Combining Big Data and Thick Data Analyses for Understanding Youth Learning Trajectories in a Summer Coding Camp

Deborah A. Fields
Instructional Technology & Learning Sciences, Utah State University
2830 Old Main Hill
Logan, UT, USA
deborah.fields@usu.edu

Lisa Quirke
Faculty of Information
University of Toronto
140 St. George Street
Toronto, ON, Canada
lisa.quirke@mail.utoronto.ca

Janell Amely, Jason Maughan
Instructional Technology & Learning Sciences, Utah State University
2830 Old Main Hill
Logan, UT, USA
jamely@aggiemail.usu.edu; jsnmaughan@gmail.com

ABSTRACT
In this paper we explore how to assess novice youths' learning of programming in an open-ended, project-based learning environment. Our goal is to combine analysis of frequent, automated snapshots of programming (e.g., "big" data) within the "thick" social context of kids' learning for deeper insights into their programming trajectories. This paper focuses on the first stage of this endeavor: the development of exploratory quantitative measures of youths' learning of computer science concepts. Analyses focus on kids' learning in a series of three Scratch Camps where 64 campers aged 10-13 used Scratch 2.0 to make a series of creative projects over 30 hours in five days. In the discussion we consider the highlights of the insights—and blind spots—of each data source with regard to youths' learning.

Categories and Subject Descriptors
K.3.2 [Computers and Education]: Computer and Information Science Education – Computer science education

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

1. INTRODUCTION
One difficulty educators have faced is how to evaluate novice learners' programming or, more broadly, computational thinking [1], especially when projects are open-ended without a single "correct" solution [2]. Since Seymour Papert introduced the idea of constructionism [3], many computer science educators have sought to engage kids in learning through interest-driven, project-based programming. This playful approach to learning-by-designing can be motivating for learners and also useful for developing complex thinking and debugging skills [4]. Learners do not work on identical problems that have a single, favored solution, instead creating original projects, which allow for diverse programming styles such as "structured" or "bricolage" [5]. Being open and free-form, this approach can result in a wide variety of programs that can differ wildly. This creates challenges for educators and researchers who are tasked with measuring and assessing novice learning, especially at larger scales of analysis.

SIGCSE '16, March 02-05, 2016, Memphis, TN, USA
 ISBN 978-1-4503-3685-7/16/03…$15.00
DOI: http://dx.doi.org/10.1145/2839509.2844631

Researchers have taken two primary approaches to tackling this problem of assessing novice learning in open-ended programming environments. One approach involves qualitative analysis to gain a rich view of novice learning. Some have approached qualitative analysis by looking at multiple saves of programs at regular intervals in order to see incremental progress [e.g., 6]. Others conduct in-depth think-aloud interviews to capture novices' thinking in the process of solving a targeted problem or bug [e.g., 7]. These provide excellent ways to explore novices' thinking and creating practices, but can only be used across small sets of youth because of the labor involved. An alternative approach includes mining programs with quantitative methods, to capture the learning of large numbers of novice youth. This is particularly useful to study a wide variety of programmers online, for instance on the Scratch website, in order to identify types of programmers [e.g., 8] or types of trajectories [e.g., 9]. However, one of the challenges of this approach is evaluating the quality of programs. Often, the measures of programming are relatively basic, involving simple frequency counts of commands (or blocks) in a program as representative of learners' understanding of a concept.

How then can researchers and educators assess novices' learning of programming in rich ways that also allow for analysis across a wide number of youth? In this paper we explore the potential of an approach that begins with mixed methods, using both qualitative and quantitative means to understand novices' learning. The overarching goal is to develop an approach that applies quantitative measures to the analysis of programming education in a way that moves beyond frequency counts. This approach could be used to assess novices' understanding and use of several programming concepts across 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. Our goal is to combine analysis of frequent, automated snapshots of programming (e.g., "big" data) within the "thick" social context of kids' learning, With this information we consider how many times, or over what time period, does it take for kids to pick up a presented concept?

