Automatically Generating Tutorials to Enable Middle School Children to Learn Programming Independently

Kyle J. Harms[1], Dennis Cosgrove[2], Shannon Gray[3], Caitlin Kelleher[1]

[1]Washington University in St. Louis
One Brookings Drive
St. Louis, MO 63144
harmsk@seas.wustl.edu
ckelleher@cse.wustl.edu

[2]Carnegie Mellon University
5000 Forbes Avenue
Pittsburgh, PA 15213
dennisc@cs.cmu.edu

[3]Bard College
PO Box 5000
Annandale-on-Hudson, NY 12504
sg6337@bard.edu

ABSTRACT

Enabling middle school children to learn from code shared on the internet may provide computer science learning opportunities to those who would not otherwise have them. We augmented a programming environment designed for middle school children to automatically generate tutorials from code snippets in order to help users learn new programming skills. In our new system, users select code snippets from a program shared on the web and then complete an automatically generated tutorial in order to re-create that snippet within their own program. To evaluate the potential learning gains from our generated tutorials, we conducted a between-subjects study in which we evaluated the performance of children introduced to new programming constructs through automatically generated tutorials. Participants who used the automatically generated tutorials performed 64% better on a near transfer task compared to participants without generated tutorials.

Categories and Subject Descriptors

H.5.2 [**User Interfaces**]: Training, help, and documentation.

General Terms

Design; Human Factors.

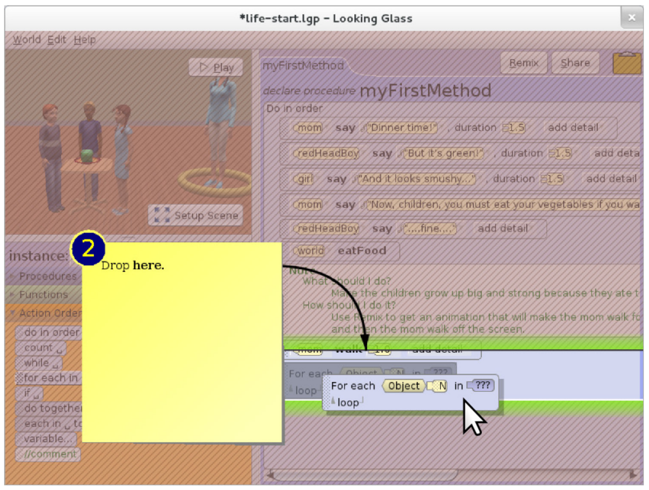
Keywords

automatically generated tutorials; programming systems for children; code reuse

1. INTRODUCTION

Today, computing technologies play an increasingly critical role in progress across a wide range of disciplines. To sustain the promise for improved computing technologies in the future requires a large technical workforce. In the United States alone, experts predict that approximately 1.4 million computing jobs will be created between 2008 and 2018 [38]. Yet, based on current graduation rates, many predicted jobs will go unfilled in both the United States [38] and Europe [23].

Despite strong job prospects, recent enrollment in computer science degree programs remains low [40]. Furthermore, by the time students reach college, many have already opted out of math and science courses and are too far behind to succeed in computing degree programs [34, 41]. Meeting the potential demand for computer scientists requires that we introduce students to computer science before college. While programming is taught in some high schools, middle school is the time that many children, especially girls, begin to opt out of math and science related fields [39]. Few middle school children have access to the resources and formal opportunities, such as programming courses, that can expose them to computing and foster an interest in the field. In the absence of formal opportunities to explore computing, enabling children to learn programming independently while following their own interests may offer a viable alternative.



```
declare procedure myFirstMethod
Do in order
    mom say "Dinner time!" , duration 1.5 add detail
    redHeadBoy say "But it's green!" , duration 1.5 add deta
    girl say "And it looks smushy..." , duration 1.5 add detail
    mom say "Now, children, you must eat your vegetables if you wa
    redHeadBoy say "....fine...." add detail
    world eatFood
    Note
      What should I do?
          Make the children grow up big and strong because they ate t
      How should I do it?
          Use Remix to get an animation that will make the mom walk fo
          and then the mom walk off the screen.
    mom walk 1.0 add detail
    For each Object N in ???
    loop
```

Figure 1. An automatically generated programming tutorial shown in the stencils walk-through mode.

Novice programming environments like Storytelling Alice [25] and Scratch [35] motivate children to program, but currently lack the support to enable users to learn programming concepts independently. Professional and end-user programmers frequently use code found on the web to learn new skills [4, 9, 36]. The plethora of shared source code available on the web affords users the option to find relevant code while pursuing a personally meaningful project. Empowering children to utilize shared code as a learning resource necessitates giving them the tools necessary to easily reuse source code and the support needed to effectively learn from it.

Existing code selection tools can help children to find relevant code snippets in unfamiliar code [13]. However, users may not understand the programming concepts used in the snippet. In this paper, we present a system for middle school children that automatically generates interactive, in-context programming tutorials from code snippets for the Looking Glass [28] programming environment. Our tutorials, as shown in Figure 1, provide explanations of new programming constructs found in code snippets and a detailed walk-through that guides users through the mechanics necessary to re-create those code snippets. We conducted a between-subjects study to evaluate the learning gains for unfamiliar programming constructs found in code snippets. We found that participants who completed an automatically generated tutorial after selecting a code snippet with a new programming construct performed 64% better on a near transfer task than those participants without a generated tutorial.

2. RELATED WORK

Our work on the generation of tutorials from a code snippet is built upon two specific areas of research: 1) leveraging existing source code and 2) presenting and generating tutorials.

2.1 Leveraging Existing Source Code

Existing source code usually comes in two forms: 1) carefully crafted examples and 2) raw source code. Prior research has investigated several methods in order to leverage both types of source code in an effort to help programmers with their tasks.

Currently, many of the tools designed to leverage existing source code integrate pre-authored examples into development environments to assist programmers when authoring code [2, 3, 21, 31]. Some of these tools assist experienced programmers to more efficiently write code by associating source code with API documentation [2] or linking source code to web browsing history to maintain the programmer's context [21]. Others enable programmers to directly search the web for code examples from within the development environment [3]. Codelets helps programmers integrate example code into their programs by allowing users to tweak parameters within the example to see how the changes affect the program's output [31]. These tools can reduce development time and improve code quality but do not directly support learning from the existing source code [2, 3, 31]. Unfortunately, all of these tools require pre-authored examples, which are typically time consuming to produce and may not match the programmer's current context.

