A Qualitative Study of Animation Programming in the Wild

Aniket Dahotre
Oregon State University
1148 Kelley Engineering Center
Corvallis, OR 97330
dahotrea@onid.orst.edu

Yan Zhang
Oregon State University
1148 Kelley Engineering Center
Corvallis, OR 97330
zhangy3@onid.orst.edu

Christopher Scaffidi
Oregon State University
1148 Kelley Engineering Center
Corvallis, OR 97330
cscaffid@eecs.oregonstate.edu

ABSTRACT
Scratch is the latest iteration in a series of animation tools aimed at teaching programming skills. Scratch, in particular, aims not only to teach technical skills, but also skills related to collaboration and code reuse. In order to assess the strengths and weaknesses of Scratch relative to these goals, we have performed an empirical field study of Scratch animations and associated user comments from the online animation repository. Overall, we found that Scratch represents substantial progress toward its designers' goals, though we also identified several opportunities for significant improvement. In particular, many Scratch programs revealed significant technical mastery of the programming environment by programmers, and some animations even demonstrated design patterns. On the other hand, while the Scratch repository has successfully served as a supportive environment for generating constructive feedback among users, we did not find any occasions within our sample where this interaction led to online collaboration. In addition, we found low levels of code reuse, in terms of both frequency and success. Based on these results, we identify implications for improving the design of animation tools, for using these tools to teach programming skills, and for fostering successful collaboration and code reuse among end-user programmers.

Categories and Subject Descriptors
K.3.1 [**Computers and Education**]: Computer Uses in Education – *Collaborative learning.*

General Terms
Design, Experimentation.

Keywords
Animation, end-user programming, collaboration, repositories.

1. INTRODUCTION
The world is rapidly shifting from one where people are passive consumers of digital media to one in which ordinary people can contribute new media. This evolution began with simple chat forums where people could contribute textual content, then progressed to online communities like YouTube where people could exchange images, movies and other multimedia artifacts.

The evolution continues and now encompasses online communities where people can share interactive programs that they have created. One of these is MIT's Scratch environment, which claims to be "the YouTube of interactive media" [19]. With the Scratch programming tool, people can create "video games, interactive newsletters, science simulations" and a host of other interactive animated programs [19]. They can then publish their work in an online repository, where other people can see it, comment on it, be inspired by it, download it, and "remix" it (Scratch parlance for whitebox reuse).

Scratch is swiftly attracting new users, whose count now stands at nearly 500,000 [20]. Most of these are children who have learned about the programming environment from other children or from teachers. Scratch was originally introduced through after-school programming clubs [12], but recently it has been more widely adopted by users on the web at large [19]. The Scratch site now reports that 34% of registered users are 17 or older. In short, Scratch has moved from being primarily used under classroom supervision to being used "in the wild" by children and young adults toward self-directed goals.

Scratch's primary purpose is not to turn every user into a professional programmer, but rather to help people develop programming skills that they can apply later in everyday work and life [12][19]. These include technical skills related to making programs, social skills related to collaborating with other users, and socio-technical remixing skills related to using community resources to produce new programs. In short, Scratch is meant for teaching well-rounded end-user programming skills. In that regard, it represents the latest iteration in a series of animation environments that includes Logo [17], KidSim [4], AgentSheets [18], Alice [2], and Hands [15].

Where Scratch primarily differs from earlier environments is that it includes a rapidly-growing online repository containing animation projects and user comments [20]. The functional purpose of this repository is to support collaboration and remixing. A valuable side-effect is to provide a supply of animations and user comments that capture, with high ecological validity, the ways in which people actually use Scratch in the real world. As a result, we are now able to evaluate the real-world strengths and weaknesses of Scratch in a way that has never before been possible with earlier animation programming environments.

Our goal in this study was to assess whether Scratch is succeeding as a basis for developing various programming skills. In particular, we analyzed 100 Scratch animation projects and associated user comments to address the following questions:

RQ1: To what extent is Scratch succeeding as a basis for developing technical programming skills in the wild? More precisely,

what fraction of projects demonstrate correct usage of programming primitives toward functionally meaningful goals? Do their implementations tend to demonstrate a coherently organized structure, or are they a haphazard mess?

RQ2: To what extent is Scratch succeeding as a basis for developing social programming skills in the wild? In particular, how frequently do users provide useful feedback to one another? Do animation owners reply constructively to other users' feedback, do they act on feedback, and does any meaningful collaboration result? Is the overall culture supportive and constructive, or is it condemnatory and nit-picking?

RQ3: To what extent is Scratch succeeding as a basis for developing remixing skills in the wild? How frequently are animation projects remixed? How often does the downloading of source code actually result in remixing? Does remixing typically involve code, multimedia content, or both?

Overall, we found that the Scratch environment represents substantial progress toward its designers' goals of supporting the development of programming skills, though we also identified several opportunities for significant improvement.

With regard to technical skills, we found that projects in the Scratch repository generally did demonstrate mastery of programming primitives and program design, though only half were oriented toward functional purposes (such as storytelling or gaming) rather than experimental or artistic purposes (e.g., spinning a picture of a face). Animation projects tended to be approximately as complex as end-user programs in other domains (such as spreadsheets). Within animation projects, we found particular recurring implementation structures, which we describe in terms of design patterns.

With regard to collaboration and remixing, we found that the repository served as a supportive environment for generating constructive feedback from other users. However, we count not find even a single case where programmers actually acted on bug reports or feature requests from other people, nor did we observe any animation projects created by multiple programmers working in collaboration. Moreover, we found a low quantity of reuse, in that only 5% of projects were ever remixed by people other than the original author, and reuse was generally multimedia-centric rather than code-oriented. These weaknesses present opportunities for improvement.

The remainder of this paper is organized as follows. Section 2 provides background on Scratch as well as prior empirical work studying other repositories of end-user code. Section 3 describes the methodology of our process for acquiring and analyzing data. Section 4 discusses the technical aspects of animation projects that people created (RQ1), including common design patterns. Section 5 analyzes how users interacted with one another (RQ2), while Section 6 describes the low level of code remixing that we observed (RQ3). Sections 7 and 8 conclude by discussing the threats to validity as well as implications of these results for future research, including opportunities for improving animation tools and cultivating online communities where end-user programmers not only interact but actually collaborate.