2. BACKGROUND
In recent years researchers have taken several approaches to using "big" data to understanding novice programmers' learning paths (for a review see [10]). A few overarching approaches stand out. First, some use big data to find patterns across hundreds or thousands of participants in free choice environments, participants

who might have vastly different projects. To do this, researchers have sorted individual code blocks into categories of programming concepts. For instance, in Maloney, Peppler, Kafai, Resnick and Rusk's [11] two-year study of a Computer Clubhouse, frequency counts of programming blocks in Scratch stood as measurements for how many youth used various concepts such as loops, variables, conditionals, or Booleans. Building on this general approach, others have used frequency counts of blocks to assess the quality of users' programs on the Scratch site, adding on analytical methods like latent class analyses or K means clustering to document classes of programmers [8] or types of trajectories of programming [9]. The Scrape tool developed by Wolz, Taylor and Hallberg [12], allows any researcher or educator to manipulate this frequency count technique by dragging individual or multiple programs into the tool and categorizing the blocks by type (motion, looks, variables, control, etc.). These studies have carved new territory in making sense of what massive numbers of people are doing on websites such as Scratch, or in large free-choice informal programs. The basic presence of specific blocks in programs, however, does not necessarily denote sophisticated understanding of the related programming concepts. Work by Brennan and Resnick [13] highlights this point, noting the potential for major differences between learners' programs and their actual understanding of how their own code worked.

Alternatively, research may focus on process-based data to gain insights into novices' pathways of learning. In some cases these studies focus on comparing novices' programs to an ideal solution, noting divergences from a "correct" form of solving a problem. For instance, Repenning et al [14] used game code developed by hundreds of kids to assess its similarity to set solutions. Similarly, Blikstein et al [2] captured novices' thinking in the process of solving a targeted checkerboard problem, documenting "sink states" and different pathways to the most efficient solutions. Yet these approaches do not facilitate assessment in open-ended, constructionist-type programming. A few have begun to develop ways to document different pathways to learning in open-ended programming environments. For instance, Werner, McDowell and Denner [15] laid the groundwork to use logfile data from Alice to study "high level actions" as a precursor to analyzing types of problem solving strategies. Similarly, Berland et al [16] used learning analytics to identify students' progress through different programming strategies (tinkering, exploring, and refinement) across two hours of a playful introduction to programming in iPro. While these studies lay the foundation for using big data to study processes of programming over time, we take an approach that uses process-based "big data", or large, automatically-generated quantitative datasets, in conjunction with in-depth qualitative analyses [17] in order to gain insights into the richness of constructionist-based learning that might be hidden by just one method [10, 18].

Our approach is uniquely mixed in methodology: we developed quantitative measures of programming concepts, after deep qualitative analysis of changes in students' projects over time, applying them across more than 600 project saves per student (roughly every 2 minutes). We subsequently evaluated the utility of this approach by comparing these analyses with knowledge from rich case studies of students' changes to projects over time. This allowed us to take an exploratory approach to studying learning in an open-ended programming environment with the long-term goal of using the quantitative measures across a variety of open-ended programming environments. Below we briefly describe the Scratch programming environment, outline our study's methods, and share our findings regarding measures tracking youths' learning of three concepts: initialization, events, and parallelism.

3. CONTEXT: SCRATCH SUMMER CAMP

This study examined ways of developing measures to assess the learning of middle-school youth learning to program using Scratch. Scratch 2.0 (Scratch.mit.edu) is an online, blocks-based visual programming environment, designed to engage children in creating and sharing projects, while learning the foundations of computer programming [19]. Developed by researchers at MIT and launched in 2007, Scratch is one of the most-used programming environments for novices. Scratch and other block-based programming tools (e.g., Alice, Greenfoot, App Inventor) are designed to remove the frustration of syntax errors, while allowing users to develop their knowledge of coding concepts.