Allowing users to utilize existing source code without crafted examples may help users find more personally relevant code to use in their own programs. Scratch and Kodu allow children to create modified versions of shared programs [29, 35]. However, reusing entire programs may limit the utility of shared code in user's programs. Researchers have identified several strategies and potential barriers to enable novice programmers to select code snippets from unfamiliar programs [13, 14]. In fact, one such system enables novice programmers to select snippets by linking the graphical output of a program to the source which caused that output [15, 20]. Interestingly, this system found that a majority of users modified the behavior of or re-appropriated concepts from selected code snippets [15].

2.2 Presenting and Generating Tutorials

There is a long history of research related to presenting tutorials and, more recently, automatically generating tutorials.

2.2.1 Presenting Tutorials

Early studies of text and image-based tutorials revealed two common problems: 1) users skipping or making mistakes in the execution of steps [27] and 2) users struggling to perform on-screen instructions [27]. In response, researchers have explored alternative presentation styles for tutorials. Palmiter et al. found that users of animated tutorials were able to complete tutorials more quickly than users of a purely text-based tutorial, but did not retain the material as well [32]. Other researchers have explored presenting procedural information within the context of the application by visually indicating the components needed for a step [8], or overlaying a graphical, event-intercepting stencil atop the interface [26]. In-context tools can help users to both find the UI components needed for each step and reduce the potential for error [26]. Alternative presentation styles that ask users to attempt a tutorial task before guiding them through the task may alleviate the retention issue noted with animated tutorials [19].

Additional research suggests that users often search for materials relevant to a personal task [6]. This observation has inspired work exploring short, task-based guides as an alternative to tutorials [5] and placing short video clips documenting features of the interface within tooltips to support user interface exploration [17].

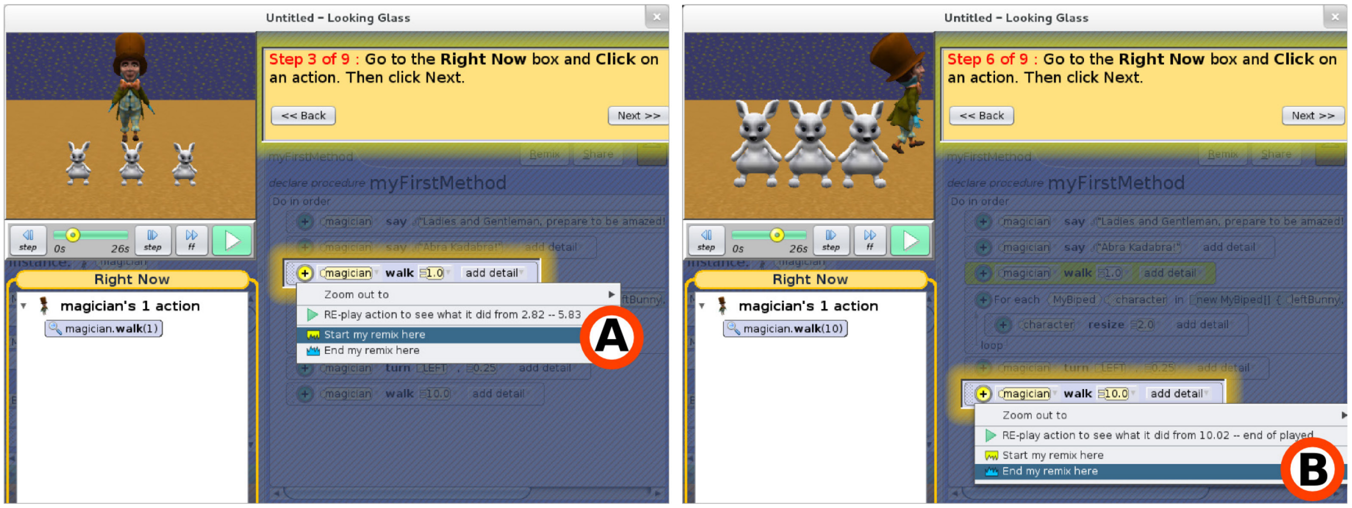
2.2.2 Automatically Generating Tutorials

In Mindstorms, Seymour Papert envisioned learning as self-motivated and connected to popular culture [33]. Particularly for learners with limited community support, the success of self-directed learning may depend on the availability of learning materials appropriate to each learner's self-motivated project. Reducing the cost of creating tutorials, or removing it altogether, creates the potential for a dramatic increase in available learning materials. This may, in turn, make learning in pursuit of personally meaningful projects more broadly achievable. Automatically generated tutorials can be created and presented separately from the application (e.g. text and image based or video tutorials) or within the application's context.

Several systems generate image-based tutorials by listening to an event stream and capturing pictures. These events and pictures are used to create a traditional text and image based tutorial [11], a comic-style visual history of changes [30], and graphical summaries of a task presented in a single page [22]. Rather than simply capturing an event stream, mixT records a full screencast and generates tutorials containing a list of textual steps supplemented by short video illustrations of those steps [7]. SmartTutor has a similar end-user experience but, rather than capturing videos, simply re-sends the captured events to the live system [37]. While systems that generate textual, image, or video-based tutorials dramatically reduce [7, 11, 22, 37] or eliminate [30] the authoring time necessary to create a tutorial, they cannot prevent or help users recover from mistakes.

Automatically generated in-context tutorials present the opportunity to help users locate and correctly interact with the components needed for each step. DocWizards and Sketch-Sketch Revolution both generate in-context tutorials based on events captured during a user interface performance by an expert [1, 10]. In DocWizards, tutorials are presented as a list of steps supplemented with annotations drawn on the interface to highlight relevant interface elements [1]. Sketch-Sketch Revolution presents drawing tutorials through callouts and drawn strokes for the user to trace [10]. Using the recorded event stream, both systems can guide users through exactly recreating the expert's performance. However, because user tracking is done based on a recorded event history, if users deviate from the intended path, either intentionally or by mistake, the tutorials may become less relevant [10] or unable to proceed [1].

While not a tutorial system per se, the Chronicle system uses a recorded workflow to support learning from a document created by other users [16]. As a user creates a document, Chronicle records a workflow history and a video of the performance. When



```
Untitled – Looking Glass
Step 3 of 9 : Go to the Right Now box and Click on an action. Then click Next.
<< Back                                    Next >>
declare procedure myFirstMethod
Do in order
  magician say "Ladies and Gentleman, prepare to be amazed!
  magician say "Abra Kadabra!"  add detail
  magician walk 1.0  add detail
    Zoom out to
    RE-play action to see what it did from 2.82 -- 5.83
    Start my remix here
    End my remix here
  magician turn LEFT , 0.25  add detail
  magician walk 10.0  add detail
Right Now
  magician's 1 action
    magician.walk(1)
```

```
Untitled – Looking Glass
Step 6 of 9 : Go to the Right Now box and Click on an action. Then click Next.
<< Back                                    Next >>
declare procedure myFirstMethod
Do in order
  magician say "Ladies and Gentleman, prepare to be amazed!
  magician say "Abra Kadabra!"  add detail
  magician walk 1.0  add detail
  For each MyBiped character in new MyBiped[] { leftBunny,
    character resize 2.0  add detail
  loop
  magician turn LEFT , 0.25  add detail
  magician walk 10.0  add detail
    Zoom out to
    RE-play action to see what it did from 10.02 -- end of played
    Start my remix here
    End my remix here
Right Now
  magician's 1 action
    magician.walk(10)
```

Figure 2. Marking the (A) beginning and the (B) ending of a code snippet in Looking Glass.

another user interacts with the document later, they can explore how the document was created by replaying video clips that are overlaid on the active interface. Further, a user interested in a particular piece of content can identify the video clips associated with that selected content.