2. BACKGROUND AND RELATED WORK

In Scratch and other typical animation design environments, a program consists of a set of animated sprites—images whose position and appearance are controlled with scripts (Figure 1) [19]. Perhaps the earliest such environment was Logo, a textual language and supporting toolset introduced in the late 1960's to teach children problem-solving skills, as well as technical skills such as how to properly construct loops and conditionals [17]. More recent environments such as Alice have added support for 3D animation [2], while other tools such as AgentSheets [18], KidSim [4] and Hands [15] have augmented the textual scripting

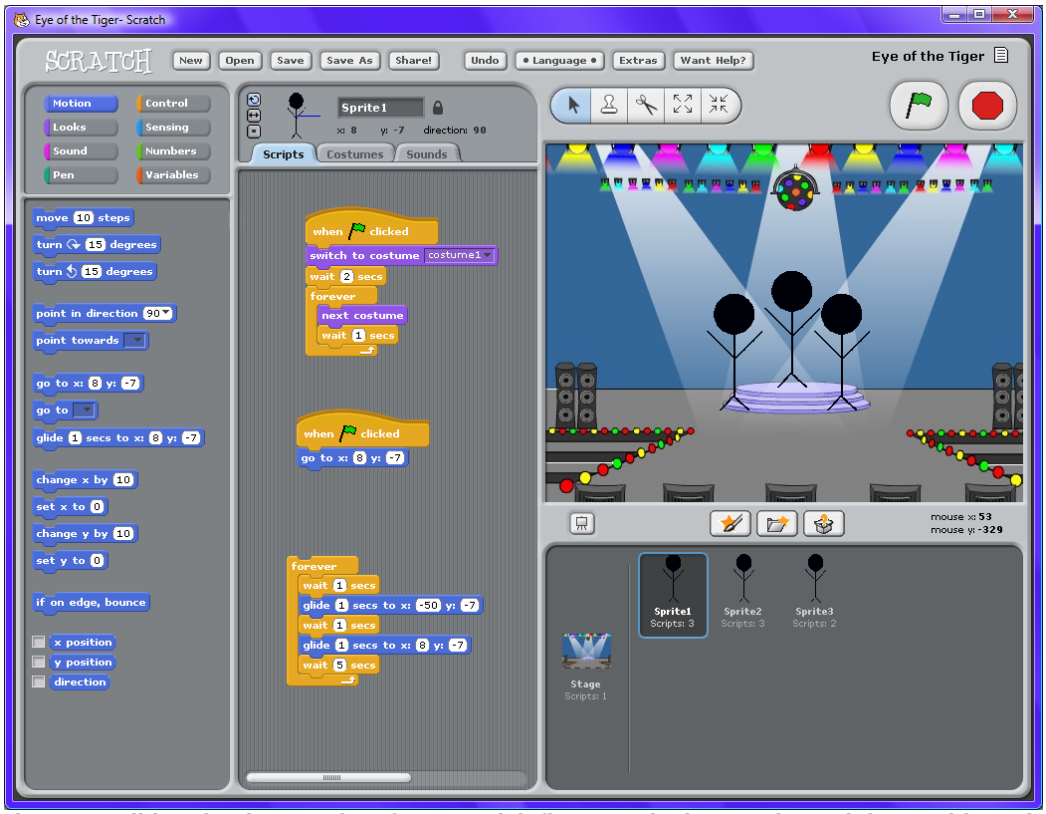

Figure 1. Editing the three scripts for one stick-figure sprite in an animated dance video. The programmer lays out sprites on the right; clicking a sprite brings up its scripts for editing in the center. Primitives can be dragged-and-dropped from the toolbox at left.

language with programming-by-demonstration (PBD) features: the programmer shows examples of inputs as well as examples of outputs, and the tool infers a program that produces the desired input-output relationship. The programmer can then review and modify the program if desired. (Scratch lacks PBD support.)

Laboratory experiments and field tests in classrooms have shown that animation tools like these can serve as effective platforms for teaching use of programming primitives [12][15], algorithm design [2] and problem-solving skills [15][18], as well as for motivating children and computer science majors to continue programming after they leave the classroom [4][14].

Scratch goes beyond prior tools by also including an online repository, whose purpose is to help programmers to learn social and remixing skills [13][19]. Specifically, Scratch's designers hoped that this repository would enable students to practice providing constructive feedback to one another, to form collaborative partnerships, and to creatively appropriate and adapt one another's animations.

Since the repository was launched in May 2007, many anecdotes have been gathered about its effectiveness for achieving these goals [13][19]. These anecdotes show that several programmers have indeed received significant feedback on some of the animations that they published to the repository; one project even obtained over 100 user comments. Several programmers have formed partnerships aimed at producing animations. And some 15% of repository animations have been created from old projects through remixing; for example, one particular Tetris game has been remixed dozens of times. These anecdotes suggest that the repository is succeeding in helping some programmers to develop social and remixing skills.

While we recognize the value of anecdotes in supporting qualitative statements, such as the claim that the community is supportive and collaborative, anecdotal evidence has a serious methodological weakness: because anecdotes focus on specific cases, there is a risk that they may have the unintended effect of portraying unusual episodes as common. Thus, it is difficult to generalize from anecdotes, unless if they are supplemented with statistical information indicating how representative they are. Moreover, when statistics are provided (such as the 15% number, above), it is difficult to understand whether to interpret these statistics as high or low unless if a basis for comparison is also provided.

Therefore, our goal in this paper is supply this statistical information, as well as to make informed comparisons to other relevant statistics, in order to assess whether and how we can generalize these anecdotes of success to the rest of Scratch's users. We do this by analyzing projects and online user comments, then comparing results to other end-user code repositories.

Methodologically-similar field studies have previously examined data from other repositories of end-user programming code, and four of these will periodically serve as a point of comparison in our work below. Two of these field studies examined spreadsheet programming; one was an analysis of over 4000 spreadsheets downloaded from the web [5], while the second was a survey of several studies looking at error rates in spreadsheets that companies had created [16]. The other two studies examined web macros (scripts that automate browser actions, for instance to direct the browser to visit a certain URL, to fill out a web form, then to submit it); one study of 120 macros aimed to characterize what kinds of macros were being created [1], while the other study of over 700 macros identified traits that differentiated oft-reused macros from rarely-reused macros [21]. These studies are useful points of comparison because they yielded statistics about the size, complexity and reuse of end-user programs, providing a quantitative basis for assessing how well Scratch is succeeding relative to other end-user programming environments. We also compare our results to several statistics that were collected during an intervention aimed at teaching students how to use Scratch in an after-school "Clubhouse" [12], in order to assess if end-user programmers in the wild have succeeded as well as those who had access to mentors.