The questions examined in this paper emerge from a research study on kids' learning trajectories with Scratch. The project includes data collected at a set of three, week-long "Scratch Camps" held at an intermountain university in partnership with the local 4-H club during the summer of 2014. The camp took place over 30 hours in five days, with approximately 25 hours devoted to programming. Campers aged 10-13 used Scratch 2.0 to make a series of creative projects. They were mentored and taught by one professor and three graduate students, with several high school 4-H club members also offering assistance. We created a series of open-ended, genre-specific project challenges with constraints intended to impel students to learn particular techniques. Campers were given feedback on their projects throughout the camp, and encouraged to use the programming strategies (e.g., initialization, debugging) taught in the daily lessons. We included a variety of projects: Scribble time & Name project (Day 1), Story project (Day 2), Music Video project (Day 3), Video Game project (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.

4. METHODS

4.1 Data Collection

Sixty-four campers (34 girls and 31 boys) consented to participate in the study across three separate Scratch Camps (out of 67 campers total). Campers ranged in age from 10-13. Some had prior experience with Scratch, but most had only played with it for a couple of hours or minutes prior to the camps.

Several types of data were collected. Since the campers were programming on the online version of Scratch 2.0, we were able to use JSON files of student program code collected online with the help of the Scratch Team at MIT, with snapshots of campers' code every time they shifted from editing costumes, backgrounds, or sound, back into coding, and by default every two minutes. This provided 32,465 code snapshots in text-based format, the big data component of our project. Although JSON files contain all of the text of a program, this file cannot be run in Scratch to see if a project works or what it actually looks like while running.

To complement these data, we manually collected projects for 32 campers: five campers from the first camp, five from the second camp, and all campers (22 consenting) from the final camp. These projects were collected roughly once every 75 minutes, at the end of each day's four programming sessions. One researcher also wrote field notes at each camp, focusing on five campers per camp, providing more context on campers' goals, struggles, interactions, and learning moments.

4.2 Analysis

Analysis took shape through three major iterations, and although we include it in the methods section here, the process of analysis is also a significant part of our findings. First, we conducted a qualitative analysis of the 32 campers for whom we had Scratch project saves, creating case studies of these youths' learning. These case studies documented each addition and edit to campers' Scratch projects at the end of each 75 minute session. Every change in projects between sessions was noted and analyzed. Where available, we considered the observational notes to provide context on campers' intentions and any help they received.

Second, with backend data in hand, we began a process of cleaning and reducing it. We created a Scratch JSON parser that captures and quantifies details of each code snapshot (a text-based version of a Scratch project). Then we developed a second layer of analyses that quantified changes in Scratch programs over time. These became a series of quantitative measures in several categories of programming concepts, inspired by Brennan and Resnick's [13] outline, but including other concepts we found pertinent to evaluating youths' learning in the Scratch Camps: Booleans, conditionals, event driven & parallelism, initialization, loops, randomization, and sensing (for more details on all of the measures see: www.workingexamples.org/example/show/727). Each measure was tested and developed iteratively, carefully comparing the quantitative measures with the 32 case studies. Further investigation of campers' Scratch programs was performed as needed to see what each measure revealed about youths' learning and what they missed or glossed over. Finally, we sought to glean which measures were the most useful. Initially we began with a list of 46 measures, and after analyzing them we narrowed the list down to 26 measures to use in further analysis.

Although we included frequency counts of blocks (as done previously in other studies [8, 11, 12]), our quantitative measures of programming concepts include several features that take a next step in terms of evaluating whether youth are using a concept correctly. Each measure we developed only included blocks attached to event hats in Scratch. In Scratch, students can drag blocks into the programming space somewhat randomly—this allows for them to test blocks individually without connecting them to an "event hat" that integrates the test blocks into the program. After some analysis we decided not to include these "floating blocks" in our study of kids' learning in Scratch.

To keep with a constructionist mindset and authentic practices on the Scratch website, we purposefully allowed, and sometimes encouraged, remixing. Remixing is an authentic practice on the Scratch site and can be used effectively to introduce youth to new programming strategies. However, it introduced a large number of blocks that the campers did not program themselves, creating a dilemma in analyzing youths' learning, as we wanted to see only what campers added, not what they had remixed from others' work. After some trial and error, we developed two sets of visualizations. One included all changes, but marked changes that were likely part of a remix (e.g., adding 30 blocks between saves within 2 minutes—see Figure 1). The other included only the changes between saves that involved fewer than 30 blocks, removing any instance where added blocks within 2 minutes numbered over 30. This visualization showed cumulative changes over time (see Figure 2).