At its core, our tutorial system combines the ability to select a subsection of a document (as in Chronicle) with the ability to generate an in-context tutorial, similar to those created by DocWizards and Sketch-Sketch Revolution. Unlike these three systems, we generate tutorials and track progress based on the underlying application models rather than a recorded event history. This model-based generation and tracking enables greater robustness in the face of mistakes and minor changes in the application's state. For our end users, this translates into a system where a middle schooler can find a program that contains an interesting animation and request a tutorial to learn to build that animation using their own characters and story context.

3. LOOKING GLASS

We implemented our automatically generated programming tutorials in the novice programming environment Looking Glass [28]. Looking Glass is designed to enable middle-school aged children to program by dragging and dropping programming statement tiles to create 3D animated stories. We chose to implement our automatically generated tutorials in Looking Glass because it features an interface for selecting code snippets from shared code [15] found through an online community [18] and because it provides an API for interactive tutorials [19].

3.1 Selecting Snippets from Code

The Looking Glass Community [18] provides an online repository of programs shared by other Looking Glass users. Users can browse the community for motivating programs and later access these programs inside of Looking Glass. Then, users can remix an animation from a shared program by 1) selecting their animation as a code snippet and 2) choosing characters from their own program to perform the actions in that code snippet.

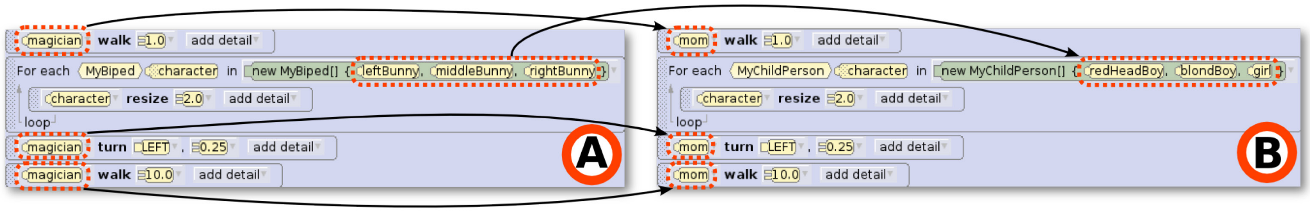
Users select a code snippet using a code selection interface that connects the graphical output of each program statement to the code that caused that output [12, 15]. To select a snippet, users mark the beginning (Figure 2-A) and ending (Figure 2-B) programming statements that bound an animation. After selecting the snippet, the system identifies valid character substitutions for the user to choose from in order to adapt the snippet into the user's current program. Figure 3-A shows the original code snippet and Figure 3-B shows the remixed snippet with character substitutions applied. Following character substitution, the remixed code snippet is automatically copied into the user's program.

3.2 Interactive Tutorials Interface

Looking Glass features an interface for stencils-based tutorials [26] with two presentation styles: on-request and detailed walk-through [19]. The on-request stencils provide high level goals that users can attempt to complete independently. If users do not know how to complete a step, they can request a detailed walk-through of that step. Previous research suggests that asking users to complete a step independently before providing detailed guidance results in a 47% percent improvement on a near transfer task [19].

In the on-request stencils mode, the interface of the application looks normal; however, there is a small on-screen note that provides goal instructions and a button to request additional help. For example, an instruction note might read "Drag and drop a For Each ordering box" (see Figure 4). To complete this step, a user would need to open the control flow constructs tab and then drag and drop a *For Each* programming construct into the code editor. If the user doesn't know how to do this, he or she can request a detailed walk-though version of the same step by clicking the "Show Me How" button shown in Figure 4.

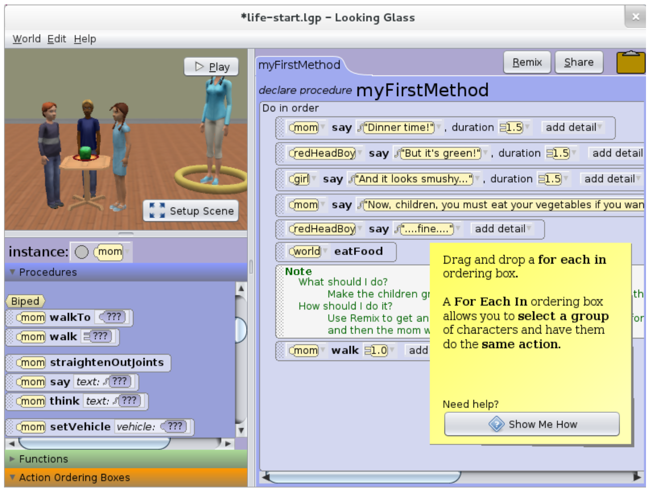
In the detailed walk-through mode, each action necessary to complete a step is presented with a detailed instruction note. For



```
magician walk 1.0  add detail
For each MyBiped character in new MyBiped[] { leftBunny, middleBunny, rightBunny }
  character resize 2.0  add detail
loop
magician turn LEFT , 0.25  add detail
magician walk 10.0  add detail
```

```
mom walk 1.0  add detail
For each MyChildPerson character in new MyChildPerson[] { redHeadBoy, blondBoy, girl }
  character resize 2.0  add detail
loop
mom turn LEFT , 0.25  add detail
mom walk 10.0  add detail
```

Figure 3. The (A) original code snippet from Figure 2 and the (B) remixed code snippet.



```
myFirstMethod
declare procedure myFirstMethod
Do in order
  mom say "Dinner time!" , duration 1.5  add detail
  redHeadBoy say "But it's green!" , duration 1.5  add detail
  girl say "And it looks smushy..." , duration 1.5  add detail
  mom say "Now, children, you must eat your vegetables if you wan
  redHeadBoy say "....fine...."  add detail
  world eatFood
  Note
    What should I do?
      Make the children gr
    How should I do it?
      Use Remix to get an
      and then the mom w
  mom walk 1.0  add
```

```
Biped
mom walkTo ???
mom walk ???
mom straightenOutJoints
mom say text: ???
mom think text: ???
mom setVehicle vehicle: ???
```

Drag and drop a **for each in** ordering box.

