UNIVERSITY OF CALIFORNIA

Santa Barbara

Programming Environments for Children: Creating a Language that Grows with you

A Thesis submitted in partial satisfaction of the

requirements for the degree Master of Science

in Computer Science

by

Charlotte Hill

Committee in charge:

Dr. Diana Franklin, Lecturer, Co-chair

Professor Tobias Hollerer, Co-chair

Professor Danielle Harlow

June 2015

The thesis of Charlotte Hill is approved.

____

Diana Franklin, Committee Co-chair

____

Tobias Hollerer, Committee Co-chair

____

Danielle Harlow

June 2015

ABSTRACT

Programming Environments for Children: Creating a Language that Grows with You

by

Charlotte Hill

Recent efforts have increased the number of elementary and middle schools teaching computer science — but do they have the right tools for the job? Elementary school teachers are usually responsible for teaching all subjects, and often do not have a background or training in computer science. Fourth through sixth grade students are still developing their math and reading skills as well as learning how to type and use computers. Fortunately, computer science is one of the only domains that can adapt to meet the needs of the user. Unlike math or physics, computer science has few constants; computers, languages, and development environments have changed over the last decades and will continue to evolve. How can programming languages and environments better meet the needs of upper elementary classes learning computer science? This paper looks at designing block-based programming environments for upper elementary school students as a part of a larger research study on early computer science education.

Block-based programming environments let children create complex, visual programs without worrying about compiling or syntax errors. This paper describes the research studies completed in the design and implementation of block-based programming

environments created alongside the development of KELP-CS, a computational thinking curriculum for 4th — 6th grade. Both the programming environment and curriculum were piloted in schools across California as part of a design-based research project.

TABLE OF CONTENTS

I. Introduction 1

II. Related Work 3

A. Block-based Programming 3

B. Computer Science in Elementary & Middle School 6

III. Prior Experience 10

A. Animal Tlatoque 10

B. Hairball.. 11

C. My Contributions 12

IV. KELP-CS 14

A. Overview of Depict & KELP-CS 14

B. Guiding Research Questions 14

C. Learning Progressions... 15

D. Overview of the Curriculum 16

E. Digital Storytelling Module, 2013-2014 17

F. Digital Storytelling Module, 2014-2015 17

G. My Contribitions.. 19

H. Future Work... 20

V. The Octopi Application Suite 21

A. Overview.. 21

B. Guiding Research Questions 22

C. Background & Motivation 22

D. Implementation 23
E. OctopiStudent & OctopiDeveloper 25
1. Script and sprite options 26
2. Customizing the language and interface 27
3. Separating development and runtime 28
4. Snapshots of student work 29
F. OctopiResearcher & Collecting Student Work 30
G. KelpPlugin & Understanding Students' Ideas about Initialization 31
H. Lessons Learned & Future Work 34
VI. LaPlaya & Octopi 37
A. Overview 37
B. Guiding Research Questions 37
C. Methods & Findings 38
1. The interface 3842
2. The language 39
D. Motivation & Design Principles 40
1. Support multiple types of tasks 4242
2. Require only grade- and age- appropriate content 42
3. Include an age-appropriate interface 43
4. Support project developers 43
E. Implementation 43
F. Reading LaPlaya Programs 4444
G. Future Work 46

VII. Conclusion.. 48

References.. 50

LIST OF FIGURES

Figure 1. A Blockly project from Code.org's "Frozen" Hour of Code 3

Figure 2. A Scratch project 4

Figure 3. A ScratchJr project 5

Figure 4. Comparison of Hairball and student researcher scoring of broadcast and receives. Hairball detected 100% of instances of broadcast and receives in student projects, while student researchers missed 12 instances when assessing projects 12

Figure 5. Depict's hypothesized learning progressions for computational thinking . 16

Figure 6. Components of the revised digital storytelling module 18

Figure 7. A project in Scratch version 1.4 21

Figure 8. The mammals project from the KELP-CS digital storytelling module in OctopiDeveloper 25

Figure 9. The mammals project from the KELP-CS digital storytelling module in OctopiStudent 26

Figure 10. Student view of KelpPlugin feedback for a KELP-CS assignment on the Octopi Submit website 31

Figure 11. KelpPlugin results of initialization in two KELP-CS initialization projects – Animal Sprint, the original project, and Pinata, the revised project. 33

Figure 12. The ballerina project from the animation assignment from the revised KELP-CS digital Storytelling module, open in LaPlaya in developer mode 41

Figure 13. The ballerina project from the animation assignment from the revised KELP-CS digital Storytelling module, open in LaPlaya in student mode 42

Figure 14. How students used visual cues to predict aspects of a LaPlaya program. Parentheses distinguish visual cues that were categorically false affordances. “X” signifies that students used a visual cue when making predictions about the first project 46

I. Introduction

Graphical programming environments make code come to life for elementary school students. In many modern block-based languages, colorful characters appear next to the scripts that make them run, jump and dance. These environments do more than just reduce syntax errors; they draw novice programmers into a world that can be programmed and modified almost as quickly as their imaginations come up with possibilities. Over the past decade, options for block-based languages have grown with their increase in popularity. The social media boom has impacted graphical programming as well, and new websites and online communities encourage programmers to share their block-based programs online for others to play and modify. Unlike most text-based languages, block-based languages are often designed for children. Scratch, a popular online block-based programming environment, was developed for 8-14 year olds.

Studies show that introducing students to concepts at a younger age makes them more likely to pursue that field in the future. Programming has been taught to children for decades, but with mixed results. There are many layers to this problem, but this work will attempt to address one: common programming languages weren't designed with children in mind, and pose many challenges when used in elementary school classrooms. This thesis addresses the question: *How can programming environments better meet the needs of upper elementary classes learning computer science?* To answer this question, I created and implemented design principles for a block-based programming environment for upper elementary school students as a part of a larger, design-based research study on early computer science education. I began researching early computer science education as an undergraduate at UCSB, and continued in the masters program at UCSB. For my masters, I

worked as part of a group researching how 4th — 6th graders learn computer science, and my research focused on creating programming environments for those age groups.

This paper starts with a look at related work and my prior work in the field. Next, I describe the KELP-CS curriculum, a computational thinking curriculum for 4th — 6th grade. Then, I describe two different programming environments and associated tools created for KELP-CS: Octopi, a modification of Scratch; and LaPlaya, a reimplementation of Octopi with design principles based on the KELP-CS pilot and other block-based languages. Finally, I conclude with a review of design principles for block-based environments for children and suggestions for future work in this area.

II. Related Work

A. Block-based Programming

Growing interest in computer science education has led to an increase in the language and editor options for children learning to code. Many of these languages are block-based, allowing students to ignore syntax and focus on creating programs. Block-based languages provide a set of programming commands in Lego-like blocks that users drag together like puzzle pieces to create scripts. These scripts often control characters or other images that can be used to create a game, tell a story, or more. Block-based programming languages lower the cognitive barrier to programming by reducing possible syntax errors and providing a smaller set of commands than most textual programming languages.

Many block-based environments are designed for touch screen devices, since typing

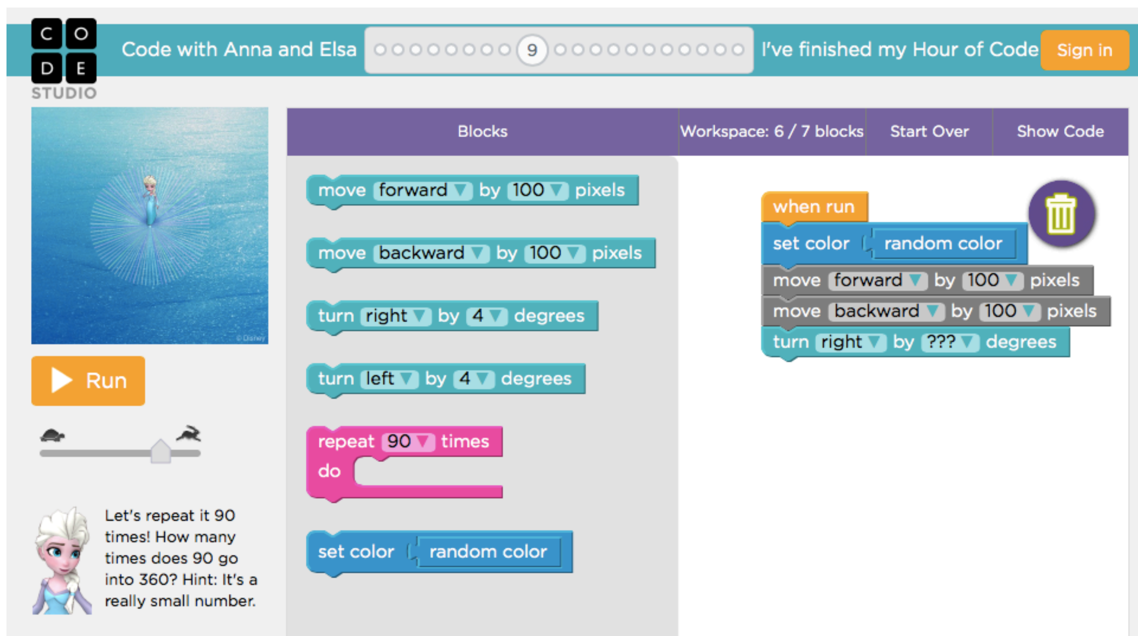

Figure 1. A Blockly project from Code.org's "Frozen" Hour of Code

is limited and drag and drop is the primary method of interacting with the application. Hopscotch, a block-based environment available for iPads, provides a set of colorful characters that users can program to create their own stories or games [23]. Other applications provide a more structured environment for learning to code. Lightbot (available for iPads, Android, and online) guides users through a set of interactive tutorials and puzzles to teach users how to program [29]. Similarly, code.org has sets of programming tutorials for different age groups in Blockly, a block-based language developed by Google (see Figure 1) [5, 1]. The inspiration for these applications can be seen in Scratch (Figure 2), a block-based programming environment developed by MIT [40]. Scratch also has an online community where users can share their Scratch projects. Other users can “remix” (similar to “save as”) and create their own project using one posted to the website. Block-based languages like Scratch can improve students’ attitudes and increase confidence in

Figure 2. A Scratch project

programming [28].

Block-based environments have colorful, interactive interfaces with limited typing, making them a natural fit for children. However, developers of block-based environments for children should still consider the developmental levels and limitations of the target age group. Young children have had fewer linguistic experiences than adults, making them less likely to understand common technological metaphors such as a button with an image of a floppy disk indicating that the button is used to save [4]. Children may also have difficulties with using the mouse and typing [17]. For example, children struggle more than adults to hold down the mouse for longer periods of time, leading to difficulties with the drag and

Figure 3. A ScratchJr project

drop control scheme common in most block-based languages [17]. Designers should consider hiding advanced tools that children may get "lost" in, as complicated interfaces can distract younger users [18].