5. FINDINGS

In this section we outline three selected measures we developed: initialization, events, and parallelism. These were chosen because they were present in campers' programs early on and throughout the camp allowing us to see how well our methodological approach showed learning over time. We also highlight examples that illustrate how these reveal campers' learning over time.

5.1 Sets of Measures

We begin by briefly describing initialization, events, and parallelism in Scratch, as these concepts may differ in their representation in other languages. Our measures represent the common ways in which our novice campers approached these programming concepts, but do not include every presentation of these concepts in the Scratch environment.

Initialization means setting a starting state or point for all of the actions and actors in a program. In Scratch, *initialization* requires setting the start state of a character (i.e., "sprite"), and needs to be clearly defined by programming. Initialization states in Scratch include: position, direction, costume, graphic effect, layer, showing or hidden, and size. For instance, if a sprite moves, then its starting position needs to be set; if a sprite is hidden, then it needs to be set to *"show"* at a particular time. Although Brennan and Resnick [13] do not include initialization in their outline of programming concepts, we found it to be a challenge for youth to learn and implement in the camps and elsewhere [6].

We initially developed 15 measures for initialization, narrowing them down to 10 that showed the most promise in differentiating campers' learning. The ten measures focus on initialization in position, graphic effects, show/hide, scene (e.g., background), and user-created variables. Five of these measures focus on "missing initialization" where a need for initialization is present. For instance, for the "Missing Position Initialization" measure, if a sprite moves in the program, this measure will increase if there is no programming to set the sprite's starting position. Alternatively, for "Sprites with Position Initialization," we counted the number of sprites that move and whose positions were initialized *before* that movement occurred in the program. By setting the position initialization before the movement of the sprite, the campers' program would return to the correct start position that they had envisioned, instead of restarting in the wrong location.

Event and parallel programming involve coordinating processes and events within a program such that they can be executed sequentially or in parallel without bringing the program to a standstill. In Scratch, coordinating multiple sprites' actions involves learning different forms of synchronization and event driven programming [11]. In particular, events can be triggered with the commands *"broadcast"* and *"when I receive".* Our measure for events, "Broadcasts Received" includes only broadcasts that have a matching receive (i.e., "broadcast scene 2" and "when I receive scene 2") with some code blocks following the receive. Related to the broadcast event, parallelism in Scratch can be accomplished in two prominent ways (at least for novices): by using multiple event hats such as *"when green flag clicked,* or multiple *"when I receive"* hats to trigger new actions to start at the same time, within or across sprites. We trace these by counting the number of "broadcasts with multiple receives" and "sprites with multiple green flags."

Below we illustrate how these measures, compared with contextual data, provide insight into the learning of one camper, Ginger, a 12 year-old girl with very limited Scratch experience (i.e., "four days") before the camp began.

5.2 Initialization

In our analysis of campers' trajectories, we found that campers learned initialization early but continued to refine their learning throughout the projects they created. As projects became more complex throughout the camp and as new forms of initialization were needed (for instance variables in the Game Project), youth struggled to identify all of the missing initializations in their projects. In other words, although nearly all youth began to introduce initialization into their projects on Day 1, they missed incidences on subsequent days. This became visible in a pattern of spikes and plateaus in the five measures of missing initialization shown in our code snapshots visualization.