3. RESEARCH METHODOLOGY

Obtaining data: To obtain animations and user comments for our analyses, we wrote a "screen scraper" program that downloaded 100 animations using the Scratch repository's randomized "surprise me" feature. In addition to each animation's code, we downloaded the HTML for the repository web page that gave the animation's usage statistics (counts of views, comments, downloads, and remixes) as well as the log of comments that users had posted about the animation (yielding a total of 269 comments).

Coding and analysis: At several points in our study, we needed to develop coding schemes to achieve our goal of assessing whether Scratch is succeeding as a basis for developing various programming skills required developing coding schemes. Although we were motivated by this research concern, we did not want to impose a particular pre-ordained coding scheme on the data but rather wanted to observe what the data told us about animation programming that was happening in the wild.

Therefore, except for specific situations discussed below, we used a grounded theory [8] approach to develop and apply codes. In grounded theoretical approaches, artifacts such as interview text are analyzed in pieces, concepts are identified in pieces, and codes are used to label pieces according to the concepts they reflect. Note that the codes are derived from the concepts reflected by the pieces, rather than imposed from an outside theory. In our case, we wanted to analyze animations and user comments, and the "pieces" of interest in our analyses of animations were visual pieces concretely implemented in sprites and scripts. For example, if one piece of an animation was a scoreboard, then this indicated that the animation was a game.

Our procedure was as follows. First, one co-author examined a subset of the animations or comments (~20%) and developed a tentative coding scheme. This coding scheme was then provided to a second researcher. They each applied the scheme to half of the animations and identified situations where the concept set represented by the codes were incomplete or a poor fit. In some cases, the scheme needed to be extended or otherwise modified, which they discussed until agreement was reached. After each had separately coded half of the animations, they tested the application of the coding schemes by checking each other's work. This resulted in disagreements for several animations (<10% of the total), all of which were completely resolved after discussion. Finally, we coalesced codes in each scheme when doing so increased clarity without reducing informativeness.

We used this procedure to generate *new* coding schemes when necessary, but there were a few situations where there was an old existing coding scheme that was clearly appropriate to use

(which we explicitly indicate in the following sections). For example, in Section 4.2, we used a categorization scheme developed in related work, as well as the scheme that is concretely reflected in Scratch's user interface. In Section 4.3, we reviewed animations for the presence of certain design patterns that related work had hypothesized would be useful in animation programming. In such cases, using a grounded approach to develop a novel coding scheme would have interfered with comparing our results to those presented in related work.

4. SCRATCH AS A BASIS FOR DEVELOPING TECHNICAL SKILLS

Our first research question asks to what extent Scratch is succeeding as a basis for developing technical programming skills. This question was motivated by the fact that one of Scratch's primary purposes is "to nurture a new generation of creative, systematic thinkers comfortable using programming to express their ideas" [19]. Scratch's effectiveness for teaching technical skills had been previously demonstrated in a Clubhouse setting, where mentors were available to guide students as they learned programming concepts such as variables [12]. To assess whether Scratch users on the web would prove equally adept at demonstrating "creative, systematic" use of programming concepts, we examined what sort of functionality people were creating, what programming constructs they were using, and whether the implementations reflected any coherent structural patterns suggesting systematic design.

4.1 Functional roles of animations

Scratch's designers summarize the repository's contents by writing, "The site's collection of projects is wildly diverse, including video games, interactive newsletters, science simulations, virtual tours, birthday cards, animated dance contests, and interactive tutorials, all programmed in Scratch" [19]. Similar lists elsewhere also mention stories and dance videos [12][13]. We wondered, how diverse is "wildly diverse" in practice? Are all of the kinds of projects mentioned in the summary list above actually common, or are Scratch programmers *mostly* succeeding at creating a few particular kinds of animations?

The repository summary list above was striking to us in that each animation mentioned has a functional role—a game's function is to entertain and challenge, a newsletter's function is to communicate knowledge, and so forth. Scratch's designers note that Clubhouse students sometimes created "pre-scripting" projects whose purpose was to experiment with Scratch rather than to implement particular functionality [12]; however, they do not mention the presence of such experimental projects in the repository. We wondered whether programmers in the wild were also creating experimental projects.

As we examined projects, we identified three codes—`education`, `game` and `story`—for labeling projects based on primary functional role (Table 1). The animations mentioned in the summary list above exemplify these codes: we coded video games and contests as `games`, tours and tutorials as `education` projects, stories and plot-like dance videos as `stories`. We observed no newsletters or birthday cards.

In contrast with these functional projects, we also observed many experimental projects with no clear functional role (except perhaps as art). In fact, some might not even be properly called "animations," in that their appearance was static. For example, one such `static trial` project simply played a Japanese tune while displaying a non-animated sketch of a purple-tailed cat. Projects coded as `animated trial` added some animation, for example by spinning or moving images around, though for no clear functional purpose. The `ani-inter trial` projects performed animation in response to user interaction, such as spinning or flashing images after receiving a keystroke or mouse input (though again to no clear functional purpose).

After coding projects, we computed the distribution of projects among categories (Figure 2). We found that only 44% had a functional role. Slightly over half of these were games, a quarter were stories, and the remainder were educational. `Education` and `game` projects almost always had animation and interactivity; stories always had animation but rarely had interactivity. Among the 56% remaining "trial" projects, which had no clear functional purpose, virtually all had animation but only a third had interactivity. Overall, 44% of all projects had some functional role, 94% had animation, and 45% had user interactivity.

When Scratch's designers examined 536 Clubhouse projects, they found that 79% had scripts [12]. They characterize the remaining 21% as pre-scripting experiments. Our results indicate that experimentation continues long after programmers begin

Table 1. Coding scheme for categorizing projects

Category	Main functional role	Example
`Education`	To teach a procedural skill or declarative facts, including do-it-yourself	Indicate when to perform the Heimlich maneuver
`Game`	To entertainingly challenge user to complete a goal	Get user to steer cat to collect food and avoid dog
`Story`	To communicate a fictional plot or storyline	Tell tale of mouse who has problems during visit to beach
`Ani-inter trial`	No clear functional role, has animation, has user interaction	Show letters J and L; flash J when user types the letter j
`Animated trial`	No clear functional role, has animation, no user interaction	Show several faces, one spinning and others flashing
`Static trial`	No clear functional role, no animation, no user interaction	Play background music and display a cat with a purple tail

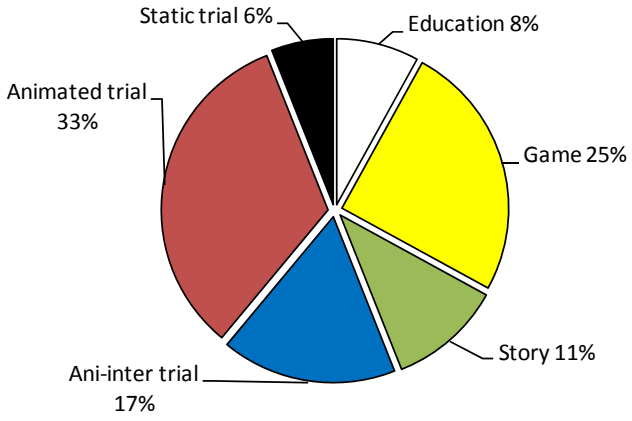

Figure 2. Distribution of projects by functional role

using programming primitives. Moreover, the resulting experimental animations make their way in large numbers into the repository where they are mixed indistinguishably among non-experimental projects.