The development of ScratchJr (Figure 3), which is based on Scratch, is an interesting example of adapting a successful block-based environment for a younger age group. Developers made numerous changes to both the interface and programming language to better fit the needs of the target audience, 5-7 year olds, rather than the 8-16 year olds that Scratch was designed for. The ScratchJr designers removed many of the interface options, such as the "instant gratification" buttons that produced an immediate effect (like adding a sprite), so that children would spend less time playing with the interface and more time creating [26]. However, *creating* does not aways translate to *coding* — kindergarteners using ScratchJr in their class spend a significant amount of time using the paint editor rather than coding [12].

B. Computer Science in Elementary & Middle School

Children in elementary and middle school are often referred to as "digital natives" — since children today have never known a time without computers or the internet, adults may see them as naturally technologically adept. However, children are still at different developmental stages than adults and their academic knowledge is rapidly increasing as they progress throughout elementary school and into middle school. Developmental psychology is an important factor when deciding whether – and if so, how – to teach programming to children [8]. Some computational thinking concepts required for more advanced programming might be beyond the developmental stage of an age group. Seiter and Foreman created a model showing the progression through which elementary students move when

learning computational thinking [39]. Older age groups are better suited to reach higher levels of understanding of computational thinking material; for example, using a "repeat" loop versus using a "repeat until __" loop and specifying a condition that makes the repeat end.

Elementary and middle school students' academic knowledge is expanding, a development environment for this age group must be flexible and expand as they learn, or the students will outgrow it. The focus of this paper is on fourth through sixth grade, an important transitional time for children's development. In second and third grade, "the mechanical demands of learning to read are so taxing at this point that children have few resources left over to process the content. In fourth through eighth grade, children become increasingly able to obtain new information from print" (p. 333) [38]. Fourth through sixth graders differ academically and developmentally in important ways from their younger and older counterparts. These students are developing linguistic, kinesthetic, and cognitive skills necessary to successfully interact with computers. At the same time, this age group is learning key concepts from other fields, like math and science, needed to successfully program. Many math concepts used in Scratch, such as division, negative numbers, and percentages, are taught during upper elementary school. These factors should be taken into account when developing programming environments or programming curricula for children.

Children have been programming for decades. In *Mindstorms: Children, Computers, and Powerful Ideas*, Seymour Papert discusses the invention of Logo, an early educational programming language, and case studies of young students learning the language [35]. Papert's work on students learning other subjects such as math and physics through

programming and his theory that teachers at any grade level could incorporate programming into their curricula set the stage for the next decades of computer science education research. Since computer science is not a part of the standards for K-12, resources for early computer science education range from lesson plans which teachers incorporate into their regular lessons to online tutorials that guide students through the material. ScratchEd takes the first approach, with lesson plans for teaching coding with Scratch and an online community where educators can help each other with the material [41]. The Computer Science Teaching Association offers more general standards for teachers to follow when incorporating programming into their K-8 classes [7]. Other curricula, such as those provided by code.org, are made up of interactive tutorials that can be used in or out of the classroom. Computer Science Education Week has grown in popularity in recent years, and now multiple platforms provide short tutorials or projects every December for students to try out programming [6].

Despite these developments in programming environments and curricula as well as a nation-wide push to include programming at more grade levels, the vast majority of children do not learn coding in schools prior to high school. Limited teacher education and lab resources make it challenging to introduce computer science in middle and elementary schools. Elementary school teachers are usually responsible for teaching all of the subjects, and rarely have a background in computer science. Some strategies to assist teachers without computer science subject knowledge are all-inclusive online tutorials (such as the previously mentioned code.org) or automated grading tools to make grading finished projects or finding struggling students easier for teachers [3]. Additionally, although most schools now have computer labs with internet access in order to accommodate standardized testing, these labs

are in various states of usefulness — when we piloted our curriculum, labs we encountered ranged from new, networked netbooks controlled by a district technology supervisor to rooms of old PCs without lab managers or anyone at the school with administrator privileges. Successful computer science curricula must be flexible enough to work in all these types of classroom environments.

III. Prior Experience

A. Animal Tlatoque

Prior to entering the masters program, I worked as an undergraduate researcher at the Animal Tlatoque summer camp [13]. Animal Tlatoque was a part of a larger research effort to study middle school students' attitudes towards computer science and how extracurricular programs could influence these attitudes, as well as to teach programming concepts. Although the underlying goal of the summer camp was to teach Scratch programming, computer science was deliberately intertwined with other subjects to attract students who might not otherwise be interested in a computer science camp. Mayan culture and animal conservation were picked as the other subjects of the camp to attract both Latinas/os and girls, respectively, two groups who are minorities in computer science.

The first summers of Animal Tlatoque successfully focused on developing a camp that would appeal to these demographics and improve students' perceptions of computer science [13]. During the third summer, the Animal Tlatoque team also studied whether or not the camp participants were actually learning computer science concepts [14]. UCSB students working as camp assistants recorded the level of help they gave to students during the camp. Then, after the camp, a team of computer science undergraduate researchers (myself included) looked at each of the students' projects for the different assignments to determine to what extent the students met the goals of the assignments. We also ran these projects through an automated grading assistance tool to double-check the undergraduates' assessments of the projects. This assessment showed that students were able to use several key computer science concepts — such as event driven programming and initialization — without much help from the camp assistants [14].

B. Hairball

Assessing Scratch projects can be tedious and time-consuming work. There isn't any easy way to open multiple files at once or search for items inside of a file. Additionally, code is split up by sprites, so a grader has to look at each sprite's code and determine its place within the larger program. We created an auto-grading tool, Hairball, to speed up the grading process for Scratch projects. The base of Hairball takes any number of Scratch projects and uses a Python library, Kurt [27], to turn these projects into dictionaries. Then, Hairball can run any number of Hairball plugins on these projects and return a report of the results. Custom Hairball plugins assess all of the assignments for Animal Tlatoque that the undergraduate researchers also graded by hand. Afterwards, we compared Hairball's results with the undergraduate researchers' results.

Hairball grades projects as "correct" or "incomplete" for each attribute checked by a plugin. Hairball was built to be an auto-grading tool, not the complete grading system, so "incomplete" projects are ones that need a closer look by a human grader. This is partially because of the visual nature of Scratch and the types of things we were assessing. For example, Hairball did not find a lot of "correct" instances of animation, because what we recognize as an animation is hard to define by the code that produces it — you just know it when you see it. In this case, there were a lot of "incomplete" projects marked by Hairball that would need manual grading by a person who could watch the projects to determine whether or not they include animation. Hairball was more effective for other subjects, such as message passing. In Scratch, correct message passing includes a broadcast block with a name of a message, and then a corresponding script that starts with a "When I receive" block with the same message name. Complex broadcast/receive projects may have multiple

messages and broadcast/receive scripts across many sprites, making them hard to grade by hand. In this case, we found that Hairball did a better job than the undergraduate researchers at determining whether projects were correct or not (Figure 4). Although Hairball cannot take away the job of the grader completely, it simplifies grading for large numbers of Scratch projects.

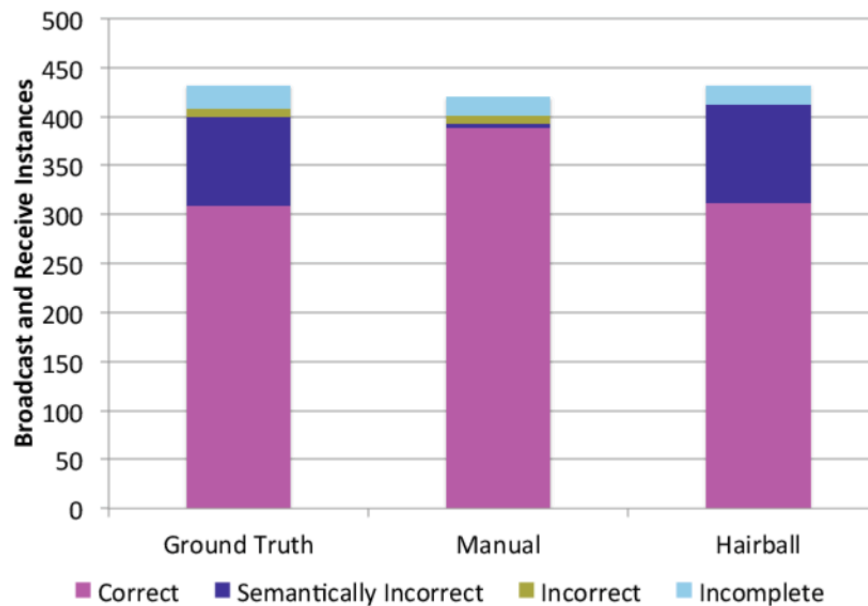

Figure 4. Comparison of Hairball and student researcher scoring of broadcast and receives. Hairball detected 100% of instances of broadcast and receives in student projects, while student researchers missed 12 instances when assessing projects

C. My Contributions

I worked on numerous aspects of Animal Tlatoque during my summer as an undergraduate researcher. I worked alongside the rest of the team to create camp curricula before the start of the camp, and answer students' questions and assist the lesson instructors during the camp. However, my primary focus was on Hairball and assessing student work. Throughout the summer, I wrote the Hairball plugins used to assess the Scratch projects.

After the camp, two other undergraduate researchers and I manually graded all of the students' Scratch projects. Then, I ran the Hairball plugins on the projects and compared the results. Some projects received different "grades" from Hairball and the manual graders, so I went through each of those projects a second time to establish the "ground truth" grade, or the final, accepted grade. Finally, I made graphs of the results for all of the projects (See Figure 4 for an example). These results, and the other work I did during the summer, were showcased in two papers at SIGCSE [3, 14].

IV. KELP-CS

A. Overview of Depict & KELP-CS

Depict is an interdisciplinary research group with members from the Education and Computer Science department. Led by Danielle Harlow and Diana Franklin, its goal is to study how 4th-6th graders learn computational thinking. In 2013, Depict developed a set of learning progressions for computational thinking at this age group, and tested the lower anchor points in focus groups. The results of these focus groups informed both the learning progressions, and the design of a computational thinking curriculum and programming environment for 4th — 6th grade [10]. We piloted this programming environment and the 4th grade module of the curriculum in the 2013-2014 school year, and then used a refined version of each in additional classrooms during the 2014-2015 school year. My research focus in the Depict group has been on developing programming environments for 4th — 6th graders [20]. In this section, I'll describe the learning progressions and curriculum created for KELP-CS. In the following sections, I'll go into more detail on the design, implementation, and testing of the programming environments developed for KELP-CS.