Ginger's data on missing initialization (see Figure 1) shows how many initializations are missing at each automated save of her projects. A close look at the spikes and plateaus reveals an interesting insight into her debugging patterns (indicative of

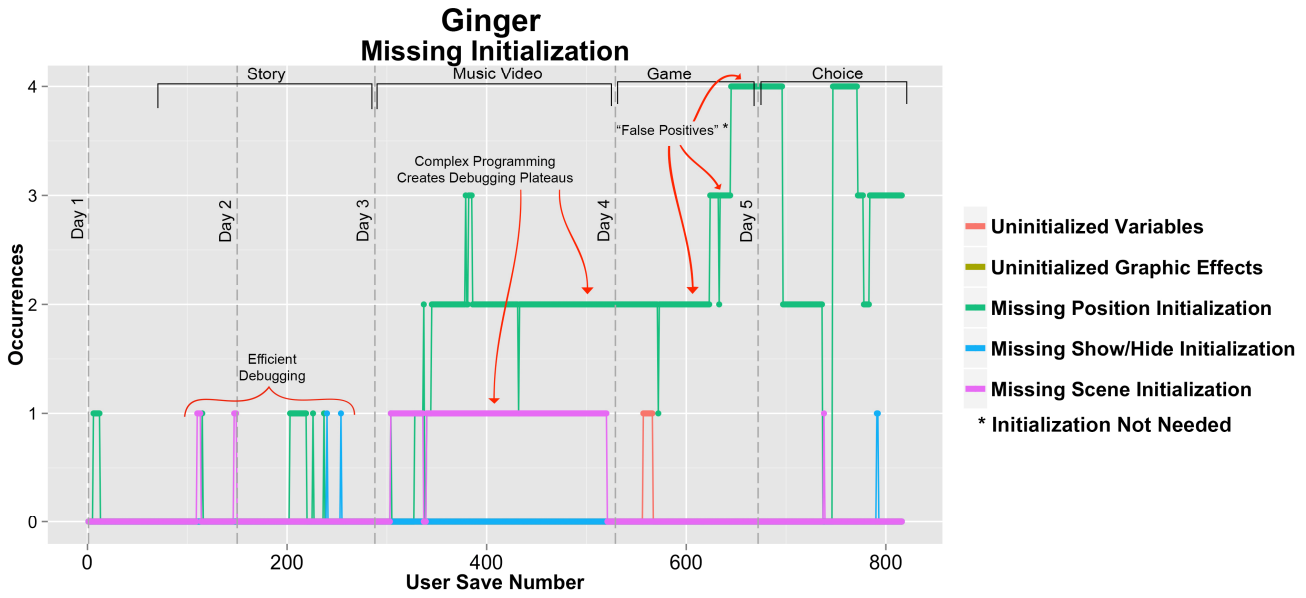

Figure 1. Missing initialization in Ginger's projects across saves. Code snapshot view (includes remixes).