A **For Each In** ordering box allows you to **select a group** of characters and have them do the **same action.**

Figure 4. An automatically generated programming tutorial shown in the on-request stencils mode.

example, to insert a *For Each* statement, three actions are necessary: 1) "Select control flow" (an interface tab), 2) "Click and drag a For Each statement" and 3) "Drop here." The notes are presented on a graphical overlay that contains highlighted holes and arrows pointing to the widgets necessary to complete the step as shown in Figure 1. The holes guarantee that the user interacts only with the components needed for each action; all events from widgets without holes are intercepted.

4. GENERATING TUTORIALS

To support users learning from shared code, we generate a tutorial for every remix. Now, when users remix, instead of copying remixed snippets directly into users' programs, we generate tutorials that guide users through re-constructing the snippets within their own programs. Our process for generating tutorials requires two phases: 1) generating a draft tutorial of the code snippet and 2) advancing the tutorial based on the user's progress.

4.1 Generating the Draft Tutorial

We generate each tutorial based on a remixed code snippet. Fundamentally, this process requires translating the selected code snippet into a sequence of actions users can take in order to re-create that snippet in Looking Glass. To facilitate this, we have made two fundamental changes to Looking Glass: 1) the code base follows a model-driven architecture and 2) each program statement in a code snippet knows the model responsible for its creation. In our model-driven architecture, each model is responsible for creating and tracking the state for all of its widgets. Together, these changes enable the system to identify the widgets needed to re-create each statement in a code snippet.

For each programming statement in the remixed code snippet, we generate a draft tutorial step from each statement's model. Suppose that a user has requested a tutorial from a code snippet that requires adding a *Count Loop*. Because the *Count Loop* programming statement knows that it was created by the Count Loop Model, and since the model tracks its widgets, we can find the *Count Loop* widget in the active interface (see Figure 5). However, we need to be able to go one step further. In Figure 5, the control flow tab happens to already be selected. If the tab was not selected, the tutorial would need to first tell the user to change the selected tab to the control flow tab. We call this relationship a model dependency: in order for the user to access one model's widgets, another model must be in a particular state. In this case, the Count Loop Model depends on the Tab Model.

Because we cannot know the states of a model's dependencies in advance, we create a draft tutorial of the code snippet. Each statement in the code snippet becomes a step in the draft tutorial. We do not create any steps for model dependencies during this phase. Later, when we present the tutorial to the user, we check the model dependencies for each step and insert prerequisite steps when necessary to satisfy any dependencies.

4.2 Advancing Through the Tutorial

Once we have produced the draft tutorial we can begin the process of presenting the tutorial to the user. Advancing through the tutorial requires three separate processes: 1) verifying each step's dependencies and inserting prerequisite steps, 2) presenting the steps to the user, and 3) validating the user's progress. The algorithm we use to advance through the tutorial only processes each step in the tutorial when it is time to present that step to the user. This ensures that if a previous step causes the current step's dependencies to go unsatisfied we can adjust by inserting new steps to fulfill these dependencies. We use this algorithm to advance through the tutorial:

```
For each draft tutorial step do:
   If the step's dependencies are satisfied then:
      Present the step to the user.
      Validate the user's progress.
      Advance to the next step.
   Else:
        Create and insert a prerequisite step.
```

4.2.1 *Satisfying Model Dependencies and Inserting Prerequisite Steps*

When a tutorial step is ready to be presented to the user, we query that step's model and verify that all of its dependencies have been satisfied. If the dependencies are satisfied, there is at least one way to create the step's programming statement using the widgets currently visible to the user. In our Figure 5 example, the Count Loop Model's dependencies are met if the tab's current selection is the Control Flow Tab.

If the current state of the interface does not have a widget visible for the model, then its dependencies are not satisfied. In this case we insert prerequisite steps that will change the state of the interface so that a widget is visible to the user. To do this, we use our model-driven architecture to discover a step's dependencies. We first query the current step's model for its dependent models. We then ask the dependent models to create prerequisite steps and insert these steps before the current step. In our example (see Figure 5), suppose that the methods tab is selected and, therefore, the Count Loop Model's dependencies are not satisfied. In this case, we would ask the tab model to generate a new step that

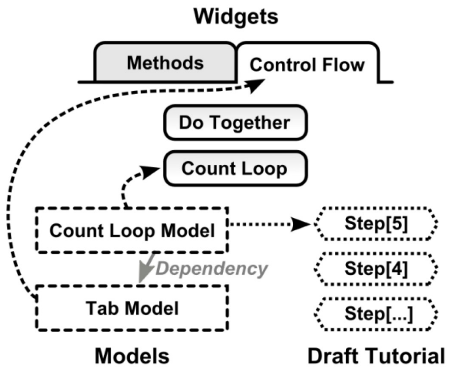


Figure 5. Generating a draft tutorial.

guides the user to select the Control Flow Tab. We then insert this prerequisite step before the *Count Loop* step.

4.2.2 Presenting Steps

Once a step's dependencies have been satisfied, we initialize the stencils-based interface for the step using one of the model's on-screen widgets. We then create a hole in the stencils graphical overlay where that widget resides and query the model to generate the instructions for the step's notes. To help users learn new programming constructs, we append an explanation of any control flow constructs within the note's generated instructions as shown in Figure 4.

4.2.3 Tracking Progress

To validate a user's progress, we need to 1) track the changes the user has made through completing the tutorial and 2) compare the recorded changes to the changes required by the current step.

When a user performs any action in Looking Glass, that action is recorded by its model as a transaction. Each transaction stores the model that created it, the current state of that model, and the states of the model's dependencies. All transactions are stored in a history that we can query when determining the correctness of the current step. Figure 6 shows a user working through the draft tutorial created in our Figure 5 example. In Figure 6, the user first selected the control flow tab and then used the *Count* loop. Both actions are recorded as transactions in the transaction history.

While the stencils-based interface can prevent incorrect clicks in a tutorial, it cannot ensure that a step is completed correctly. In order to ensure correctness and support auto advancing, we compare the user's current actions to the current tutorial step. We check to see if the latest recorded transaction exactly matches the change made in the current step. If the transaction and step changes match, we automatically advance to the next step. This removes any uncertainty about whether the current step was completed correctly. Otherwise, we roll back the latest transaction. By validating the user's progress, we prevent accidental mistakes from derailing the tutorial.