B. Guiding Research Questions

Our work on the KELP-CS curricula is inspired by several research questions. First, *what are the lower anchor points, or knowledge that students have of computational thinking before formal instruction?* And *what are the learning progressions fourth through sixth graders follow when learning computational thinking?* In Section C, I briefly describe the KELP-CS learning progressions. We tested the lower anchor points of two strands of these learning programs during focus groups at local elementary schools [9, 10]. Next,

Sections D, E, and F describe the curriculum we created to answer the question, *how can these learning progressions be successfully implemented in an elementary school classroom?* Finally, Sections V and VI address the question, *what types of tools are needed in an elementary school computational thinking class?* Sections V and VI describe programming environments and other tools I created for classes and researchers to use with the KELP-CS curricula.

- What are the lower anchor points, or knowledge that students have of computational thinking before formal instruction?
- What are the learning progressions fourth through sixth graders follow when learning computational thinking?
- How can these learning progressions be successfully implemented in an elementary school classroom?
- What types of tools are needed in an elementary school computational thinking class?

C. Learning Progressions

Depict created a series of learning progressions for computational thinking (Figure 5) based on the CSTA learning progressions and findings from Animal Tlatoque [13, 14]. Learning progressions are the series of partial understandings, or steps one takes when learning a new subject. The lower anchor points of these learning progressions are the base knowledge on which the rest of the learning progressions are constructed, and are concepts or skills that children of the target age group would have with little or no instruction. We tested our lower anchor points in focus groups at elementary schools in Santa Barbara [9, 10]. Our findings from the focus groups changed the lower anchor points of our learning progressions and informed our curriculum design.

D. Overview of the Curriculum

The Kelp-CS curriculum was originally intended for 4th — 6th grade, with a module for each grade level. These modules would teach different aspects of computer science and computational thinking according to the learning progressions, and would each have their own overarching theme. As of April 2015, the fourth and fifth grade modules are complete,

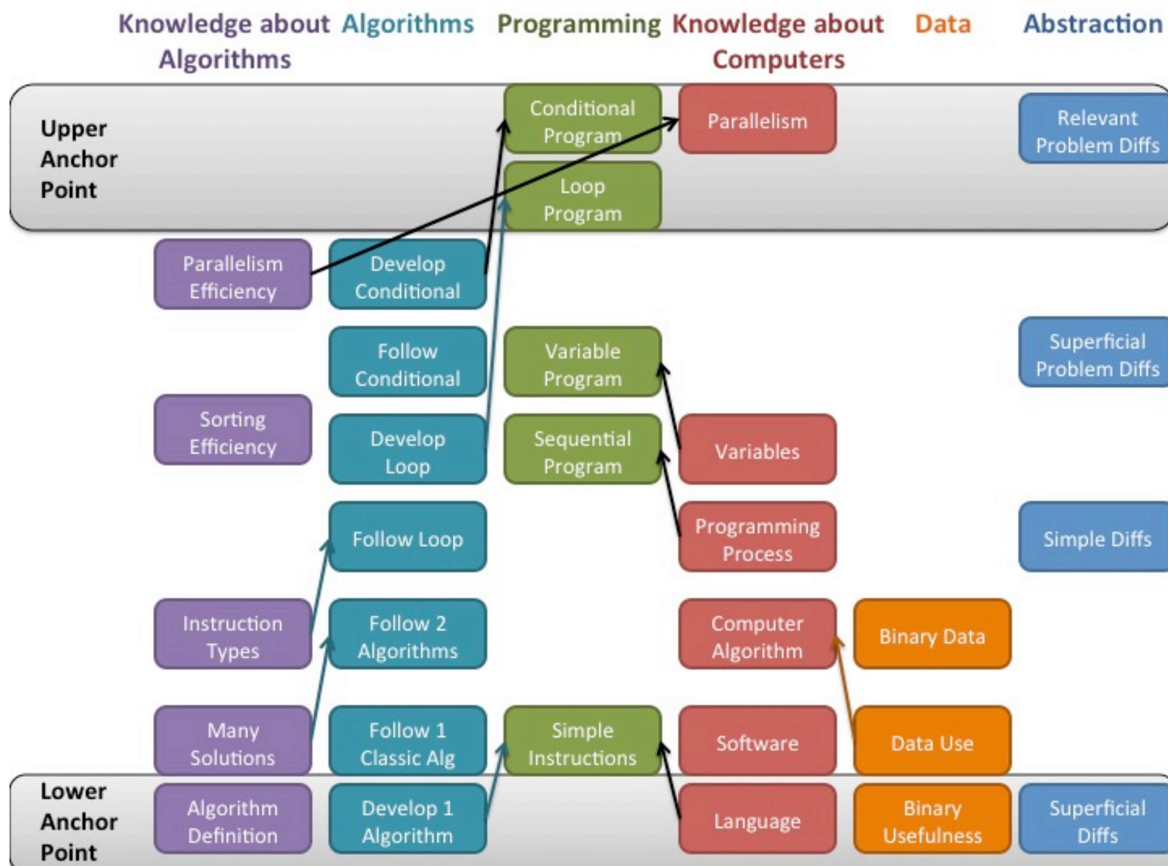

Figure 5. Depict's hypothesized learning progressions for computational thinking

and the fourth grade module was piloted in classrooms across California. The fourth grade module, Digital Storytelling, teaches sequential and event-driven programming, initialization, and animation, which the students then incorporate into their own digital stories [15]. The fifth grade module, Game Design, teaches message passing, loops, sensing

and decisions, and variables, which students then use to create their own games. The proposed sixth grade module would teach parallelism and using hierarchy and abstraction to break down problems. The original fourth grade module was piloted in the 2013-2014 school year, and then taught in more schools during the 2014-2015 school year with modifications based on findings from the pilot.

E. Digital Storytelling Module, 2013-2014

The fourth grade module, Digital Storytelling, was originally imagined with multiple tracks: each track would tie to different grade-level content such as, for example, California history. The programming assignments used for the pilot spanned the themes, so that different assignments would focus on different subject areas, rather than only testing one subject track. The module consisted of 8 programming assignments, or Wired-Up activities, and 4 off-computer exercises, or Fired-Up activities. Each Wired-Up activity taught a new concept, such as initialization, which the accompanying Fired-Up activity tied back to students' every day lives.

The fourth grade pilot was taught in fifteen classrooms in California with over 400 students. Graduate student researchers from Depict taught or helped out at local classrooms, and computer lab teachers taught the module at the distant classrooms. We collected on-computer and on-paper student work from all the classrooms, as well as video and audio recordings and graduate student researchers' analytic memos from the local classrooms.

F. Digital Storytelling Module, 2014-2015

Depict made numerous changes to the module based on the results of the pilot (Figure 6). The pilot showed that students struggled with programming assignments that relied on other subject knowledge, so the programming assignments were modified to eliminate outside subject knowledge [16]. A new programming environment, LaPlaya, and online curriculum platform, Octopi, were created to better fit the needs of the students and their teachers. The programming assignments were split into smaller tasks that built off of each other, so that students could receive more feedback while working during the short lab time. The pilot showed that students struggled with initialization and message passing, so the initialization assignment was expanded to present the need for initialization in multiple ways, and the programming language was altered so that message passing could be moved into the fifth grade module instead. Additionally, the digital storytelling aspect of the module was extended so that students could work on their stories throughout the quarter, instead of starting them at the end. The new digital storytelling format also includes an emphasis on design thinking, which is a part of the Next Generation Science Standards [30]. Finally, we added multiple choice assessment questions to all of the programming

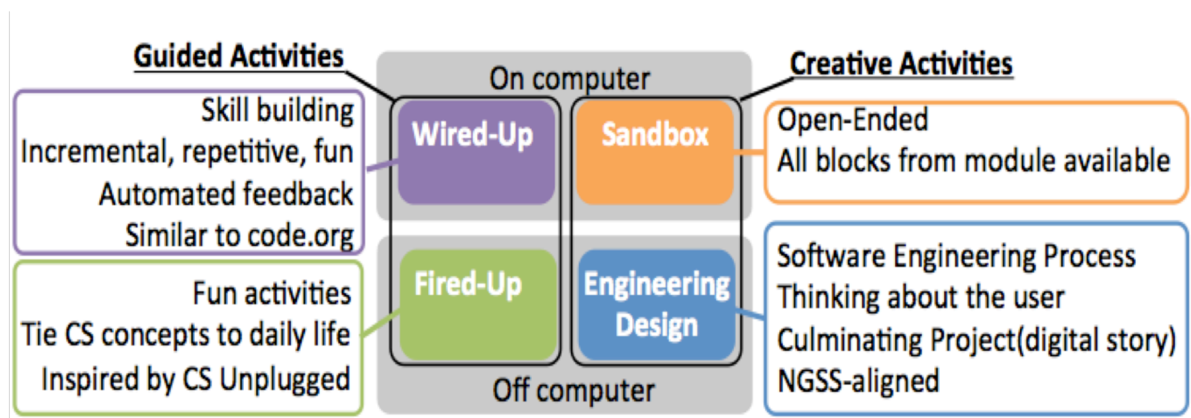

Figure 6. Components of the revised Digital Storytelling Module

assignments. Although they have not yet been tested, these assessment questions will gather more data on how students learn computational thinking and programming in KELP-CS.

The revamped module was also taught in classes across California. Depict sent graduate student researchers to local classrooms, and collected data through online and on-paper student work, videos of student work and classrooms, graduate student researchers' analytic memos, audio recordings of graduate student researchers and teachers helping students, and iPhone videos taken by students of their own work. Additionally, we interviewed teachers and held focus groups with students to gather data on how KELP-CS could be improved, and how students engaged with the programming content.

G. My Contributions

My primary focus as a member of the Depict team was developing programming environments to use with KELP-CS. I created three Scratch modifications, OctopiStudent (and a PC version), OctopiDeveloper, and OctopiResearcher, as well as a Snap! modification, LaPlaya [32, 33, 34, 21]. I was the sole developer for the Octopi applications, and the primary developer for LaPlaya (I was the sole developer for about four months, and then worked and supervised others working on it as well for around six months). I was also the primary developer for KelpPlugin, an extension of Hairball with a new set of plugins. I wrote the original architecture as well as several plugins. I used these plugins to research student ideas about initialization during the KELP-CS pilot [22].

I also worked with the rest of the Depict team on curricular development and research. I analyzed the videos of focus groups that informed our learning progressions and curricula [10]. I helped create multiple assignments for Module 1 and 2, and ran the teacher and student websites during the Module 1 pilot. I participated in multiple teacher training

sessions, as well as running one at a remote school. I also visited local schools to teach lessons, assist students during the lessons, answer teachers' questions, install applications and download assignments onto lab computers, and serve as tech support. I lead student interviews and focus groups at multiple schools [11]. Finally, I gave talks on Depict projects at multiple conferences, including ICER and SIGCSE. As a member of Depict, I contributed to numerous papers referenced in this thesis [9, 10, 11, 15, 16, 20, 21, 22].