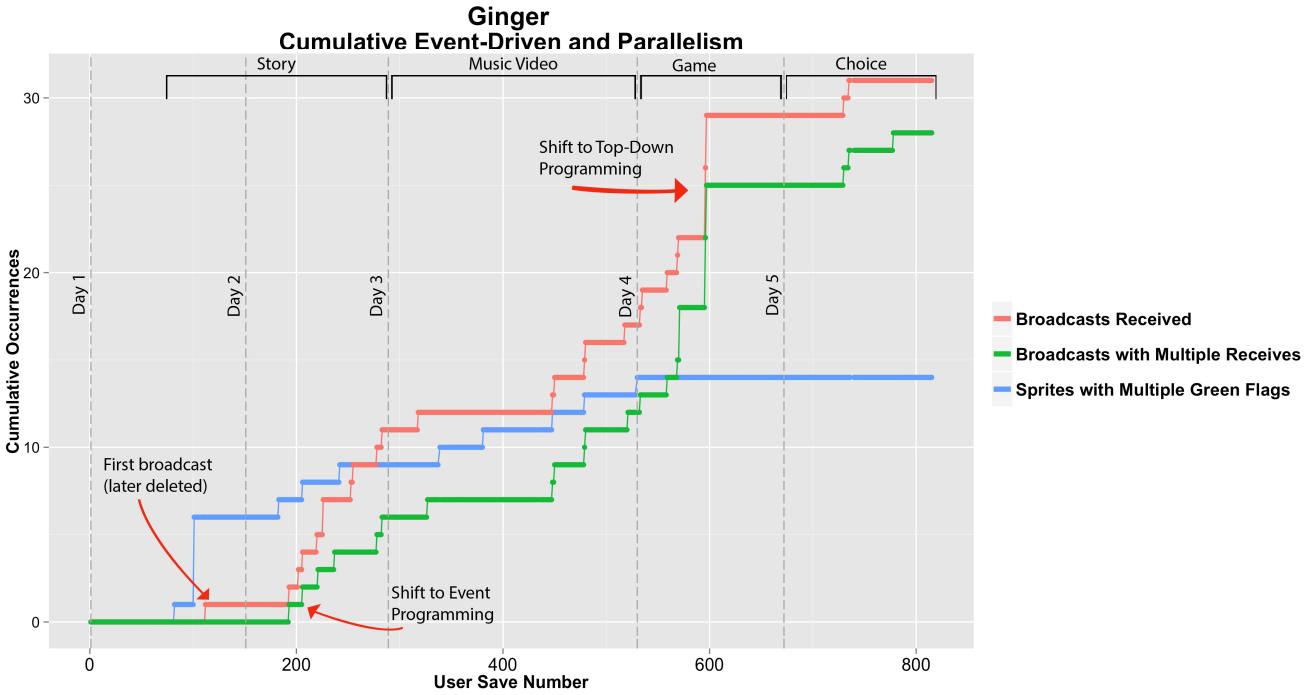

Figure 2. Event-driven and parallel programming in Ginger's projects. Cumulative view (no remixes).

patterns we witnessed in other youth). The spikes represent an introduction of an occurrence of a missing initialization that is almost immediately fixed. For instance, the green spikes on Days 1 and 2 show how Ginger introduced movement into some sprites, and quickly adjusted to set their initial positions. In contrast, the plateaus show where missing initializations remain in her projects across many saves. Plateaus occur for scene initialization (in pink) in Day 3, and both spikes and plateaus for position initialization (in green) begin on Day 3 and continue into Days 4 and 5. These later projects quickly became larger and more complex. For many, it became more difficult to detect and fix missing initializations.

This insight into quick or slow adjustments—the debugging of missing initializations—is a special view provided only through the automated saves. Since our manual project saves were only captured once per session (approximately every 75 minutes), hand-coding could not reveal the quick debugging strategies campers used. Careful study of Ginger's hand-saved projects (.sb2 files), however, reveals some limitations of the current measures for initialization. The chart (Figure 1) shows that Ginger was missing position initialization for four sprites by the end of her game day. However, looking at her game saves, we see a larger perspective. Ginger chose to remix a "catch" game where the player had to move a basket left and right at the bottom of the screen in order to catch an object falling from the sky at random points. The basket did not require the same kind of position initialization needed in other projects, where setting both the x and y coordinates was necessary. In this case, the user-controlled basket only moved left and right. Its initial position had no influence on the game, so it created a false positive in the missing position initialization measure. In a sense, Ginger's project was more sophisticated with its use of user-controlled position than these particular measures could detect.

5.3 Events and Parallelism

While initialization patterns described above were similar among most campers, events and parallel programming revealed several differences. Early on in our analysis we noticed that campers struggled with the concepts of events and parallelism, even when introduced to the concept multiple times in the front of class through targeted instructions and "show and tell" of campers' projects. Some campers picked up on the idea of using multiple green flags in a single sprite early on during the first day as Ginger did, visible in the blue line in Figure 2. Others did not introduce multiple green flags until day 3, or began to use multiple receives with a single broadcast before using multiple green flags (a different use of parallel programming). A similar range of trajectories was apparent with events: some campers began early and used it often, others picked up on it later. Although Ginger was one of the early adopters of event programming in the camps, a closer look at her trajectory of learning shows some interesting moments. The chart in Figure 2 illustrates Ginger's cumulative use of broadcasts, broadcasts with multiple receives, and sprites with multiple green flags. Unlike Figure 1, this chart illustrates only *cumulative* progress across saves with incremental changes (less than 30 changes in blocks between saves), a strategy we adopted with many measures to remove the noise created by remixes.