5. EVALUATION

We hypothesized that automatically generated tutorials for selected code snippets could increase learning gains for unfamiliar programming concepts. To evaluate this, we conducted a between-subjects experiment comparing the performance of participants who only remixed and participants who remixed and then completed a tutorial on a near transfer task. During the experiment, participants in the control condition remixed three different programs, each containing a different programming construct. Participants in the experimental condition remixed the same programs, but each remix was followed by a generated tutorial that guided the participant through reconstructing the remixed code snippet. To evaluate their learning gains, we asked participants in both groups to complete a transfer program following each remix.

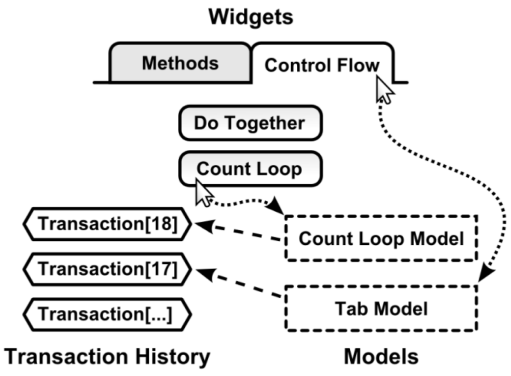


Figure 6. Tracking the user's progress in a tutorial.

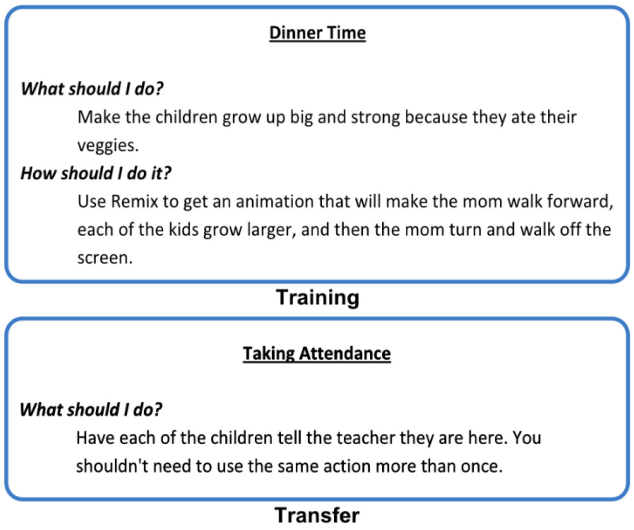
5.1 Participants

We recruited our participants for this study from the Academy of Science of St. Louis mailing list. The Academy of Science is a not-for-profit organization dedicated to scientific outreach. We pre-screened 43 participants prior to their arrival to the study in an attempt to ensure they had no prior programming experience. On arrival, we again questioned each participant about their computer programming experience and discovered that three participants did have past programming experience. We have excluded the three users with prior programming experience from our results. The remaining 40 participants (23 female, 17 male) ranged in age from 10 to 16 years ($\mu = 12.29, \sigma = 1.75$). We gave each participant a \$10.00 gift certificate to iTunes or Amazon.com in recognition of their participation.

5.2 Materials

To test learning gains for unfamiliar programming constructs, we created three tasks that each introduced a different programming construct. We designed the tasks to introduce three control flow constructs of varying levels of difficulty within Looking Glass: *Do Together* (parallelism), *Count Loop* (loop *N* iterations), and *For Each* (iterate over an array). We chose these constructs because, based on our prior experience with Looking Glass users, many new programmers have difficulty learning these constructs independently.

We divided each task into a training phase and a transfer phase. For the training phase we created two programs: 1) a *snippet selection* program that is 2) *remixed into* another program. The *snippet selection* program contained one instance of the programming construct specific to the task, while the *remix into* program contained only simple sequential statements. For example, in the *For Each* task the *remix into* program is a story about a mother trying to get her three kids to eat their vegetables. The related *snippet selection* program features a magician demonstrating a trick that makes three rabbits grow extremely large using the *For Each* construct. When completed through remixing, the final program shows the three kids growing large because they ate their vegetables.


Dinner Time

What should I do?

Make the children grow up big and strong because they ate their veggies.

How should I do it?

Use Remix to get an animation that will make the mom walk forward, each of the kids grow larger, and then the mom turn and walk off the screen.

Training

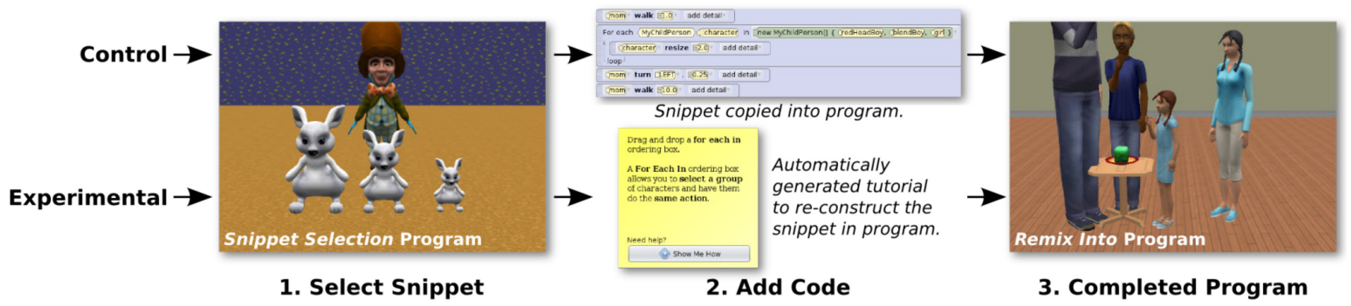
Taking Attendance

What should I do?

Have each of the children tell the teacher they are here. You shouldn't need to use the same action more than once.

Transfer

Figure 7. Task cards for the *For Each* task.


Control
Experimental
Snippet Selection Program
Snippet copied into program.
Automatically generated tutorial to re-construct the snippet in program.
Remix Into Program
1. Select Snippet
2. Add Code
3. Completed Program

```
mom walk 1.0 add detail
For each MyChildPerson character in new MyChildPerson[] { redHeadBoy, blondBoy, girl }
    character resize 2.0 add detail
loop
mom turn LEFT , 0.25 add detail
mom walk 10.0 add detail
```

Drag and drop a **for each in** ordering box.

A **For Each In** ordering box allows you to **select a group** of characters and have them do the **same action.**

Need help?

Show Me How

Figure 8. The *For Each* task's training phase.

For the transfer phase we created a transfer program similar to the *remix into* program used during the training phase. We designed the transfer program to require the user to use the programming construct from the training phase to complete the program. For example, the transfer program for the *For Each* task depicts a teacher taking attendance for three students. This program is completed by using the *For Each* construct to make each student say "here." To provide consistent directions to all participants, we created task cards which outlined the requirements for each phase of every task. The task cards for the *For Each* task are shown in Figure 7.

