314.

26. McDaniel, R. G., and Myers, B. A. Building applications using only demonstration. In *Proceedings of the 3rd international conference on Intelligent user interfaces*, ACM (1998), 109–116.

27. McIntyre, D. W., and Glinert, E. P. Visual tools for generating iconic programming environments. In *Visual Languages, 1992. Proceedings., 1992 IEEE Workshop on*, IEEE (1992), 162–168.

28. Myers, B. A. Taxonomies of visual programming and program visualization. *Journal of Visual Languages & Computing 1*, 1 (1990), 97–123.

29. Myers, B. A., Ko, A. J., and Burnett, M. M. Invited research overview: end-user programming. In *CHI'06 extended abstracts on Human factors in computing systems*, ACM (2006), 75–80.

30. Nintendo. Super Mario Maker. `http://supermariomaker.nintendo.com/`, 2014. [Online; accessed 20-May-2016].

31. Olsen Jr, D. R., Ahlstrom, B., and Kohlert, D. Building geometry-based widgets by example. In *Proceedings of the SIGCHI conference on Human factors in computing systems*, ACM Press/Addison-Wesley Publishing Co. (1995), 35–42.

32. Pane, J. F., and Myers, B. A. The impact of human-centered features on the usability of a programming system for children. In *CHI'02 Extended Abstracts on Human Factors in Computing Systems*, ACM (2002), 684–685.

33. Pierce, J. S., Audia, S., Burnette, T., Christiansen, K., Cosgrove, D., Conway, M., Hinckley, K., Monkaitis, K., Patten, J., Shochet, J., et al. Alice: easy to use interactive 3d graphics. In *Proceedings of the 10th annual ACM symposium on User interface software and technology*, ACM (1997), 77–78.

34. Repenning, A. 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*, ACM (1993), 142–143.

35. Repenning, A., and Ioannidou, A. Agentcubes: raising the ceiling of end-user development in education through incremental 3d. In *Visual Languages and Human-Centric Computing, 2006. VL/HCC 2006. IEEE Symposium on*, IEEE (2006), 27–34.

36. Resnick, M., Maloney, J., Monroy-Hernández, A., Rusk, N., Eastmond, E., Brennan, K., Millner, A., Rosenbaum, E., Silver, J., Silverman, B., et al. Scratch: programming for all. *Communications of the ACM 52*, 11 (2009), 60–67.

37. Salen, K. Gaming literacies: A game design study in action. *Journal of Educational Multimedia and Hypermedia 16*, 3 (2007), 301–322.

38. Smith, D. C. *Pygmalion: A computer program to model and stimulate creative thought*. Birkhäuser Basel, 1977.

39. Smith, D. C., Cypher, A., and Spohrer, J. Kidsim: programming agents without a programming language. *Communications of the ACM 37*, 7 (1994), 54–67.

40. Smith, D. C., Cypher, A., and Tesler, L. Programming by example: novice programming comes of age. *Communications of the ACM 43*, 3 (2000), 75–81.

41. Sony. Little Big Planet. `http://littlebigplanet.playstation.com/`, 2008. [Online; accessed 20-May-2016].

42. Staffieri, A. GameMaker. `http://alstaffieri.com/gamemaker.html`, 1995. [Online; accessed 20-May-2016].

43. Sweller, J. Cognitive load theory, learning difficulty, and instructional design. *Learning and instruction 4*, 4 (1994), 295–312.

44. Turkle, S., and Papert, S. Epistemological pluralism: Styles and voices within the computer culture. *Signs* (1990), 128–157.

45. Walsh, G., Foss, E., Yip, J., and Druin, A. Facit pd: a framework for analysis and creation of intergenerational techniques for participatory design. In *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems*, ACM (2013), 2893–2902.

46. Wilson, D.-M. Multimodal programming for dyslexic students. In *Proceedings of the 6th International Conference on Multimodal Interfaces*, ICMI '04, ACM (New York, NY, USA, 2004), 343–343.

47. Wolber, D. Pavlov: Programming by stimulus-response demonstration. In *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems*, ACM (1996), 252–259.

48. Wolber, D., Abelson, H., Spertus, E., and Looney, L. *App Inventor*. " O'Reilly Media, Inc.", 2011.