Shifts in Ginger's programming style become apparent when we compare our qualitative findings with the automated saves. As seen in Figure 2, Ginger first introduced a broadcast late in Day 1, about the time the concept was first presented to the camp. However, she did not add any more broadcasts until part-way through Day 2 (see the flat red line). Qualitative analysis of her projects revealed that she did not use any broadcasts on Day 1: any code that involved broadcasts introduced during instructions was quickly deleted. The stairway-like progression on Day 2 (Figure 2) illustrates Ginger's introduction of new broadcasts and broadcasts with multiple receives. Qualitative analysis revealed a distinct shift in Ginger's programming. Earlier, she programmed very linearly, for instance having Sprite 1 *"say"* something then *"wait 2 seconds"* while Sprite 2 would *"wait 2 seconds"* and then *"say"* something in return. Midway through the second session on Day 2, Ginger began using the broadcast command to coordinate sprites' actions as events, quickly introducing three working broadcasts, most with multiple receives, to orchestrate conversation, movement, and costume changes between sprites. She continued to program with events across the Music Video and the Game, although she found using multiple green flags per sprite (connected with *"wait until"* code triggered with the built-in timer) equally useful in the Music Video. The spike mid-way through Day 4 shows a second shift in Ginger's programming. Before this point, she had steadily built her code on an as-needed basis, also known as bottom-up programming. However, at this spike Ginger shifted to a more top-down programming style, laying out the eight levels of the game all at once with broadcasts, setting the requisite variables, and then filling in any details.

6. DISCUSSION

The research presented in this paper shows one possibility for the measurement of computer programming concepts across time and projects among novice learners using Scratch in an open-ended, constructionist environment. By combining big data and deeply contextual qualitative analyses, our study highlights the insights—and blind spots—of each data source with regard to youths' learning of initialization, events and parallelism. Going beyond frequency counts as representative of learning, allowed us to document the iterative ways that users approached particular concepts (e.g., the way debugging initialization occurred frequently or slowly in simple versus more complex projects). Further, these analyses opened up the possibility of documenting changes in programming styles, as Ginger did in first integrating events as a style of bottom-up programming in multiple projects, and later shifting to using events in a top-down style of programming. The tracing of specific concepts in learners' extremely diverse projects brings us closer to determining ways to assess open-ended projects in constructionist environments.

In this paper due to space constraints we have highlighted only three of the eight programming concepts developed with sets of measures for this analysis. Our goal was to highlight the contributions these types of quantitative measures could make to studying novices' learning trajectories across multiple types of interest-driven projects and to use qualitative analyses to test the interpretive utility of these methods. The remaining sets of measures provide further insights into youths' programming, especially as the campers went deeper into more sophisticated programming with nested loops, conditional logic, and data usage (variables) [20]. As part of the larger project we are developing a "FUN! Tool" (functional understanding navigator), a flexible open source data engine (github.com/ActiveLearningLab) where measures for Scratch or other languages could be input to automatically analyze findings across individual programmers. The tool is meant to help researchers write and share automated data extraction and manipulation techniques with Scratch and other types of programming language.

The multi-method structure of this research strengthened our understanding of learners' developing conceptualizations of

programming in ways that a single method could not. Measures of programming concepts applied to frequent saves provided a glimpse at small moves that youth made in their programs not captured in manual saves each session. This provided a clear look into learning that would not normally be captured in qualitative methods. Yet qualitative methods helped not only in creating the original measures, but also in pointing out blind spots in the quantitative measures and in interpreting what they showed. This "thick" approach to data analysis builds robustness into the analysis of "big" data. At the same time, our approach was also limited in certain ways. The lack of pre- and post-tests of youths' performance, for example, and the potential for false positives in some measures, either due to more complex uses of a concept or as a result of remixed content, could be addressed in future research. Our research also took place in a particular time and space with specific pedagogical goals; other approaches could result in different learning and require alternative interpretations of measures or new measures entirely.