5.3 Study Procedure

The study consisted of a series of one-time, 1.5 hour sessions with no more than five participants. At the beginning of each session, we randomly assigned participants to the control or experimental condition. Due to the excluded users, the control group contained 21 participants whereas the experimental group contained 19. To balance potential learning effects, we ordered the three tasks using a Latin squares design to alternate the order in which the tasks were presented to each participant. Participants were positioned in the same room, but they could only see their own computer monitors and not those of the other participants. At the beginning of each evaluation session we asked participants to complete a demographics survey. Afterwards, we gave an overview and demonstration of the remix process. Following the demonstration, we asked participants to complete the three tasks. During the training phase, participants used remixing to complete a program. In the transfer phase, we asked participants to finish a transfer program independently. We concluded the study with an attitude survey.

5.3.1 Demonstration

To familiarize participants with the mechanics of remixing, we began with a simple demonstration of the remix process. We designed the demonstration to be functionally identical to the training phase. For the demonstration, we created a *remix into* program where the goal was to make a bunny hop by remixing an animation of a girl jumping from the *snippet selection* program. The demonstration programs contained only simple sequential execution and no tutorial. During the demonstration we showed participants how to select the beginning and ending snippet statements and how to make their character selections using the code selection interface. We provided no other instruction to the participants.

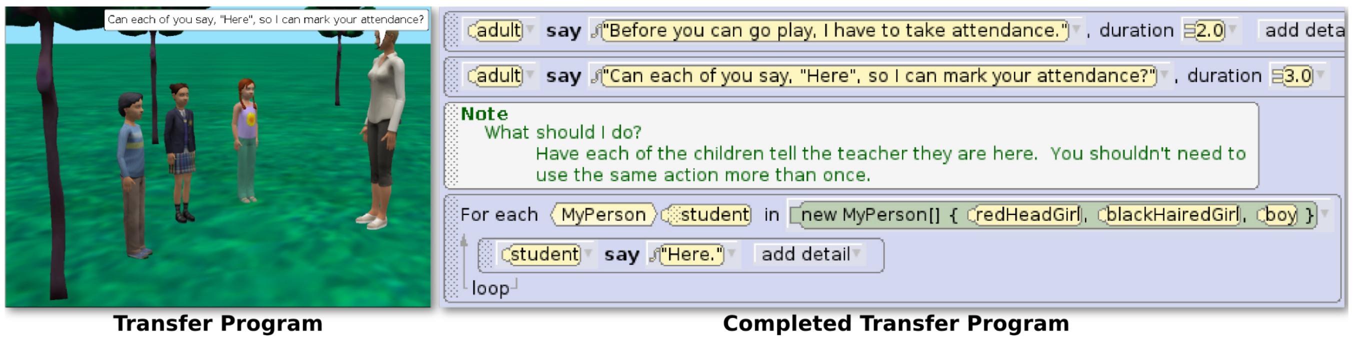
5.3.2 Training Phase

For each task, we began with the training phase as shown in Figure 8. We asked the participant to carefully read the task card for this phase. Afterwards, we opened and played the *remix into* program. Next, each participant watched the *snippet selection* program. When finished watching the *snippet selection* program, we instructed the participant to begin and reminded him or her to use only remixing to complete the task. For the control condition participants, the system automatically copied their selected code snippet into the *remix into* program. In the experimental condition, each participant completed a generated tutorial to re-construct the remixed snippet in the *remix into* program.

5.3.3 Transfer Phase

During the transfer phase, each participant attempted to finish a transfer program requiring the use of the programming construct introduced during the training phase of that task as shown in Figure 9. To begin, we asked the participant to read the task card for the transfer phase. Each participant then watched the unfinished transfer program. We then instructed the participant to finish the transfer program independently without remixing.

In our pilot study, we observed that participants who made little or no progress on the first task seemed to disengage for the second and third tasks. To prevent struggling users from disengaging, we adopted a policy of providing one specific hint for the transfer phase. If, after five minutes from starting the transfer phase, the participant did not have the correct programming construct added to their transfer program, we provided the *control flow tab (CFT) hint*: the researcher pointed to the control flow constructs tab and told the participant "To complete this task, look here." In Looking Glass, the control flow constructs are located in a tab pane, which may not be immediately obvious to new users. While this policy


Can each of you say, "Here", so I can mark your attendance?
Transfer Program
Completed Transfer Program

```
adult say "Before you can go play, I have to take attendance." , duration 2.0 add deta
adult say "Can each of you say, "Here", so I can mark your attendance?" , duration 3.0
Note
  What should I do?
      Have each of the children tell the teacher they are here.  You shouldn't need to
      use the same action more than once.
For each MyPerson student in new MyPerson[] { redHeadGirl, blackHairedGirl, boy }
    student say "Here." add detail
loop
```

Figure 9. The *For Each* task's transfer phase.

Table 1. Example questions and Cronbach's alpha for each of the four subscales in the TEQ

Scale	Example Question	Cronbach's Alpha
Interest/ Enjoyment	I enjoyed making my story in Looking Glass very much.	0.9248
Perceived Competence	I think I did pretty well at making my story in Looking Glass, compared to others.	0.7876
Perceived Choice	I felt like I had to make my story in Looking Glass. [reversed item]	0.6079
Pressure/ Tension	I felt pressured while making my story in Looking Glass.	0.8219

creates the potential for a learning effect, we felt that the risk of participants disengaging was a more significant problem. Our observations in the pilot study suggested that the hint did not provide a noticeable learning advantage. If a participant was not finished with the transfer phase after ten minutes, we recorded this phase as incomplete and moved the participant onto the next task.

5.4 Data

We collected a short demographic survey, the remix and transfer programs, and an attitude survey.

5.4.1 Remix Programs

We checked the participants' *remix into* programs from the training phase for completeness and recorded whether the programs contained the appropriate programming construct exactly as it appeared in each *snippet selection* program. If the participant failed to correctly select the beginning and ending of the code snippet during remixing they may not have been exposed to the programming construct required for the transfer program.

5.4.2 Transfer Programs

We graded the transfer programs for correctness. We developed these criteria by analyzing the transfer programs from the pilot study. Each criterion is worth one point.

We graded the *Do Together* transfer program using the following criteria. (4 points)

1. Program contains a Do Together construct. If not, stop grading.
2. Do Together contains at least two statements. If not, stop grading.
3. Correct characters in programming statements.
4. Animation is correct.

We graded the *Count* transfer program using the following criteria. (5 points)

1. Program contains a Count construct. If not, stop grading.
2. Count contains at least one statement. If not, stop grading.
3. Count index is correct.
4. Correct characters in programming statements.
5. Animation is correct.