4.2 Use of programming constructs

The Clubhouse represented a constructionist, learner-directed education environment [10]. That is, mentors "did very little to explicitly 'teach' programming concepts; rather, youth worked on projects of their own choosing and requested assistance from mentors when needed" [12]. When students wanted help, mentors showed how to use difficult programming primitives (e.g., variables [12]) or tool features (e.g., for image editing [10]).

In order to assess whether this mentoring approach enabled students to learn programming skills, Scratch's designers analyzed 536 Clubhouse projects to determine how many of the projects demonstrated usage of different programming primitives [12] (Table 2). They found that many primitives were used by some or many students, which they deemed a success. (They did not report whether the primitives were used properly.) Moreover, Scratch's designers argue that since mentors played a minimal role in teaching Clubhouse students how to program, it must therefore be true that much of students' learning resulted from the Scratch environment itself [12]. If this ascription is accurate, we would expect people outside of the Clubhouse to be just as successful at using programming primitives.

Upon examining Scratch projects from the repository, we found that they did indeed include key programming primitives at least as commonly as had previously been reported for Clubhouse projects (Table 2). This is consistent with the inference made by Scratch's designers that much of students' learning resulted from the Scratch environment itself rather than from mentor intervention. (This is not conclusive proof, however, since there could have been a selection effect in our data because programmers might choose not to upload low-quality animations.) In addition, while the Clubhouse analysis did not examine whether primitives were used correctly, we reviewed programs and only found 7 with major implementation bugs. This is a much lower rate than the 25% or higher typically found in spreadsheets [16].

We noted that repository projects demonstrated comparable or higher usage of programming primitives relative to other end-user programs. Specifically, whereas 45% of repository Scratch projects used variables, only 35% of spreadsheets used any cells as variable inputs for formulas [5]. Moreover, whereas 28% of animations used conditionals, only 8% of spreadsheets did [5]. In a study of web macros downloaded from a repository, only 20% contained variables, again lower than Scratch repository projects [1]. (The web macro tool did not support loops or conditionals, and average macro length was not reported.) Overall, Scratch projects averaged 110 primitives per project, nearly as many as spreadsheets, which averaged 168 formulas each [5].

Scratch was meant to teach not only general programming primitives (e.g., loops) but also animation-specific constructs. Using Scratch's user interface as a starting point (Figure 1), we grouped primitives into categories, then classified these categories as general or animation-specific (Table 3). We computed how many projects contained primitives of each category (Figure 3), and we computed total numbers of primitives for each kind of project, with a break down between general and animation-specific primitives (Figure 4). (We omitted Pen pri-

Table 2. Frequency of scripting and primitive usage

% of all projects with	Clubhouse [12]	Repository
≥ 1 script	79 %	87 %
≥ 2 scripts	70 %	74 %
≥ 1 input event handler	43 %	45 %
≥ 1 loop	41 %	58 %
≥ 1 conditional	21 %	28 %
≥ 1 variable	8 %	23 %

Table 3. Codes for categorizing primitives; asterisks indicate general programming primitives.

	Category	Meaning
	`start`	Event handler fired when project loads on repository (or runs in programming tool)
*	`control`	Loops, conditionals, thread synchronization
	`looks`	Modifies the appearance of a sprite
	`motion`	Slides or rotates a sprite
	`sound`	Plays music or other sounds
*	`event`	Event handlers fired by user input
	`sensing`	Functions to read values, such as to detect whether sprites are touching
*	`number`	Operators for numerical (and Boolean) values
*	`variable`	Declare or reference variable

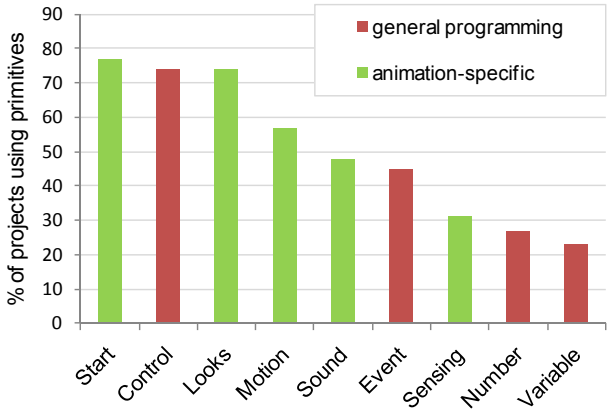

Figure 3. Commonness of primitives, by category

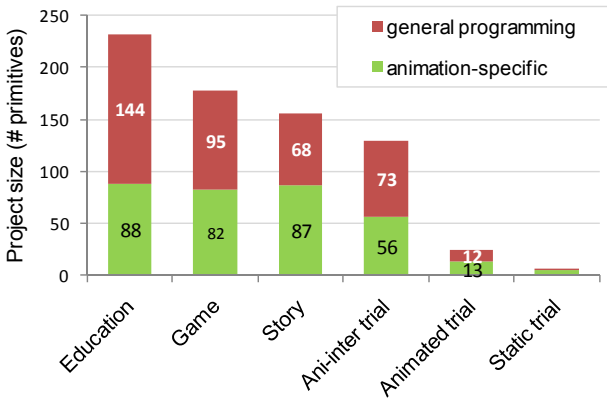

Figure 4. Numbers of primitives used in projects

mitives, since they were used very rarely. We separated out the `start` signal and the user input `event` primitives from the non-interactive control primitives, such as conditionals and loops.)

We found that most animation-specific primitives appeared just as frequently as most general programming primitives. Overall, projects contained approximately equal numbers of animation-specific and general programming primitives (averaging 51 and 59, per project, respectively), with a fairly even balance generally demonstrated within most kinds of project. Of course, interactive projects tended to contain many more primitives than non-interactive experiments. We were surprised that `education` projects contained the most animations; this high count resulted from the large numbers of primitives required to implement educational simulations. Overall, Scratch users appear to demonstrate as much competence with animation-specific primitives as with general programming constructs.