H. Future Work

The completed fifth grade module may be piloted similarly to the fourth grade module. Although an outline for the sixth grade module exists, developing and implementing it are also left as future work. While piloting the Digital Storytelling module, Depict researchers found that English Language Learners in particularly struggled with the literacy requirements of the programming language and environment. Depict plans to study the experiences of English Language Learners in computer science classes, and how to improve programming curricula to better fit their needs.

V. The Octopi Application Suite

A. Overview

The Octopi application suite is a set of three block-based programming environments developed for the first year of KELP-CS. All three applications are all based on Scratch, a block-based programming language and environment developed at MIT [37], and modified to better fit the age group and research goals of KELP-CS. OctopiDeveloper allows teachers and curriculum developers to create starting files for student assignments [32]. OctopiStudent is the most similar to Scratch, and is used by students and teachers to view and add to the starting files [34]. The third application, OctopiResearcher, provides extra options for researchers to quickly look through projects from OctopiStudent [33].

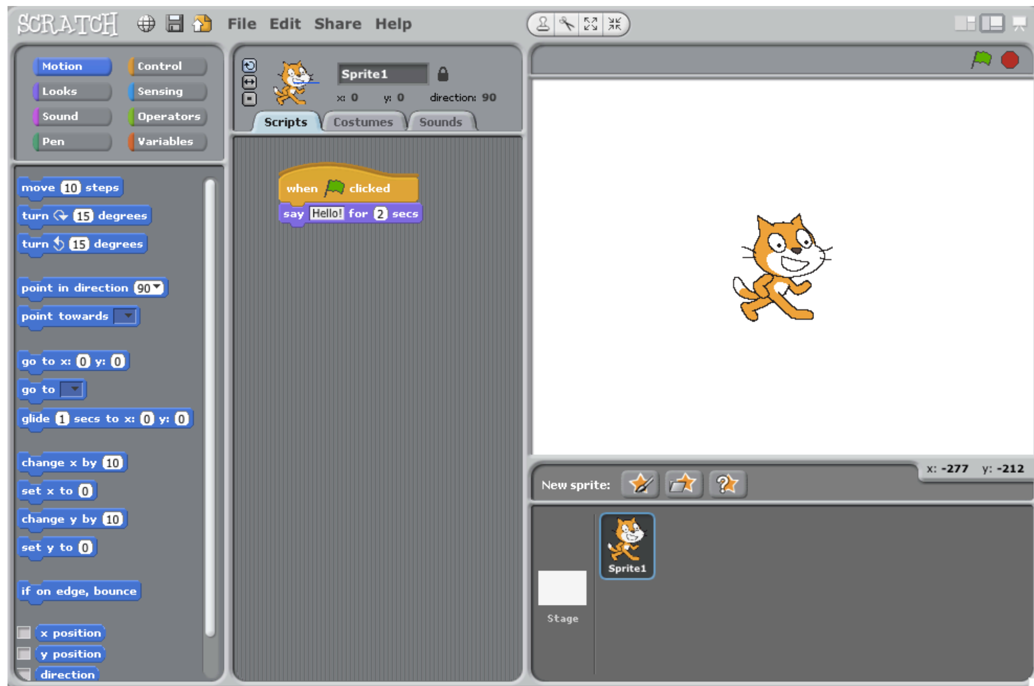

Figure 7. A project in Scratch version 1.4

B. Guiding Research Questions

I created the Octopi applications to address several research questions. Most broadly, the Octopi applications attempt to determine: *how can the programming environment support students, teachers, and curriculum developers?* OctopiStudent and OctopiDeveloper, described in Section E, look more specifically at *how much of a language should be visible to students while they learn programming?* And *how does the language (the blocks) impact student learning and success?* We changed several blocks during the KELP-CS pilot, particularly ones that included math above the fourth grade level. Then, we observed changes in student success in assignments that used these types of blocks [2]. Finally, we developed OctopiResearcher and KelpPlugin while looking at *what types of programs can we use to assess student learning in block-based programming environments?*

- How can the programming environment support students, teachers, and curriculum developers?
- How much of a language should be visible to students while they learn programming?
- How does the language (the blocks) impact student learning and success?
- What types of programs can we use to assess student learning in block-based programming environments?

C. Background & Motivation

Scratch version 1.4[1] (Figure 7), and the programming environments in the Octopi suite, are desktop applications designed for novice programmers. The environment provides a set of programming command blocks that the user drags and snaps together like puzzle pieces to create scripts that control sprites, 2D images (usually characters) that are shown to the right of the scripting area. Scripts are event-driven, and run when the user presses a key,

[1] The current version of Scratch, Scratch 2.0, is a web application

clicks on the sprite that owns the script, clicks on the green flag in the interface, or when another sprite broadcasts the corresponding message. Scratch is a programming playground designed for children and adults to experiment with code, and can be used to make anything from simple "Hello world" programs to complicated games or interactive stories. Scratch also has a booming online community where users can share their Scratch programs or play programs created by other users [40].

Scratch 1.4 was used for the Animal Tlatoque summer camp. During the camp, we found that, although Scratch is intended for ages 8-16, younger students struggled with aspects of the interface. Additionally, it didn't have all the components required for the KELP-CS curriculum. In KELP-CS, students use starting files, or files created by the KELP-CS curriculum designers with example scripts for the students to build on and other, more complicated scripts that would run in the background but that wouldn't be edited by the students. Additionally, we theorized that complicated scripts or block options would confuse or overwhelm students who were just starting out in the development environment. For KELP-CS, we wanted a programming environment similar to Scratch that was modified to better fit the needs of our curriculum.

D. Implementation

Each Octopi application is a separate modification of Scratch version 1.4. The open source version of Scratch contains all of the programming environment functionality of Scratch, but does not include the networking utilities used to connect with the online Scratch community. Scratch is a Squeak Smalltalk application, a unique language that has its roots in computer science education as well. Smalltalk is not frequently used today and works differently than most commonly used languages today, so I'll provide a short overview here.

Smalltalk was created by Alan Kay, who also made significant contributions to computer science education and today's modern operating systems [25]. Smalltalk is all about objects: like in Scratch, everything is an object, and objects send messages to each other in order to accomplish things. There are multiple varieties of Smalltalk but Scratch is a Squeak Smalltalk application. Squeak is a version of Smalltalk created by Dan Ingalls and Alan Kay for Disney [25]. Programming in traditional Squeak[2] requires three components: a virtual machine, an image file, and a change file. These components make up the entire language and your own codebase; rather than creating an application written in a language, when programming in Squeak you download a Squeak image and an associated change file, and then add features to it to create your application. You open your application with the virtual machine. Each application in the Octopi suite is a modified version of the Scratch image, and includes the Scratch objects and functions as well as the basic Squeak objects and functions.

Scratch is built on an older Squeak image— version 2, while the latest version number is 4.5 [25]. Scratch does not have all the functionality of modern Squeak, and because of the way Smalltalk works, there's no feasible way to "update" Scratch to a newer version of Squeak. This poses a few problems — the older version has less documentation, a less modern programming environment, and is missing some functionality needed for OctopiStudent. Modern Squeak has methods to create, add to, and open zip files. Although parts of these classes are in version 2, it is not fully implemented. The biggest change made to core functionality was to essentially implement this zip class from version 4 inside of the OctopiStudent image (which is built on Squeak version 2.) Unlike other changes

[2] Modern Squeak also has an all-in-one system, but Scratch is implemented in the three-component system

implemented while creating the Octopi applications, this required changing and creating primitives in the language as well as determining the associations and dependencies of the version 4 zip class in order to incrementally add functionality to the OctopiStudent image in the correct order. Other changes made to the Octopi applications were less involved, and more similar to adding classes and methods in other languages.

E. OctopiStudent & OctopiDeveloper

OctopiDeveloper is a block-based programming environment that allows developers to create customized starting files for assignments (Figure 8). These customizations change the features available when the file is opened in OctopiStudent (Figure 9). Both OctopiDeveloper and OctopiStudent are very similar to Scratch, but support new features designed specifically for KELP-CS and similar curricula. In this section, I'll describe the new features and modifications that make OctopiStudent and OctopiDeveloper unique from

Figure 8. The mammals project from the KELP-CS digital storytelling module in OctopiDeveloper

Scratch, and how they relate to the KELP-CS curriculum.

1. Script and sprite options

We designed the Kelp-CS curriculum with short lab sessions in mind; in many schools, classes only have an hour a week in the computer lab. We wanted short, focused projects that resulted in working programs that students could complete in less than an hour. We decided that the best way to do this would be to give students starting files to add to and develop in the lab. These starting files would already have some or all of the sprites students would need to program, as well as some scripts for students to use as examples or just run in the background along with the scripts they wrote themselves. However, we worried that students would change or delete the provided scripts needed to make the programs we gave

Figure 9. The mammals project from the KELP-CS digital storytelling module in OctopiStudent

them run. OctopiDeveloper provides more options for scripts and sprites when developing starting files in a block-based, Scratch-like environment. OctopiDeveloper lets curriculum

designers make scripts “visible” or “hidden”, and sprites “editable”, “locked”, or “hidden”. These options change the availability of these objects when the same file is opened in OctopiStudent.

2. Customizing the language and interface

Curriculum designers can also change how much of the language and interface are available when the project is opened in OctopiStudent. Although all blocks are available by default, blocks or block categories can be hidden in OctopiDeveloper. Hidden blocks can still be used in the project by developers, but they’re a lighter shade than the original block color and if students delete these blocks, there’s no way for them to get these blocks back; for this reason, these blocks are only used in hidden scripts.

Additionally, developers can decide how much of the interface is available to students. We decided to give the option to limit interface abilities (for example, adding or deleting sprites) because we did not want students to break the starting files we gave them (such as by deleting sprites) and because, when we were in the classroom, we found that some of the interface options were distracting or difficult for students to understand. For example, the toolbar in Scratch allows students to change the size of or delete sprites by clicking on a toolbar option and then on a sprite. Fourth graders in particular are at an age where they explore by clicking different parts of the interface almost at random, and many students deleted or changed sprites without intending to. We removed the toolbar completely from the interface (since there were other ways to do all of its commands) and let developers choose whether to allow the following features: add sprites, remove sprites, view the costumes tab, view the sounds tab, add costumes, add sounds, and edit existing costumes.

The default for all of these interface features and blocks is to make them visible and available for students.