Using big data to analyze microgenetic processes of novices' programming [10] has great potential for use across education and research, to assess learners' understanding of key concepts in situ, and potentially reach out to youth to improve learning. This study has particular implications for understanding and mapping the diversity of learning in constructionist environments, formal or informal, which can support novices' interest and creativity while challenging them to deepen their skills and thinking.

7. ACKNOWLEDGMENTS

This material is based upon work supported by a collaborative grant from the National Science Foundation (NSF#1319938) to the first author. 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. Special thanks to Sayamindu Dasgupta and Mitchel Resnick of the MIT Media Lab for help collecting data. Nicole Forsgren, Xavier Velasquez, Tori Horton, and Katarina Pantic assisted in analysis. Suzanne Fluty and Whitney King aided in leading the Scratch Camp along with 4-H volunteers.

8. REFERENCES

[1] Wing, J. 2006. Computational Thinking. *Commun. ACM*. 49, 3 (2006), 33-35.

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

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

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

[5] Turkle, S. and Papert, S. 1991. Epistemological pluralism: Styles and voices within the computer culture. *Signs* (1990), 128–57.

[6] Fields, D., Vasudevan, V. and Kafai, Y. B. 2015. The programmers' collective: fostering participatory culture by making music videos in a high school Scratch coding workshop. *Interact. Learn. Envir*, (2015), 1-21.

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

[8] Fields, D. A., Giang, M., and Kafai, Y. 2014. 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*, (2014). ACM, New York, NY, 2–11.

[9] Yang, S., Domeniconi, C., Revelle, M., Sweeney, M., Gelman, B. U., Beckley, C., and Johri, A. 2015. Uncovering Trajectories of Informal Learning in Large Online Communities of Creators. Paper presented at *L@S 2015 / Learning* (Vancouver, BC, Canada, March 14-18, 2015).

[10] Berland, M., Baker, R. S., and Blikstein, P. 2014. Educational data mining and learning analytics: Applications to constructionist research. *Technology, Knowledge and Learning*, *19*, 1-2 (2014), 205-220.

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

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

[13] Brennan, K. and Resnick, M. 2012. 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).

[14] Repenning, A, Webb, D. C., Koh, K. H., Nickerson, H., Miller, S. B., Brand, C., Her Many Horses, I., Basawapatna, A., Gluck, F., Grover, R., Gutierrez, K. and Repenning, N. 2015. Scalable game design: A strategy to bring systemic computer science education to schools through game design and simulation creation. *ACM Trans. Comput. Educ.* 15, 2, Article 11 (April 2015), 31 pages. DOI: http://dx.doi.org/10.1145/2700517

[15] Werner, L., McDowell, C., and Denner, J. 2013. A first step in learning analytics: pre-processing low-level Alice logging data of middle school students. *Journal of Educational Data Mining*, 5, 2, (2013), 11-37.

[16] Berland, M., Martin, T., Benton, T., Petrick Smith, C. and Davis, D. 2013. Using Learning Analytics to Understand the Learning Pathways of Novice Programmers. *J. Learn. Sci.* 22, 4 (2013), 564–99.

[17] Kafai. Y. B. and Fields, D. A. 2013. *Connected Play: Tweens in a Virtual World*. Cambridge, MA: MIT Press.

[18] Grover, S., Pea, R. and Cooper, S. 2015. Designing for deeper learning in a blended computer science course for middle school students. *Comp. Sci. Ed.* 25, 2 (2015), 199-237.

[19] Resnick, M., Maloney, J., Hernández, A. M., Rusk, N., Eastmond, E., Brennan, K., Millner, A. D., Rosenbaum, E., Silver, J., Silverman, B., and Kafai, Y. B. 2009. Scratch: Programming for everyone. *Commun. ACM.* 52, 11 (2009), 60–67.

[20] Velasquez, X. and Fields, D. A. 2015. When good video games promote good programming: Scratch Camp FTW. *Proceedings of the 11th annual Games+Learning+Society Conference*. (Madison, WI, July 2015)