We graded the *For Each* transfer program using the following criteria. (5 points)

1. Program contains a For Each construct. If not, stop grading.
2. For Each contains at least one statement. If not, stop grading.
3. Array is defined correctly for the animation.
4. Programming statements use the loop iterator.
5. Animation is correct.

Table 2. The percentage of correct remixes and average time needed for the training phase.

	Do Together		Count		For Each	
Control/Experimental	**C**	**E**	**C**	**E**	**C**	**E**
Correct Remixes (%)	85.7	84.2	81.0	89.5	100.0	89.5
Average Time (min.)	3.89	7.46	4.29	5.75	5.01	9.14

5.4.3 Attitude Survey

We used the Intrinsic Motivation Inventory's (IMI) Task Evaluation Questionnaire (TEQ) [24] to evaluate participants' experiences during the evaluation. The TEQ is a standardized, shortened version of the 45 item IMI. The TEQ includes 22 items that represent four subscales: interest/enjoyment, perceived competence, perceived choice and pressure/tension [24]. For all questions, participants rate their agreement using a Likert scale ranging from 1 (not at all true) to 7 (very true). To ensure that the four subscales were valid for our data, we measured the internal consistency of each subscale using Cronbach's alpha. Table 1 shows an example question and our computed Cronbach's alpha for all four subscales. The Cronbach's alpha values for interest/enjoyment, perceived competence, and pressure/tension are acceptable. The perceived choice Cronbach's alpha is below the generally accepted level for reliability (0.7), so we have chosen not to analyze the results from this subscale.

6. RESULTS

To provide insight into the impact of automatically generated tutorials on users' success with and experience while learning new programming constructs, we explore two kinds of data: task performance data and attitudinal survey data.

6.1 Task Performance Results

Participants in the experimental condition took longer to complete the training phase than participants in the control condition. This is not surprising given that the experimental conditional ended the remix process with a tutorial. Participants in the experimental condition took an average of 22.34 minutes to complete all the remixes and tutorials as compared to 13.19 for control participants. This difference in training time is significant ($F[1,38] = 29.37, p < 0.001$). See Table 2 for additional data from the training phase.

We found no significant difference between the amount of time participants in the control and experimental conditions needed to complete the transfer programs ($F[1,38] = 1.30, p = 0.26$). See Table 3 for the transfer phase's average completion times. Figure 10 shows the average performance scores for each of the transfer programs. We compared the performance of participants in the experimental and control conditions using ANCOVA with the presence or absence of the *CFT* hint as a covariate. Participants in the experimental condition averaged 1.56 correct transfer programs versus 0.95 for control participants, a 64% improvement ($F[2,37], p < 0.05$). There was a significant main effect for

Table 3. The percentage of *CFT* hints, completed transfer programs, and average time needed for the transfer phase.

	Do Together		Count		For Each	
Control/Experimental	**C**	**E**	**C**	**E**	**C**	**E**
***CFT* Hint (%)**	33.3	31.6	52.4	57.9	52.4	26.3
Average Time (min.)	5.85	5.74	6.11	5.96	4.57	4.39

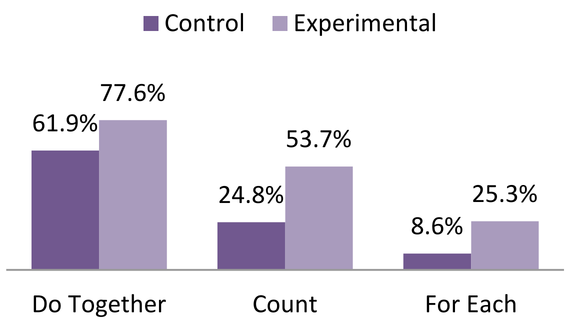


Figure 10. Average transfer program scores for each task.

condition ($p < 0.05$) but not for the *CFT* hint ($p = 0.48$). This suggests that our decision to direct users to the available control flow constructs tab did not significantly impact their performance. The transfer program results suggest that automatically generated tutorials, combined with the code selection process, enhanced learning gains.

6.2 Attitude Survey Results

We analyzed the three IMI subscales with acceptable reliability: interest/enjoyment, perceived competence, and pressure/tension. There were no statistically significant differences between the control and experimental groups for the interest/enjoyment scale ($F[1,38 = 0.06, p = 0.81)$ or the perceived competence scale ($F[1,38] = 0.617, p = 0.44$). Although it was not significant, participants in the experimental condition tended to report feeling more relaxed in working with Looking Glass than participants in the control group (experimental participants averaged -1.46 versus -0.71 for control participants, $F[1,38] = 3.13, p = 0.085$). Table 4 shows the correlations between participants' scores on the transfer programs and the three TEQ subscales. Additionally, there is a strong correlation ($p < 0.0001$) between Interest/Enjoyment and Perceived Competence. In other words, participants who felt that they were more skilled at working with Looking Glass tended to also report greater enjoyment in working with the system.

7. DISCUSSION AND LIMITATIONS

One possible explanation for the learning gains demonstrated by participants in the experimental condition is the increase in time on task. In designing the study, we elected to compare the learning resulting from a single remix. It is possible that asking users in the control condition to complete multiple remixes could result in similar learning gains. However, we think this is unlikely because the process of remixing requires only that users identify the beginning and end of the target functionality. Users could easily remix additional behaviors without needing to understand all of the code elements between the beginning and ending remix markers. In contrast, rebuilding the selected code through completing the tutorial introduces users to each of the code elements from the snippet.

Ultimately, the automatic generation of tutorials creates the potential for users to follow their own interests. However, in this study we asked users to complete a set of specified tasks in order to measure the potential learning gains from tutorials rather than remixing alone. In this context, the introduction of tutorials creates the potential for the experience of working with Looking Glass to feel more like homework. The similarity in the attitude scores between the control and experimental groups suggests that the introduction of tutorials did not negatively impact user experience. Additional studies are needed to explore long term learning and how users incorporate remixing and tutorials into the pursuit of their own projects.

Further, despite the constrained structure of our study, it provided a gratifying preview of the playful feel that this style of learning may ultimately engender. Several participants commented that they enjoyed the puzzle-like quality of the remix process. And after remixing and completing a tutorial, several participants began modifying and personalizing the program. The fact that participants enjoyed the process of searching for actions to learn from and wanted to continue programming after rebuilding those actions via a tutorial provides support for our approach.