3. Separating development and runtime

Graduate student researchers who helped in the KELP-CS pilot classrooms noticed that students did not see a distinction between development and runtime. The line between development and runtime in Scratch is blurry by design. Sprites are located on the stage next to the scripting area, so users can modify or move them while programming or even while the program is running. Unlike many textual languages, Scratch doesn't need to be compiled. Users can run scripts at any time by triggering events that start scripts or by clicking on the scripts themselves. During the KELP-CS pilot, many students ran scripts by clicking on them rather than triggering events.

The first project, Animal Maze, instructed the students to "pick up the animals with the net". Students interpreted this to mean that the goal of the task was to pick up each animal at some point during the lab. However, the instructions meant to say that students should write a program that will make the net pick up all the animals. Many students instead wrote and deleted short scripts that made the net pick up each animal. Students clicked directly on the script to run it each time, so the initialization scripts that make the previously picked up animals reappear never ran [2]. However, if students had restarted the whole program rather than clicking on the script (as we intended), they would see that the animals they previously picked up were no longer in the net since the script they wrote only picked up the next animal. Additionally, young children have less dexterity and had issues distinguishing between clicking and double clicking on a script, often running it by accident

when they meant to edit or move it instead. We decided to disable "click to run" on scripts to alleviate these issues.

Although the implementation of this decision is simple, its consequences are not. Development and runtime are interesting abstractions, and it's debatable whether or not they should actually be distinct. Bret Victor's work explores this area and its applications in textual programming [44]. Victor theorizes that programming environments should respond immediately to the programmer, and in a sense, get the programmer's ideas on the page as soon as possible. Victor's ideas can be seen in Apple's Playgrounds, a programming environment developed to blur the line between runtime and development in textual programming [24]. However, it's not always clear how to visualize code in the environment, particularly a text-based one. Exploring the boundaries of runtime and development in a block-based programming language such as Scratch is an interesting issue and has potential for future work.

4. Snapshots of student work

During Animal Tlatoque, we collected and analyzed student projects using Hairball. However, final projects don't show how the student's project evolved over time, or the different methods or problem solving techniques students may have tried out and then deleted. Additionally, students often played in the programming environment after finishing the assignment without starting a new file, so the file they turned in might not actually be the "finished" project. We wanted to capture the ongoing process of students' work as well as the final projects in LaPlaya. However, writing to the file was sometimes slow, particularly in networked computer labs. As a compromise, OctopiStudent automatically saves a "snapshot", or version, of the project whenever the student clicks on the green flag button

after making at least five changes to the program since the last time it was saved. These snapshots show how the student changes a project throughout the class, since most assignments require students to click on the green flag to start the program. OctopiStudent zips the new “snapshot” of the file with any other snapshots of projects with the same name created in the last hour in that folder. When students finish working on their project, they now have two files: an Octopi file that they can open and change in any Octopi suite application, and a zip of all the snapshots created while working on the project.

F. OctopiResearcher & Collecting Student Work

We created multiple websites for students and teachers to use with the KELP-CS curriculum. Each school had its own teacher and student webpages for students to download starting files, and for teachers to download solution files and worksheets. At the end of each lab, students uploaded their assignments to an additional website, Octopi Submit, which stored all of the assignments and let students and teachers download students’ submitted work (Figure 10). These projects were than analyzed using OctopiResearcher.

OctopiResearcher is the companion to OctopiDeveloper; it is used by KELP-CS researchers to look through student projects. As described in the previous section, OctopiStudent creates a zip file of the different “snapshots” of the project while the student is working. Scratch-like projects are time consuming to look through; there’s no “quick look” or way to search through a set of projects. OctopiResearcher knows the file structure created by OctopiStudent, and has next and previous buttons so users can move to the next or previous snapshot or project without leaving the application or going through the “File -> Open” menu. We used OctopiResearcher with KelpPlugin to analyze student projects.

G. KelpPlugin & Understanding Students' Ideas about Initialization

KelpPlugin is an automated grading tool similar to Hairball used to assessing student snapshots as well as the final versions of assignments. Hairball and KelpPlugin are both based on Kurt, a Python library for analyzing Scratch projects [27]. Kurt takes a Scratch file and turns it into a dictionary, making it possible to write simple Python scripts to analyze the contents of Scratch projects. An extension made it possible to use Kurt on Octopi files, rather than Scratch files. KelpPlugin, a collection of grading plugins like Hairball, allows developers to look through all of the snapshots of an assignment rather than just the final version.

By using KelpPlugin and OctopiResearcher together, we could study the approaches, or "paths" students take when working on programming assignments and observe the problems students encounter when completing an assignments. Previous work has found that the paths students take to complete programming projects have a much stronger correlation

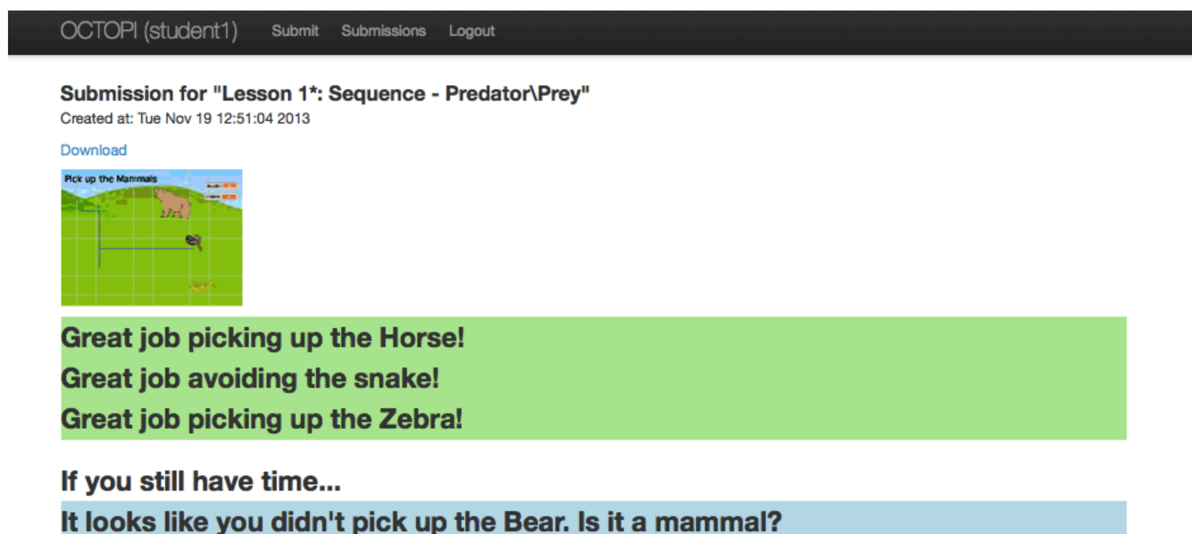

Figure 10. Student view of KelpPlugin feedback for a KELP-CS assignment on the Octopi Submit website

to future success than the final state of the project [36]. By using OctopiResearcher alongside KelpPlugin, we can determine paths that students took and also examine projects to see what these paths represent. We used OctopiResearcher to look at the trends we found with KelpPlugin to better define students' problem-solving approaches. We used this combined approach to study students' ideas about initialization during the KELP-CS pilot [22].

In traditional programming, variables are initialized at the start of the program or before the code segment that uses those variables. The variables are initialized, or set to initial values, in order for them to be useful; it doesn't make sense to use a variable that you're not storing something in. However, the visual nature of Scratch, OctopiStudent, and similar environments makes initialization work differently. The "variables" you're initializing in OctopiStudent are usually aspects of the sprites — their location on the stage; their color, size or other visual attributes. These attributes are never "undefined" in the same way that variables can be in text-based programming; sprites always have a size, a location, etc. However, sprites might move or change color during a program, and if the program doesn't initialize these attributes then they'll still be in that last state the next time the program is run. This makes initialization in Scratch more like "resetting" than actually "initializing". When we analyzed student projects from the initialization assignment from the KELP-CS pilot, we found that students did not understand when sprites needed to be initialized — at the beginning or the end of the program (Figure 11).

We also found that the students who initialized their sprites did it in ways we did not expect. Some attributes, such as location, are affected by multiple types of blocks — blocks that set the attribute to a specific value, blocks that change the attribute by a value, blocks that change or set the value over time, and blocks that change or set the value instantaneously. We were surprised to find that many students initialized attributes with blocks that included timing — for example, *glide _ seconds to x: _ y: _* rather than *go to x: _ y: _*. In traditional programming, initialization is instantaneous and invisible; it's not really something that the user would normally see. However, in OctopiStudent the barriers between development and runtime are blurred, and there's no predefined starting point for programs. We theorized that the nature of the programming environment changed the ways that novice programmers thought about initialization, and that aspects of initialization in text-based languages might not be as relevant in visual languages like Scratch. We made

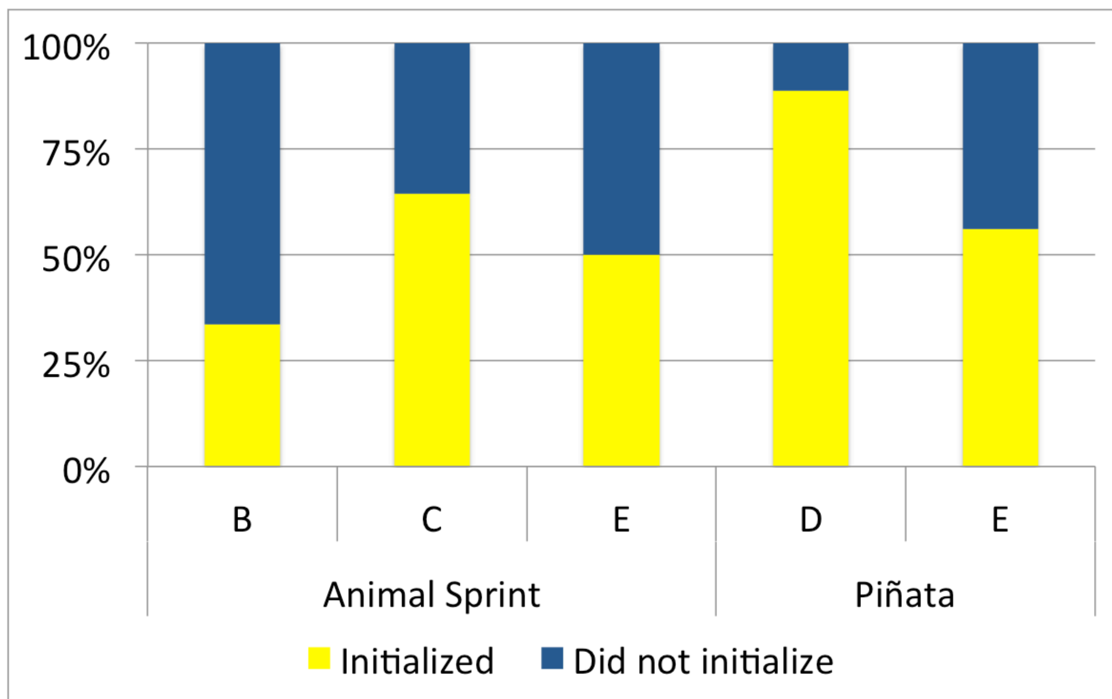

Figure 11. KelpPlugin results of initialization in two KELP-CS initialization projects – Animal Sprint, the original project, and Pinata, the revised project.

several changes based on our research of student views of initialization. We realized that many initialization blocks rely on math concepts that are taught after fourth grade, such as x,y coordinates, negative numbers, and percentages, so we modified and added new blocks to give younger students other ways of initializing. We also changed the way students run starting files — in the pilot, most assignments were started by clicking on the green flag, but in the new version of the module, we put a stronger emphasis on starting programs the same way every time. We added another button next to the green flag button, the "get ready" button. The "get ready" button runs "get ready" scripts, giving students a predefined place to initialize their sprites.