4.3 Design patterns

While Scratch's designers have primarily focused on Scratch's usefulness for teaching children how to use programming primitives [12], we wondered whether Scratch projects also commonly reflect any sort of higher-level programming design skills, or if instead they are what professional programmers sometimes refer to as "spaghetti code." To frame our investigation, we focused on whether Scratch projects demonstrate design patterns, which are general solutions for structuring programs in response to common programming problems [6]. Related work examined one particular game (implemented in Java) in order to identify design patterns that might be expected to appear in many games, and which therefore might be teachable by having computer science students implement games [7]. Using this prior study as a starting point, we asked whether and how Scratch projects commonly implement these six design patterns, in order to assess if Scratch might be useful for teaching the patterns. While the case study mentions six patterns, only two are particularly prominent in Scratch animations, so we focus on these below.

Game Loop design pattern

In the Game Loop pattern originally identified by the case study, a Controller object with a "while(!game is over)" loop continually watches for user inputs and sends messages to sprites, telling them to update their state accordingly (Figure 5). (In Java, the loop typically contains some code for double-buffering [7], but the Scratch environment handles all buffering and rendering, so Scratch scripts omit this part of the loop.) By centralizing all input-processing code and the mapping to sprite messages in a single Controller object, this pattern makes it easy to find and maintain this code later [7].

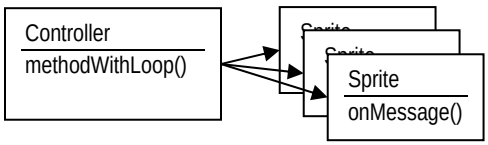

Figure 5. Game Loop design pattern

A handful of complex Scratch games implement this pattern by making the Stage sprite serve as Controller. The Stage is a global Scratch object that serves as a white visual background in all projects. Like other sprites, the Stage can have a script. Within this script, the programmer can place a "repeat until game ends" loop, and inside of that loop, lines of code can read inputs, may set global variables, and then send messages to other sprites. For example, when one particular game's Stage receives keystrokes from the user, it sends all "enemy" sprites a message so that they move a step toward the "player" sprite.

Frequently, Scratch animations only contain a single interactive sprite. In these cases, programmers placed the game loop directly in that sole interactive sprite's script. This is sensible, since in these cases, applying the pattern to separate logic into a Controller would increase rather than decrease complexity.

A handful of games had multiple interactive sprites but did not apply this pattern, even though doing so would have helped to centralize input collection. Instead, these attached game loop-like logic to every single sprite (with a great deal of repetitive code). Another handful applied the pattern but inconsistently, using it to centralize some aspects of interactivity but not others.

Collision Detection design pattern

The Game Loop pattern originally identified in the case study associated a CollisionHandler object with each sprite (Figure 6). This handler for each sprite implements a variant of the Visitor Pattern [6], in that its "onHit" method would be called for each sprite that had collided with the handler's sprite. The handler subclass could then update the state of its sprite if appropriate by overriding whichever onHit methods are needed for whatever kinds of collision are important for gameplay. For example, if a "player" sprite hits a "bomb," then the player sprite's handler could update the player sprite to a "dead" state.

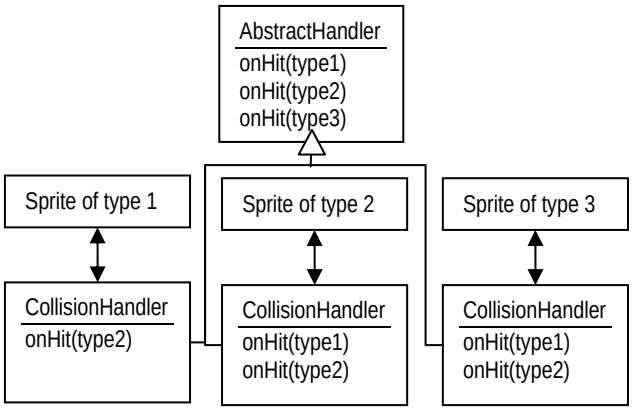

Figure 6. Collision Detection design pattern; note that the handler subclasses polymorphically implement only selected versions of the onHit method

Since Scratch lacks double-dispatch, polymorphism, and even inheritance, it is impossible to precisely implement this pattern as shown above. However, we observed over a dozen projects that implemented an approximation of this pattern.

Specifically, when a project's gameplay required testing whether two sprites intersected, each sprite contained a loop that constantly checked for collisions with certain other sprites. For example, in one project, a "cat food" sprite continually checked for collisions with a "cat" sprite so that the food could be changed to an "eaten" state on contact. Many sprites also had a second loop that continually watched for collisions with a second kind of sprite. For example, the "cat food" sprite also watched for collisions with a "dog" sprite (and, on collision, the food went away). Separating these collision checks into distinct scripts represents an approximation of the separation of concerns into

distinct methods that is reflected by the Visitor-like pattern above (essentially implementing the equivalent of two "onHit" methods). We only saw one situation where this pattern was not implemented correctly (omitting the loop, resulting in collisions not being reliably detected).

In short, we found some evidence of two design patterns in approximately a dozen of 100 animations, suggesting that Scratch might be useful for developing skill with these patterns, just as it seems to serve as an effective basis for learning primitives.

5. INTERACTION AMONG USERS

Our second research question asks to what extent Scratch is succeeding as a basis for developing social skills related to programming. Scratch's designers write, "For Scratch to succeed, the language needs to be linked to a community where people can support, collaborate, and critique one another" [19]. As evidence that Scratch achieves this goal, at least two papers describe a collaboration between six children who work together to create an entire gallery of animations, each person contributing different skills aimed at implementing specific features such as sliding backgrounds [13][19]. Another piece of evidence cited in support of Scratch's success is that fact that one particular programmer once received over 100 comments on a specific project, and that she then organized a contest for other programmers to create extensions for her project [19].

We wondered whether the experiences of the seven children mentioned above are actually typical, or if these are instead representative of the "high end" of collaboration and socialization on the site. We wondered whether the majority of Scratch programmers are demonstrating effective collaboration and other supportive social skills, or if the community is rude and hyper-critical. Therefore, we analyzed all of the 269 comments that people exchanged with one another in the context of the 100 projects that we downloaded, enabling us to assess what kinds of feedback are exchanged, whether comments are constructive or nit-picking, and what collaboration is revealed by communication among programmers.