H. Lessons Learned & Future Work

The Octopi application suite changed over the course of the KELP-CS pilot in response to feedback from teachers, students, and graduate student researchers in the classroom. However, by the end of the pilot it was apparent that there were fundamental issues with using OctopiStudent in the classrooms that could not be addressed through minor changes.

The frequent changes to the application that resulted from the design-based structure of the research project were time consuming and frustrating, since each update meant that the application needed to be downloaded and reinstalled. The variety of lab setups meant that no schools updated the same way. One school had a computer lab manager who could install the updates on the networked computers. Another had a remote district administrator in charge of installing updates on the networked computers. A school near UCSB did not have networked computers or a computer lab manager, so updates had to be installed on each individual machine any time we changed the program.

The starting files also posed problems in the classrooms. Each programming assignment required a starting file for students to modify and add to. These files were packaged separately from the application, so someone (such as a computer lab instructor or me) had to download all the files and put them in an easily accessible area on the computers, or the students had to download that assignment's file in the beginning of each lab. However, fourth graders do not have a good understanding of file systems. Students couldn't find files in the "File-> Open" window that they had placed on the desktop. In classes that downloaded files at the start of the lab, students often didn't understand how to tell if a file had been downloaded and where it was stored when it did, so some students had dozens of copies of the same file in their downloads folder. Even in the best case, where the files were already downloaded and placed in the "Octopi projects" folder that the application goes to first when "File -> Open" is selected, the students struggled to select the right file since the names were complicated and their reading skills weren't always at grade level. Like the application, the starting files changed throughout the course of the pilot, so if the files had all been downloaded at the start of the pilot, someone would have to replace them with the new versions. Schools started at different times of the year, so different schools would end up with different versions of each project. This meant that each school needed its own teacher and student page, which would be updated and maintained separately.

Collected student work was challenging for similar reasons. At the end of each lab period, students uploaded the zip file for the assignment to the Octopi submit website. Students struggled to navigate to the website, log in, and select the correct assignment name to upload to. Students also had problems navigating in the upload dialog. Some schools had multiple classes participating in the KELP-CS pilot, and students would accidentally upload

another student's assignment from the shared lab computer. OctopiStudent creates two files: filename.oct and filename.octx. The former is the project file that is read by the Octopi application suite, and the latter is the zip file of snapshots created by OctopiStudent. Students were expected to upload the octx file, not the oct file, but many uploaded the oct file by accident so we did not have snapshots for all the students. For the first half of the pilot year, students had to manually select which project they were uploading, and the original file name of the uploaded file would be replaced. However, students often skipped this step, leaving it set to the first name in the project drop down so the projects had to be manually sorted into the proper categories.

In response to these issues, we decided to move completely online. An online programming environment would allow us to push updates easily and more frequently. Students would not have to download or upload any files. The new online version of Scratch did not yet have an open source version available, so instead the successor to OctopiStudent is based on the open source programming environment Snap!, created by UC Berkeley [42]. Snap! is a Javascript application based on Berkeley's previous block-based environment, a Scratch modification called BYOB. Snap! is inspired by Scratch that extends its functionality to make it better for more advanced programmers. LaPlaya, the new KELP-CS programming environment, took it in the other direction, simplifying Snap! and implementing many of the features we created in the Octopi application suite.

VI. LaPlaya & Octopi

A. Overview

LaPlaya is an online programming environment created for the second year of running the KELP-CS curriculum in classrooms. LaPlaya is a modification of Snap! by UC Berkeley [42] and inspired by Scratch by MIT [40]. LaPlaya runs inside of Octopi, a web application for hosting block-based programming curricula. In designing LaPlaya, we kept many of the features created for the Octopi application suite. Developers can create starting files with hidden or sample scripts and customized interface and language options in developer mode, which are modified by students in student mode. Starting files are organized by the curriculum developers in Octopi, allowing them to open projects directly from the website. Rather than saving multiple "snapshots", LaPlaya logs all changes made while students are working on the project, allowing researchers to get a better idea of how the projects were changed over time. More detailed information about LaPlaya and Octopi can be found in [21] and [19], respectively, but I'll summarize the design and implementation of LaPlaya and Octopi here.

B. Guiding Research Questions

Our work with LaPlaya continued addressing the research questions we looked at with the Octopi applications. In addition, we developed *design principles for block-based programming environments for fourth through sixth grade*, which are described in the next section. We also addressed the research question, *how do students engage with block-based programming environments?* Section E describes our research on how students read and engage with block-based programs.

- Design principles for block-based programming environments for fourth through sixth grade
- How do students engage with block-based programming environments?

C. Methods & Findings

During the KELP-CS pilot, we found that students struggled with aspects of the programming environment (OctopiStudent) interface and language [21]. We piloted the curriculum in fifteen 4th – 6th grade classrooms at five schools across California. We refer to these schools as A, B, C, D and E, with A being the first school trial and E being the last. In schools B and E, we collected only student projects. In schools A, C, and D, we observed instruction and interviewed students. The schools had varying numbers of classrooms, grades participating, start dates, and order of projects. During this pilot, we found that students struggled with some math concepts in the language and with parts of the interface. For this analysis, we focus on schools A and B, which used a version of OctopiStudent that was very similar to Scratch. I made multiple changes to the programming environment based on our findings in these classrooms before the later schools used it.

1. The interface

Students in the KELP-CS pilot struggled with aspects of the interface that were distracting or made it too easy to delete parts of the starting files without clear ways to recover. Some students deleted the scripts or sprites in the starting file, which are difficult or even impossible in some cases to add back later. In schools A and B (142 students, 516 projects), we found that students deleted provided sprites in at least 4.5% of projects, and deleted provided scripts in at least 9.6% of projects. These numbers might be lower than the actual amount – students who deleted parts of the starting files often restarted the

assignment with a new, unchanged version of the starting file, and our numbers do not include those students.

Some aspects of the programming environment were distracting for students. For example, a “surprise sprite” button adds a new random sprite to the stage every time you click on it. These buttons can be distracting for younger age groups. As an example, one student in our study added 34 sprites to one project. In schools A and B, students added unnecessary sprites in 10.1% of projects. Although it is important to allow students to explore and find different ways of solving the problem, “instant gratification” buttons can switch from being vehicles for exploration to distractions that spread through the computer lab as students observe their peers’ computers.

2. The language

Many of the blocks used to move the sprites and change their appearances rely on math concepts above the fourth grade level. Throughout the KELP-CS pilot, I made changes to the OctopiStudent language based on findings from schools A and B. Other graduate student researchers and I observed that students struggled with Cartesian coordinates, negative numbers, and percentages, which are all above the fourth grade level, as well as decimals, which are taught during fourth grade. Since KELP-CS could be taught at any time during the fourth grade school year, its content cannot rely on math content at grade level, as well as above it. Next, I’ll describe how these math concepts are used in Scratch, as well as the original version of OctopiStudent.

Cartesian coordinates are used to position the sprites on the stage. Setting a sprite to an absolute location is done with a *go to (sprite or mouse pointer)* block or a *go to x: y: (x and y coordinates)* block. Alternatively, students can drag the sprite to the location where they

want it to go before selecting the block, as OctopiStudent and Scratch auto-populate the x and y values based on the sprite's location. This is not a great solution however, since it requires a lot of repetition and memorization. Negative numbers are also used in these blocks, since the coordinate plane in OctopiStudent was originally centered on center of the stage (I later moved this to the lower left corner to get rid of negative coordinates). Additionally, negative numbers are used in the *change (something) by X* blocks to reduce the size, volume, x or y coordinate, and variable value. Finally, percentages are used to control the size and volume. Not only must students understand percentage parts of a whole, they also need to understand what 100% means for that variable; for example, the size percentages are of the original picture size, which the students are unlikely to know.

D. Motivation & Design Principles

LaPlaya is based on the Octopi application suite, which, as described in the previous section, is a set of Scratch modifications tailored to fit the KELP-CS curriculum that failed to fit the needs of the classroom environment. Some of the biggest implementation concerns were that Octopi was a desktop application that needed to be installed and required students or teachers to download and upload assignments from a separate website. Additionally, Octopi did not run on iPads, which some classrooms wanted to be able to use. To address these concerns, we moved our programming environment completely online. LaPlaya is a Javascript application that runs inside of Octopi, a Ruby on Rails web application that manages student, teacher, and researcher accounts and access to the KELP-CS assignments. In Octopi, students can click on an assignment to open it in LaPlaya and then just save and close at the end of the lab, rather than downloading the starting file and then uploading the result later on. We also made several changes to the programming environment itself.