5.1 Commenting on each other's projects

Based on project statistics, we noted that virtually all projects had received enough exposure on the site that they could plausibly have received comments. Specifically, every one of the 100 projects was viewed at least one time, and approximately 90% were viewed at least 5 times, with the overall average being 24.1 views. On average, projects had been published on the repository 397 days, with approximately 90% exceeding 200 days.

We found 60% received at least one comment, with an overall average of 2.66 and maximum of 20 (Figure 7). Thus, most projects did receive some user feedback, with 1 in 9 views leading to a comment. However, none even approached the 100+ comments that some project once received (above).

In performing the coding procedure outlined in Section 3, we discovered three categories that involved feedback about project quality—`criticism`, `feature suggestions`, and `compliments` (Table 4). Outright `compliments` constituted 27% of comments, and the 21% that were `feature suggestions` also tended to be positive (Figure 8). Of the 6% that were `criticisms`, some were polite bug reports, while others were just nit-picking or rude. Together, these three categories of comments

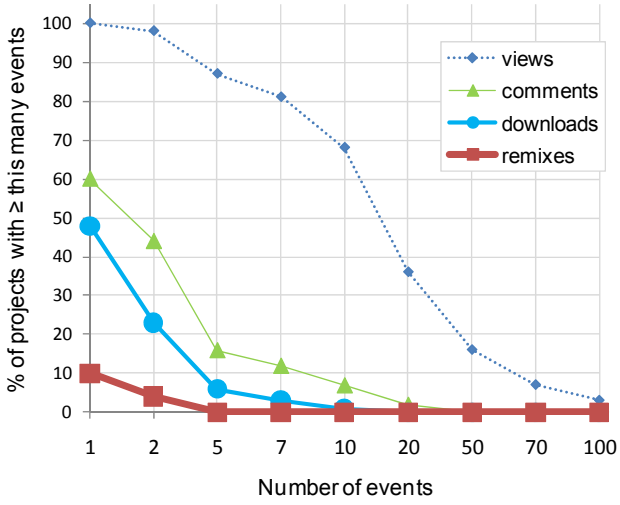

Figure 7. Distributions of numbers of views, comments, downloads, and remixes

Table 4. Coding scheme for categorizing comments

Category	Main message	Example
`Criticism`	I do not like your project; may mention specific bug	"this is creepy and stupid no offence"
`Feature suggestion`	I like your project and have an idea to improve it	"nice drawing but a bit tall. but aren't L.E.P. flight suits jet black? not green."
`Compliment`	I like your project; does not mention improvement ideas	"its so nice and like wat spongie06 said its relaxing"
`Chit-chat`	I have something to say that isn't a criticism, suggestion, or compliment	"did you make mario out of clay?"
`Author replies`	I acknowledge your comment and have a response	"ur strainge. no i dont torture stuffed animals. i take good care em."

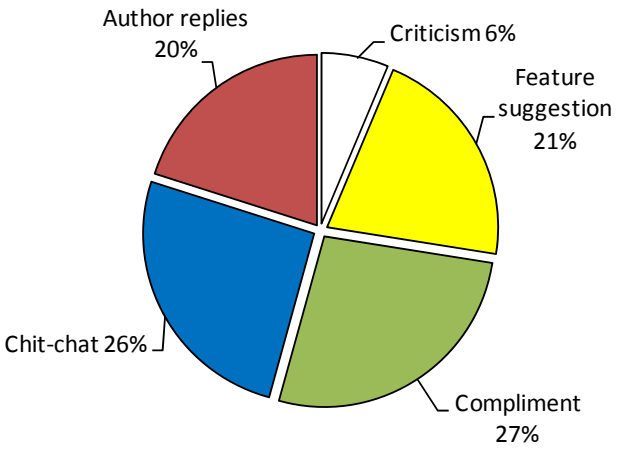

Figure 8. Distribution of kinds of comments

demonstrate that Scratch participants overwhelming are offering helpful critiques without sacrificing congeniality.

The repository has also become a venue for socializing, as reflected by the fact that 26% of comments were `chit-chat`. These represent comments that were not specifically about the project but seemed to be sparked by the project. For example, commenting on a `game` project, one person said that he had a similar game for his Xbox. Very many `chit-chat` comments were questions to the project author about prior experiences, either with this project or with other projects. For example, one person asked, "Why do you have so many projects?"

One in four comments garnered an `author reply`. It appears that most replies were directed in response to `chit-chat`, and we noted that certain authors seemed much more likely than others to habitually respond.

5.2 Active collaboration on projects

As we reviewed projects and comments, we looked for any indication that interactions led to collaboration. By "collaboration," we mean joint design or implementation by a team of multiple people consciously working on a common intellectual goal (as in the anecdotes mentioned at the start of this section). Succinctly: we hoped to see people working together. For example, collaboration could hypothetically be reflected in comments like, "Thanks for the compliment—my friend helped me to make that," or "I have implemented the redesign that you suggested; how do you like the result?" or perhaps comments from multiple people implying that they had contributed code to a project. We also compared the `feature suggestions` to the projects themselves, looking for any indication that authors appeared to have actually acted on any suggestions (without an explicit `author reply`).

We found no such indications at all for even one project. While we accept that people might be reusing each other's code (as discussed below), and they might be inspired by each other's projects (as argued by Scratch's designers [13][19]), these interactions differ from people consciously working together toward a common goal. Thus, we must conclude that the design and implementation of Scratch projects is rarely collaborative, or at least that it rarely leaves discernible traces in online comments.

6. REMIXING

Our final research question asks to what extent Scratch is succeeding as a basis for developing remixing skills. By "remixing," Scratch's designers mean "creative appropriation… the utilization of someone else's creative work in the making of a new one" [13]. Professional programmers would recognize this as whitebox reuse. In the context of animation programming, remixing encompasses code as well as multimedia content.

Remixing differs from collaboration, in the sense that it is possible to appropriate another person's belongings without having any shared intellectual goals with that other person. Indeed, the other person might not even be aware that remixing occurred. (Soon after the repository opened, some users expressed irritation that their materials had been appropriated [13]. This prompted Scratch's designers to tweak the site's features and wording to emphasize that the remixed project's author has been successful rather than robbed [19].)

Prior reports indicate that 15% of repository projects have been created by remixing materials from other projects [19]. We wondered about the inverse question: How proportion of animations are remixed? The two statistics could differ quite a bit, for example if one or two projects are remixed often but no others are remixed. We also wanted to understand whether remixing frequently follows downloading, or if people frequently download code without successful remixing. Also, thinking of the analogy to whitebox reuse, we wanted to understand what changes are made by programmers during remixing.

6.1 Frequency of downloading and remixing

Based on statistics provided by the repository for our 100 randomly selected end-user projects, we found that half had been downloaded at least once, with an overall mean of 1.16 downloads per project (Figure 7). After these 116 downloads, 15 remixes were created. It is not possible to know whether any of these downloads accounted for multiple remixes, but it is certain that at least 116-15=101 downloads did not lead to remixing. Thus, for the most part, downloading does not lead to successful remixing. Instead, when downloading occurs, something else must normally ensue: perhaps programmers read the code to learn from it, or perhaps they attempt to remix it and fail.

We found that 10% of projects had been remixed once, 3% had been remixed twice, and 1% remixed three times. That is, 40% of the remixed projects accounted for 60% of the remixes—so the remixes that do occur are not concentrated in a small subset of remixed projects, but rather are spread around. Of the 15 remixes, 10 were done by the programmer who initially created the code, and 5 were done by others.

These levels of reuse do not appear to be markedly higher than the level of reuse previously observed in a study of CoScripter web macro reuse [21]. In that study over a 90 day period, only 4% of web macros were ever copied and used to create another macro by any person other than the original macro author. Here, in the Scratch repository, we noted a corresponding 5% rate of reuse by other people. Observe that this figure for Scratch includes remixes over a much longer period averaging 397 days. Given this consideration, it is notable that the Scratch rate was so close to that of CoScripter macros rather that conspicuously higher.

For comparability to the Scratch statistic previously reported, that 15% of all Scratch projects are remixes of old projects [19], we computed what fraction of our 100 projects were remixes. We found that 20 were remixes, fairly close to the previously-reported statistic. Of these, 7 were remixes of the author's own prior work, 11 were remixes of other end-user programmer code, and 2 were remixes of a sample project previously created by Scratch's designers. [1]

[1] This sample project that was remixed twice is one of provided by Scratch's designers at MIT. According to the repository, a total of 3870 of its 912,431 current projects are remixes of this one sample project. Scratch's designers are not end-user programmers, of course, and the sample project is a well-written, engaging "Pong" game that serves as a Scratch tutorial. Therefore, it is unsurprising that the sample's level of remixing is so much higher than that of typical end-user projects.

6.2 Changes during remixing

In order to understand what changes are typically entailed in remixing, we compared the 20 of our 100 projects that were remixes to the old projects that they were based on.

In 70% of these remixes, we found that the new project had multimedia changes relative to the old project (Figure 9). Half of these multimedia remixes simply involved replacing one sprite image with another image and publishing the result. The other half included creating new images as well as tweaking the old.

In 40% of remixes, we found changes to scripts. Most of these changes added or changed features, such as adding a new interactive sprite. Many script changes also included bug fixes. However, approximately half of the script changes introduced new bugs that were serious enough to render the resulting project mostly or completely unusable.

In 15% of remixes, we observed no modifications at all to the remixed project. It was simply republished with a new title and/or description.

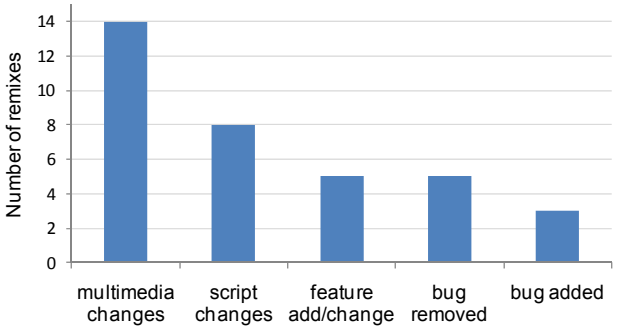

Figure 9. Events that occurred during 20 remixes

In order to give an idea of the "best case" remixing that we observed, we close by describing two remixes that we consider to be the most successful of the 20. More precisely, these remixes were not buggy, they reused substantial functionality from the old project, and they also added noticeable new features.

The first of these is a game where the user steers a "player" sprite through a series of puzzles. In each puzzle, the user must move the "player" sprite so it intersects a "door" sprite, in order that the user may progress to the next puzzle. The game includes logic for steering sprites, simulating gravity, computing the effects of barriers, moving doors around the screen, and detecting when to move to the next puzzle.

The remixed version was created by a different person than the first project's author. This remixed version includes several new puzzles, which are apparently implemented entirely by adding new images to the old code—no script changes were required. These images are like visual configuration files for the scripts, since the colors of the pixels indicate the presence of barriers and other puzzle details. In a sense, this remix successfully provides new functionality by loading new data into the old project.

The other particularly successful remix that we observed was an extension of a "paint" program. This source project had a toolbox panel for selecting a painting tool, as well as a space for users to draw lines and patterns with the selected tool. One usability problem (not a functional bug) is that the toolbox took up so much screen that little space was left for actual drawing.

In order to fix this usability problem, another user remixed this project by adding functionality for minimizing the toolbox. This is implemented with a new button for toggling between "hide" and "show." Implementing this remix required adding a new button sprite, including the image for its appearance and the hide/show script for responding to button presses.

Although these two remixes successfully reused significant functionality while adding substantial bug-free functionality, they were outliers among the 20. Most of the other multimedia tweaks in remixes were artistic tweaks to sprite appearance, and most of the other functional enhancements in remixes were more minor than the "add button" change that we described above. Thus, while 20 of our 100 projects were indeed based on prior projects, only a handful added substantial bug-free functionality.

7. THREATS TO VALIDITY

The primary threat to this study's internal validity is that examining artifacts (such as animations and web pages) offers an imperfect view of peoples' context and knowledge. Our goal was to assess whether Scratch is succeeding as a basis for developing various programming skills. Based on an analysis of the animations, it *appears* that programmers had particular skills when they created animations, but we cannot be certain that they *actually* had these skills. An alternate explanation is that they could have gotten help from other people, or they could have randomly experimented with Scratch until they finally produced animations without actually understanding the result or process (and hence without actually having any skill). These concerns are a direct result of the fact that we have very little contextual data about the people who use the Scratch repository; for example, as outlined in Section 1, we believe that Scratch is being used primarily by independent users at this point, but it is possible that some subset of them are physically co-located with one another and might collaborate in person. As sketched in Section 8, such concerns could be resolved by experimenting with programmers directly, making it possible to control the context.

In addition, our coding schemes were induced from the data (as in grounded theory research) rather than being imposed on the basis of some external theory. As a result, a threat to external validity is that the coding schemes could be "overfit" to this data set and not generalize to other data or other repositories. To mitigate against this threat, we reviewed all coding schemes and examined if the codes depended on properties of Scratch, or if the codes were instead conceptually abstracted from this particular programming platform. Reviewing the Figures and Tables in this paper quickly reveals that none of the coded concepts are specific to Scratch, though many are particular to animation rather than programming in general. Therefore, we anticipate that our codes could also be applied to other animation environments but not to all cases of end-user programming.