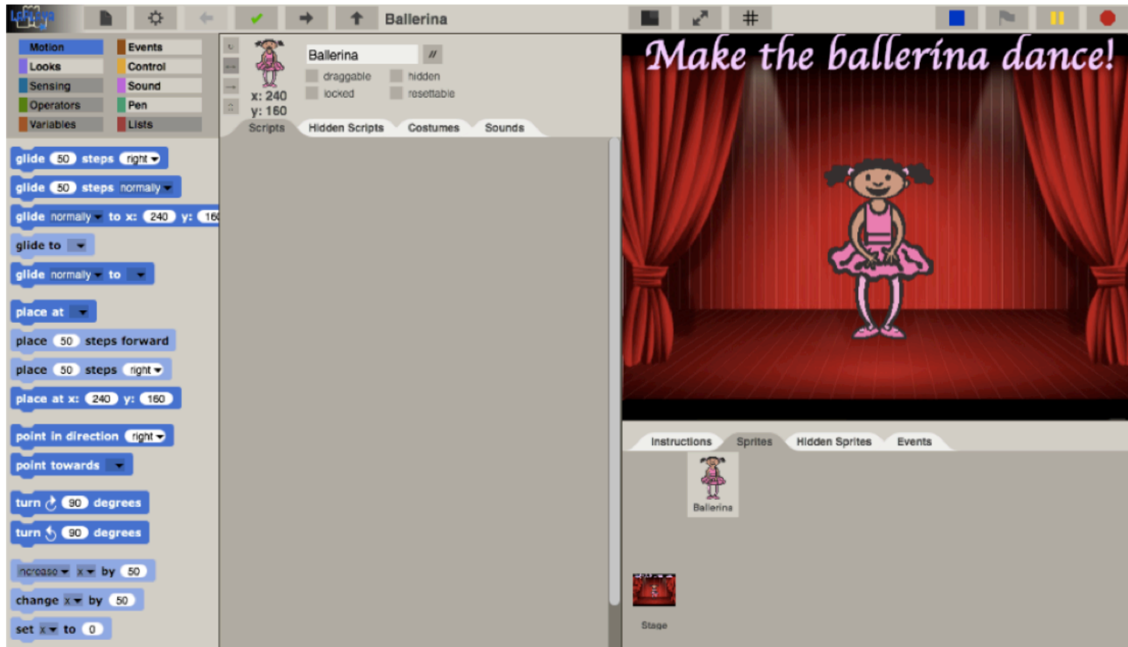

Figure 12. The ballerina project from the animation assignment from the revised KELP-CS digital Storytelling module, open in LaPlaya in developer mode

Our findings from the KELP-CS pilot — as well as our experiences with other block-based languages — informed our design principles for LaPlaya. Scratch, ScratchJr, and LaPlaya are all block-based environments designed for or used in curricula for children. However, they are all structured differently. Scratch was designed to be an open-ended playground for a wide age group of programmers to explore. ScratchJr, which is based on Scratch, has a similar design but is targeted for a younger age group, so it has fewer blocks and uses symbols instead of words. Blockly was designed to be a flexible block-based environment for multiple block-based languages. Unlike Scratch or ScratchJr, Blockly does not have an editor for working from blank files — instead, each Blockly project uses a starting file created in Javascript. The following design principles build off of aspects of each of these languages.

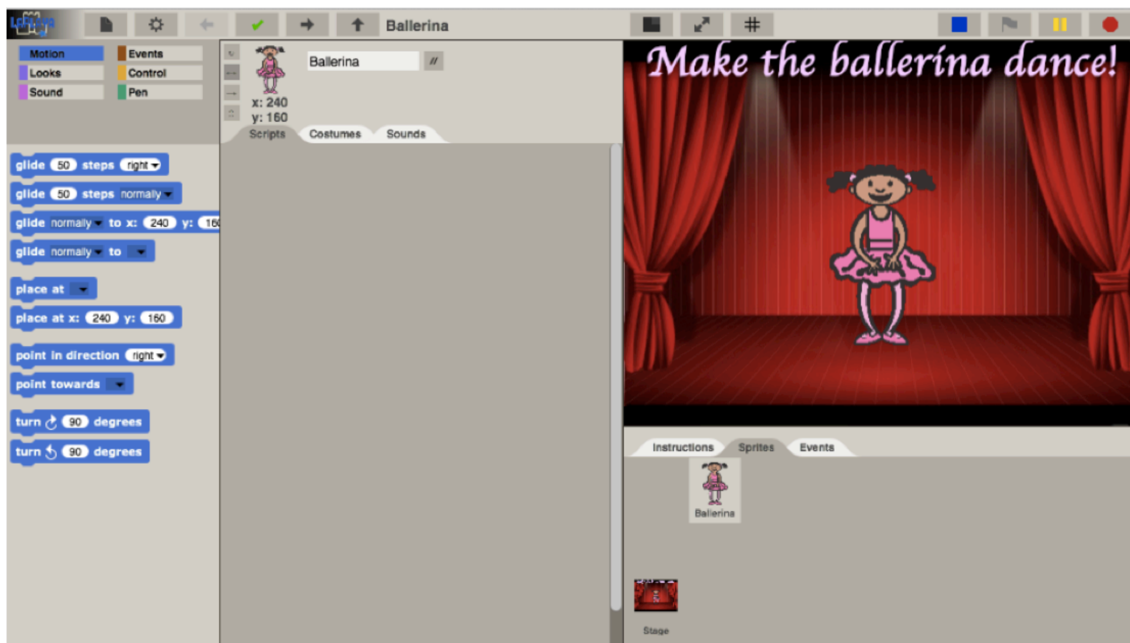

Figure 13. The ballerina project from the animation assignment from the revised KELP-CS digital Storytelling module, open in LaPlaya in student mode

1. Support multiple types of tasks

The new version of module 1 has two types of programming assignments — short, targeted tasks that use prepopulated starting files, and longer, open-ended projects where students can explore the LaPlaya language. Scratch and ScratchJr both provide playground-like environments for open-ended assignments, but don't allow for interface or language customization. Blockly is a great tool for creating customized starting files, but it lacks the type of environment needed for open-ended exploration. LaPlaya allows both of these types of projects by allowing developers to optionally hide aspects of the language and interface in developer mode, similarly to OctopiDeveloper (Figures 12 & 13).

2. Require only grade- and age- appropriate content

We also made several changes to the blocks in the LaPlaya language. During the pilot, we found that students struggled with Scratch blocks that required math knowledge above

their grade level, such as using percentages and coordinates [21, 16]. Some of these blocks required only small changes to make them more suitable for 4th grade. For example, many attributes can be changed with *change _ by __* blocks. In LaPlaya, these blocks are rewritten to *increase/decrease _ by _* with a drop down where users select increase or decrease, eliminating the need for negative numbers.

3. Include an age-appropriate interface

Students who participated in the KELP-CS pilot ran into difficulties with some parts of the interface. For example, it was too easy to delete sprites or scripts and there wasn't a good way to get them back. Other aspects of the interface were distracting for students, such as the "add sprite" button. Some students added dozens of sprites but then never wrote scripts for them — they just liked the instantaneous feedback delivered by the "add sprite" button. Similarly to OctopiDeveloper, LaPlaya developers can hide these aspects of the interface in developer mode.

4. Support project developers

The previous two design principles both support customization, a major component of Blockly. However, Blockly customization is done in Javascript. We wanted non-programmers to be able to create starting files for the KELP-CS curriculum. LaPlaya, like OctopiDeveloper, lets curriculum designers create starting files in a drag and drop environment.

E. Implementation

LaPlaya is based on Snap!, an open source Javascript programming environment by UC Berkeley. The Snap! source code is based on Morphic, a web-GUI inspired by Squeak [AN].

Morphic defines a set of Morph classes that all inherit from a base Morph, similar to the class structure in Squeak Smalltalk. All of the Snap! classes and functionality are built from these Morph classes rather than built-in Javascript classes. This design decision makes adding to and modifying Snap! more like programming in Squeak than Javascript. It also makes for a larger codebase, making it harder to both accelerate loading and running scripts in LaPlaya. Although technically able to run on iPads and less computationally intensive devices, LaPlaya takes a long time to load and larger projects run too slowly to be very useful in a classroom setting.

F. Reading LaPlaya Programs

During the second year of KELP-CS, we held focus groups with students participating in the curriculum to learn more about how 4th graders read programs in LaPlaya [11]. When reading a block-based program someone else wrote, there is more content to look at than simply the words on the blocks. Block-based programs are written and read in very different environments than text-based languages. Most are focused on visual characters or scenes, and as such are organized differently than text-based environments. Block-based scripts usually appear alongside characters or other pictures that can give contextual clues to the program's content. We wanted to know more about how children read and understand code in this type of environment.

In the context of block-based programming, affordances are objects that have possibilities for action. Visual cues, such as block color or the location of a sprite on the stage, provide information about the possible actions. In this study, we asked the question: what perceptible, hidden, and false affordances of a block-based programming environment do students use to read block-based programs? We interviewed pairs of students at two local

schools, and asked them to read and make predictions about several LaPlaya projects. These focus groups showed the types of affordances that students used to make predictions about what LaPlaya programs do. Affordances are categorized by their intended use and how students actually use them. Affordances that the designers intended to be meaningful but are not used by the students are called hidden affordances. Conversely, affordances that students use that the designers did not intend to be meaningful are called false affordances. For example, many students made predictions about the first project based on what the sprites looked like. They used these false affordances to tell stories about what they thought the sprites would do, based on their prior knowledge of those types of animals.

The first project has a desert scene with three animals: a bat, a unicorn, and a dragon. Each animal does something different, and each animal runs on a different event. During the focus groups, the interviewer asked questions about the first project before running the program; for example, "What do you think would happen if you ran the program?" or "What do you think the bat would do?" We coded students' responses to these questions by the type of affordance and what visual cues (such as the position of the sprites or the wording of the blocks) the students used. This coding scheme was then incorporated into a table of visual cues in LaPlaya, and how students use them (Figure 14).

Students used many visual cues we had intended for them to use, but also employed visual cues that we did not expect. Students use many attributes of the programming environment that designers of block-based programming environments might not have meant for them to use. For example, some students in the focus groups predicted the bat would fly down because of its position at the top of the stage. Reading programs is a skill,

like reading comprehension, that should be taught as a part of a computer science curriculum rather than expecting that students will already know how to do it.

G. Future Work

Our findings from the second year of KELP-CS could lead to new design principles, or new ways to implementing them. LaPlaya is one implementation, but it has obvious technical flaws — such as how long it takes for projects to load — that could be fixed by a reimplementation without the dependency on the morph class. Another approach could be based on Blockly. Since Blockly was created for flexible block-based languages, creating a

Visual Cues in LaPlaya			What Students Were Predicting			
			Block	Single Script	Multiple Scripts	Output
Blocks	Word choice	Prior experience with word	X			X
		Word's everyday meaning				X
	Block layout	Block argument	X			X
		Color				
		Shape				
	Same block, other script		X			X
Scripts	Ordering of blocks within scripts		X			X
	(Layout of scripts)					X
	Other blocks in script					
	Other scripts					X
Stage	(Sprites on the stage)	(Physical characteristics)	X			X
		(Orientation)				X
		(Stage position)				X
	(Background)					
Interface	(Sprite corral)		X			X
	(Costume tab)					X
	(Costume images)		X			X
	(Instruction tab)					X

Figure 14. How students used visual cues to predict aspects of a LaPlaya program. Parentheses distinguish visual cues that were categorically false affordances. "X" signifies that students used a visual cue when making predictions about the first project.

LaPlaya-like language would not be very time consuming. A Blockly project creator application could be created for developers to build their own assignments in a drag and drop environment similar to developer mode in LaPlaya.