8. CONCLUSIONS

We have analyzed animation programs and user comments from the online Scratch repository in order to evaluate how well the development environment is successfully serving as a basis for demonstrating technical, social, and remixing skills related to programming. Overall, we found that Scratch represents substantial progress toward these three goals, though we also identified several opportunities for significant improvement.

Technical skills: Concerning technical programming skills, we found that repository projects utilized key programming primi-

tives at least as commonly as had previously been reported for Clubhouse projects. Moreover, repository animations were just as complex as other end-user programs such as spreadsheets and web macros. Scratch programs were at least as bug-free as spreadsheets, as well. We even found a small amount of evidence that Scratch programmers use a few higher-level design patterns (albeit inconsistently and quite possibly at a subconscious level). All of these pieces of evidence point toward Scratch's success as a platform for developing technical programming skills.

Further research could aim to identify additional design patterns in Scratch programs, as well as to augment the Scratch environment with new features that help programmers to select and apply patterns as needed. Laboratory experiments would measure whether these enhancements enable people to more thoroughly develop crucial higher-level programming skills.

Social skills: We found that that Scratch programmers generally have offered helpful critiques online without sacrificing congeniality, yet as far as we could determine, these interactions never led to online collaborations.

To facilitate online collaboration, new data structures and supporting repository features could be provided to help multiple programmers divide up the work of creating an animation, then to coordinate and combine their pieces. Between-subjects field testing could assess whether such enhancements facilitate the development and application of social programming skills.

Remixing skills: Repository projects demonstrate somewhat more success with multimedia-oriented remixing than code-oriented remixing. Most remixes included changes to multimedia artifacts, but most did not include changes to code. Moreover, nearly half of the code-remixing activities led to the introduction of major bugs.

Researchers have developed many tools to support code reuse by professional programmers, including automated tools for code slicing [11], transformation [3] and dependency-tracing [9]. These powerful techniques enable programmers to explore, analyze, modify, combine and extend code. Further research might simplify and adapt these approaches to the context of end-user programmers so that they can more successfully perform similar tasks with Scratch animation code. The effectiveness of many such features could be evaluated in a laboratory setting.

Skill transfer: In all three areas of skill development—technical, social and remixing—the ultimate purpose of Scratch is not to provoke students into becoming professional programmers, but rather to provide them with skills that will prove useful in careers outside professional software development. It is therefore appropriate to explore whether any skills developed through Scratch successfully transfer to other domains.

The effectiveness of Scratch for teaching transferable skills could be assessed with longitudinal studies aimed at measuring whether students with Scratch experience are more adept at learning other end-user programming tools. Studies such as these require attention to adequate controls, as well as the opportunity to establish multi-year contact with study participants. Such an experiment would not only evaluate the transferability of skills taught with Scratch, but it might also identify additional opportunities to improve the Scratch environment to further assist and motivate students as they develop programming skills.

9. REFERENCES

[1] Bogart, C., et al. 2008. End-user programming in the wild: A field study of CoScripter scripts. *2008 Symp. on Visual Lang. and Human-Centric Computing*, 39-46.

[2] Cooper, S., Dann, W., and Pausch, R. 2000. Developing algorithmic thinking with Alice. *Information Sys. Educators Conf.*, 506-539.

[3] Cottrell, R., Walker, R., and Denzinger. 2008. Semi-automating small-scale source code reuse via structural correspondence. *16th Intl. Symp. on Foundations of Software Engineering*, 214-225.

[4] Cypher, A., Smith, D, and Tessler, L. 2001. Novice programming comes of age. In *Your Wish is My Command*, Morgan Kaufmann, 7-20.

[5] Fisher, M., and Rothermel, G. 2005. The EUSES spreadsheet corpus: A shared resource for supporting experimentation with spreadsheet dependability mechanisms. *1st Workshop on End-User Software Engineering*, 47-51.

[6] Gamma, E., et al. 1995. *Design Patterns: Elements of Reusable Object-oriented Software*. Addison-Wesley Professional Computing Series.

[7] Gestwicki, P. 2007. Computer games as motivation for design patterns. *38th SIGCSE Technical Symp. on Comp. Sci. Education*, 233-237.

[8] Glaser, B., and Strauss, A. 1967. *The Discovery of Grounded Theory: Strategies for Qualitative Research*, Aldine Publishers.

[9] Holmes, R., and Walker, R. 2007. Supporting the investigation and planning of pragmatic reuse tasks. *29th Intl. Conf. on Software Engineering*, 447-457.

[10] Kafai, Y., et al. 2008. Mentoring partnerships in a community technology center: A constructionist approach for fostering equitable service learning. *Mentoring and Tutoring: Partnership in learning, 16*, 2, 191-205.

[11] Lanubile, F., and Visaggio, G. 1997. Extracting reusable functions by program slicing. *Transactions on Software Engineering, 23*, 4, 246-259.

[12] Maloney, J., et al. 2008. Programming by choice: Urban youth learning programming with Scratch. *39th SIGCSE Technical Symp. on Comp. Sci. Education*, 367-371.

[13] Monroy-Hernández, A., and Resnick, M. 2008. Empowering kids to create and share programmable media. *Interactions, 15*, 2 (Mar-Apr 2008), 50-53.

[14] Moskal, B., Lurie, D., and Cooper, S. 2004. Evaluating the effectiveness of a new instructional approach. *ACM SIGCSE Bulletin, 36*, 1 (Mar 2004), 75-79.

[15] Pane, J. 2002. *A Programming System for Children that is Designed for Usability*. PhD Dissertation, School of Computer Science, Carnegie Mellon Univ..

[16] Panko, R. 1998. What we know about spreadsheet errors. *J. End User Computing, 10*, 2, 15-21.

[17] Papert, S. 1980. *Mindstorms: Children, Computers, and Powerful Ideas*, Basic Books, Inc.

[18] Repenning, A. 1993. *Agentsheets: A tool for building domain-oriented dynamic, visual environments*. PhD Dissertation, Dept. of Computer Science, Univ. Colorado-Boulder.

[19] Resnick, M., et al. 2009. Scratch: Programming for all. *Communications of the ACM, 52*, 11 (Nov 2009), 60-67.

[20] http://scratch.mit.edu

[21] Scaffidi, C., et al. 2009. Predicting reuse of end-user web macro scripts, *2009 Symp. on Visual Lang. and Human-Centric Computing*, 93-100.