VII. Conclusion

Block-based languages such as Scratch and LaPlaya were created to give novice programmers an environment as welcoming and intuitive as a box of Legos. However, this work has shown many of the potential problems with using block-based environments in elementary school classrooms. Environments with too many options can distract students from programming. Traditional languages often contain math or language beyond the scope of the elementary school curriculum. While teachers can benefit from programming environments that do not require specialize knowledge, those same environments can exclude them from the learning process. Additionally, younger children have less physical dexterity than adults and often struggle with typing and using the mouse — an issue for block-based languages that rely on drag and drop. Here, I offer some suggestions for new design principles addressing these issues to address my larger research question, *How can programming environments better meet the needs of upper elementary classes learning computer science?*

Streamline the environment and remove distractions: Consider removing file menus and limiting non-programming aspects of the environment, like we did with OctopiStudent and LaPlaya. However, ScratchJr designers found that even in a simplified programming environment, students spent a lot of time drawing rather than coding [12]. Another solution is to make programming itself more engaging, so students are less likely to be distracted by "instant gratification" buttons or the paint editor. For example, Playgrounds gives programmers "instant gratification" by showing the result of each line of code while the programmer is typing [24].

Support teachers as well as project developers: Online communities like ScratchEd can engage teachers and give them resources for teaching a new subject. Give teachers platforms to create their own projects, like “developer mode” in LaPlaya. Octopi provides grading tools, lesson plans, and sample LaPlaya project solutions, but it could do more to educate and prepare teachers before the lesson starts. How could a programming environment engage teachers as well as students? A new programming environment or interactive tutorial could show teachers not only possible solutions to assignments but also possible problems, guiding teachers though parts of the environment and language that students might find challenging.

Consider alternatives to drag and drop with mice: Develop programming environments for touch screen devices, like ScratchJr. Keyboard-based drag and drop accessibility options, such as sticky keys, can work as a quick fix for online environments like LaPlaya. Drag and drop is arguably not a crucial aspect of block-based programming, so a redesigned interface could give students other ways of selecting blocks and their new locations. Alternatively, developers can use tangible blocks that can be read by a webcam such a Tern [43] to engage students in a different way.

Computer science education is a growing field, and there is a need for IDE developers focusing on language requirements and environment needs of elementary school programmers and their teachers. New programming environments for children have the potential to change the way future generations of computer scientists think about programming. Expanding the options for entry-level programming languages may also encourage more children to be interested in programming, bringing greater diversity to the field as a whole.

References

1. Blockly. Retrieved April 14, 2015, from https://developers.google.com/blockly/

2. Boe, B. A. (2014). Enabling wide-scale computer science education through improved automated assessment tools (Doctoral dissertation). Available from ProQuest Dissertations & Theses A&I. (Accession Order No. AAT 3645611).

3. Boe, B., Hill,, C., Len, M., Dreschler, G., Conrad, P., & Franklin, D. (2013). Hairball: Lint-inspired Static Analysis of Scratch Projects. In *Proceedings of the 45th Technical Symposium on Computer Science Education (SIGCSE '13)*. Denver, CO: ACM.

4. Bruckman, A., Bandlow, A., & Forte, A. 2012. HCI for kids. In J. A. Jacko (Ed.) The human-computer interaction handbook: fundamentals, evolving technologies, and emerging applications (841 – 862). Boca Raton, FL: Taylor & Francis Group, LLC.

5. code.org: Anybody can learn. Retrieved April 14, 2015, from http://code.org

6. Computer Science Education Week. Retrieved April 14, 2015, from http://csedweek.org/

7. CSTA, Computer Science K-8: Building a Strong Foundation. http://csta.acm.org/Curriculum/sub/CSK8.html

8. Duncan, C., Bell, T., & Tanimoto, S. (2014). Should your 8-year-old learn coding? In *Proceedings of the Ninth Workshop in Primary and Secondary Computing Education (WiPSCE '14)* (pp. 60-69). New York, NY, USA: ACM.

9. Dwyer, H., Boe, B., Hill, C., Franklin, D., & Harlow, D. (2013). Computational Thinking for Physics: Programming Models of Physics Phenomenon in Elementary School. In *Proceedings of the 2013 Physics Education Research Conference (PERC '13)*. Melville, NY: AIP Conference Proceedings.

10. Dwyer, H., Hill, C., Carpenter, S., Harlow,, D., & Franklin, D. (2014). Identifying Elementary Students' Pre-Instructional Ability to Develop Algorithms and Step-by-Step Instructions. In *Proceedings of the 45th Technical Symposium on Computer Science Education (SIGCSE '14)*. Atlanta, GA: ACM.

11. Dwyer, H. A., Hill, C., Hansen, A., Iveland, A, Franklin, D & Harlow, D. (Submitted for review). Elementary School Students Reading Block-Based Programs: Predictions, Visual Cues, and Affordances. ICER, 2015.

12. Flannery, L. P., Silverman, B., Kazakoff, E. R., Bers, M. U., Bontá, P., & Resnick, M. (2013). Designing scratchjr: Support for early childhood learning through

computer programming. In Proceedings of the 12th International Conference on Interaction Design and Children (pp. 1-10). New York, NY: ACM.

13. Franklin, D., Conrad, P., Aldana, G. & Hough, S. (2011). Animal tlatoque: attracting middle school students to computing through culturally-relevant themes. In Proceedings of the 42th Technical Symposium on Computer Science Education (SIGCSE ’11). Dallas, Texas: ACM.

14. Franklin, D., Conrad, P., Boe, B., Nilsen, K., Hill, C., Len, M., Dreschler, G., Aldana, G., et al. Assessment of Computer Science Learning in a Scratch-Based Outreach Program. In Proceedings of the 45th Technical Symposium on Computer Science Education (SIGCSE ’13). Denver, CO: ACM.

15. Franklin, D., Harlow, D., Dwyer, H., Henkens, J., Hill, C., Iveland, A., Killian, A. & Development Staff. (2014). Kids Enjoying Learning Programing (KELP-CS) — Module 1 Digital Storytelling. A computer science curriculum for elementary school students. Available at www.discover.cs.ucsb.edu/kelpcs/educators

16. Franklin, D., Hill, C., Dwyer, H., Martinez, T., Iveland, A., Killian, A., & Harlow, D. (2015). Getting started in teaching and researching computer science in the elementary classroom. In Proceedings of the 46th Technical Symposium on Computer Science Education (SIGCSE ’15). Kansas City, MO: ACM.

17. Gelderblom, H., & Kotze, P. 2009. Ten design lessons from literature on child development and children’s use of technology. In IDC 2009.

18. Halgren, S., et al. 1995. Amazing AnimationTM: Movie making for kids design briefing. In SIGCHI ’95.

19. Henkens, Johan. (2014). Octopi, a Scalable Web Application for Online Computer Science Curricula Using a Block-based Language (Unpublished master project). University of California, Santa Barbara.

20. Hill, C. (2014). Computational Thinking Curriculum Development for Upper Elementary School Classes. In Proceedings of the 10th annual conference on International computing education research (ICER ’14). Glasgow, UK: ACM.

21. Hill, C., Dwyer, H. A., Martinez, T., Harlow, D., & Franklin, D. (2015). Floors and flexibility: Designing a programming environment for 4th - 6th grade classrooms. In Proceedings of the 46th Technical Symposium on Computer Science Education (SIGCSE ’15). Kansas City, MO: ACM.

22. Hill, C., Dwyer, H., Iveland, A., Kilian, A., Martinez, T., Harlow, D. & Franklin, D. (2014) The complexity of initializing Scratch programs: A design-based research study with upper elementary school students. TS. Collection of Depict research group, UC Santa Barbara.

23. Hopscotch - Learn to program. Make awesome things. Retrieved April 14, 2015, from https://www.gethopscotch.com

24. Introducing Swift. Retrieved April 14, 2015, from https://developer.apple.com/swift/

25. Introduction to Squeak Smalltalk. Retrieved April 14, 2015, from http://www.cosc.canterbury.ac.nz/wolfgang.kreutzer/cosc205/smalltalk1.html

26. Kazakoff, E. R. Cats in Space, Pigs that Race: Does self- regulation play a role when kindergartners learn to code? (Unpublished doctoral dissertation). Tufts University, Massachusetts, 2014.

27. Kurt [Computer Software] (2014). Retrieved April 14, 2015, from https://github.com/blob8108/kurt.

28. Lewis, C. M. 2010. How programming environment shapes perception, learning and goals: Logo vs. scratch. In SIGCSE '10.

29. Lightbot - Solve Puzzles using Programming Logic. (n.d.). Retrieved April 14, 2015, from http://lightbot.com

30. NGSS Lead States. (2013). Next Generation Science Standards: For States, By States. Washington, DC: The National Academies Press.

31. Octopi: A platform for learning with LaPlaya. (2015). Retrieved April 14, 2015, from https://octopi.herokuapp.com

32. OctopiDeveloper [Computer Software]. (2014). Retrieved April 14, 2015, from www.charlottehill.com/octopideveloper.zip

33. OctopiResearcher [Computer Software]. (2014). Retrieved April 14, 2015, from www.charlottehill.com/octopiresearcher.zip

34. OctopiStudent [Computer Software]. (2014). Retrieved April 14, 2015, from www.charlottehill.com/octopistudent.zip

35. Papert, S. (1980). Mindstorms: children, computers, and powerful ideas. New York, NY: Basic Books, Inc.

36. Piech et al. Modeling how students learn to program. In *SIGCSE*, pages 153–160, 2012.

37. Resnick, M., Maloney, J., Monroy-Hernández, A., Rusk, N., Eastmond, E., Brennan, K., Millner, A., Rosenbaum, E., Silver, J., Silverman, B., & Kafai, Y. (2009). Scratch: Programming for all. *Communications of the ACM, 52*(11), 60-67

38. Santrock, J. W. Child Development. McGraw Hill, Boston, 2004.

39. Seiter, L. & Foreman, B. (2013). Modeling the learning progressions of computational thinking of primary grade students. In *Proceedings of the 9th.Annual International ACM Conference on International Computing Education Research (ICER '13).* .New York, NY: ACM.

40. Scratch - Imagine, Program, Share. (n.d.). Retrieved February 9, 2015, from http://scratch.mit.edu/

41. ScratchEd. (2014). Scratch Curriculum Guide. Retrieved February 10, 2015 from http://scratched.gse.harvard.edu/resources

42. Snap! (Build Your Own Blocks) 4.0. (n.d.). Retrieved April 14, 2015, from https://snap.berkeley.edu

43. Tern — Tangible programming. Retrieved April 14, 2015, from http://hci.cs.tufts.edu/tern/

44. Victor, Bret. (2012) Learnable Programming. Retrieved April 14, 2015, from http://worrydream.com/#!/LearnableProgramming