Table 4. Attitude survey correlations among continuous variables. (*$p < .05$, **$p < .0001$)

	Task Score	Interest/ Enjoyment	Perceived Competence	Pressure/ Tension
Task Score	1.00	-0.10	0.27	-0.64*
Interest/ Enjoyment		1.00	0.67**	-0.12
Perceived Competence			1.00	-0.33*
Pressure/ Tension				1.00

8. CONCLUSION AND FUTURE WORK

In this paper, we have described a method for automatically generating and advancing interactive, in-context programming tutorials and demonstrated that the use of these tutorials resulted in learning improvements for unfamiliar programming constructs. Particularly for learners who have limited or no access to computer science learning opportunities within their own communities, the ability to create tutorials from content of interest may encourage and enable a broader range of learners to explore computing.

The ability to automatically generate tutorials already opens the door for users to personalize their own learning trajectories. However, much more is possible. Currently, given a code snippet, we generate the same tutorial for all users. By adding a user model that can track the programming constructs users have already experienced, we can customize the steps and explanations given. If a user is seeing a construct for the first time, the tutorial might include more detailed explanations as well as additional steps to demonstrate the behavior of that construct. When presenting a familiar construct, the tutorial might present a higher level goal that requires learners to synthesize multiple familiar ideas and steps. We plan to explore how to best incorporate knowledge of a learner's history to maximize learning gains.

9. ACKNOWLEDGEMENTS

This work was funded by NSF grant #1054587. We would like to thank the Academy of Science in St. Louis for their help with participant recruitment.

10. REFERENCES

[1] Bergman, L. et al. 2005. DocWizards: a system for authoring follow-me documentation wizards. *Proc. UIST*, 191–200.

[2] Bhardwaj, A.P. et al. 2011. Redprint: integrating API specific "instant example" and "instant documentation" display interface in IDEs. *Proc. UIST*, 21–22.

[3] Brandt, J. et al. 2010. Example-centric programming: integrating web search into the development environment. *Proc. CHI*, 513–522.

[4] Brandt, J. et al. 2009. Two studies of opportunistic programming: interleaving web foraging, learning, and writing code. *Proc. CHI*, 1589–1598.

[5] Carroll, J.M. et al. 1987. The minimal manual. *Hum.-Comput. Interact.* 3, 2 (1987), 123–153.

[6] Carroll, J.M. and Rosson, M.B. 1987. *Paradox of the Active User*.

[7] Chi, P.-Y. et al. 2012. MixT: automatic generation of step-by-step mixed media tutorials. *Proc. CHI EA,* 1499–1504.

[8] Coachmarks: *http://www.developer.apple.com/techpubs/mac/AppleGuide/AppleGuide-24.html*.

[9] Dorn, B. and Guzdial, M. 2006. Graphic designers who program as informal computer science learners. *Proc. ICER*, 127–134.

[10] Fernquist, J. et al. 2011. Sketch-sketch revolution: an engaging tutorial system for guided sketching and application learning. *Proc. UIST*, 373–382.

[11] Grabler, F. et al. 2009. Generating photo manipulation tutorials by demonstration. *Proc. SIGGRAPH*, 1–9.

[12] Gross, P. et al. 2011. Dinah: an interface to assist non-programmers with selecting program code causing graphical output. *Proc. CHI*, 3397–3400.

[13] Gross, P. and Kelleher, C. 2010. Non-programmers identifying functionality in unfamiliar code: strategies and barriers. *Journal of Visual Languages & Computing*. 21, 5 (Dec. 2010), 263–276.

[14] Gross, P. and Kelleher, C. 2010. Toward transforming freely available source code into usable learning materials for end-users. *Proc. PLATEAU*, 6:1–6:6.

[15] Gross, P.A. et al. 2010. A code reuse interface for non-programmer middle school students. *Proc. IUI*, 219–228.

[16] Grossman, T. et al. 2010. Chronicle: capture, exploration, and playback of document workflow histories. *Proc. UIST*, 143–152.

[17] Grossman, T. and Fitzmaurice, G. 2010. ToolClips: an investigation of contextual video assistance for functionality understanding. *Proc. CHI*, 1515–1524.

[18] Harms, K.J. et al. 2012. Designing a community to support long-term interest in programming for middle school children. *Proc. IDC*, 304–307.

[19] Harms, K.J. et al. 2011. Improving learning transfer from stencils-based tutorials. *Proc. IDC*, 157–160.

[20] Hartmann, B. et al. 2007. Programming by a sample: rapidly creating web applications with d.mix. *Proc. UIST*, 241–250.

[21] Hartmann, B. and Dhillon, M. 2010. HyperSource: bridging the gap between source and code-related web sites. *Adjunct proc. UIST*, 421–422.

[22] Huang, J. and Twidale, M.B. 2007. Graphstract: minimal graphical help for computers. *Proc. UIST*, 203–212.

[23] IEEE Job Site: *http://careers.ieee.org/article/European_Job_Outlook_0312.php*. Accessed: 2013-01-22.

[24] Intrinsic Motivation Inventory: *http://www.selfdeterminationtheory.org/questionnaires/10-questionnaires/50*. Accessed: 2012-09-16.

[25] Kelleher, C. et al. 2007. Storytelling alice motivates middle school girls to learn computer programming. *Proc. CHI*, 1455–1464.

[26] Kelleher, C. and Pausch, R. 2005. Stencils-based tutorials: design and evaluation. *Proc. CHI*, 541–550.

[27] Knabe, K. 1995. Apple guide: a case study in user-aided design of online help. *Proc. CHI*, 286–287.

[28] Looking Glass: *http://lookingglass.wustl.edu*.

[29] MacLaurin, M. 2009. Kodu: end-user programming and design for games. *Proc. FDG*, xviii–xix.

[30] Nakamura, T. and Igarashi, T. 2008. An application-independent system for visualizing user operation history. *Proc. UIST*, 23–32.

[31] Oney, S. and Brandt, J. 2012. Codelets: linking interactive documentation and example code in the editor. *Proc. CHI*, 2697–2706.

[32] Palmiter, S. et al. 1991. Animated demonstrations vs. written instructions for learning procedural tasks: a preliminary investigation. *Int. J. Man-Mach. Stud.* 34, 5 (1991), 687–701.

[33] Papert, S. 1980. *Mindstorms: children, computers, and powerful ideas*. Basic Books, Inc.

[34] Pryor, J.H. et al. 2010. *The American Freshman: National Norms for Fall 2009*.

[35] Resnick, M. et al. 2009. Scratch: programming for all. *Commun. ACM*. 52, 11 (2009), 60–67.

[36] Rosson, M.B. et al. 2005. Who, What, and How: A Survey of Informal and Professional Web Developers. *Proc. VL/HCC*, 199–206.

[37] Ying Zhang et al. 2009. SmartTutor: Creating IDE-based interactive tutorials via editable replay. *Proc. ICSE*, 559–562.

[38] 2011. *Computing Education and Future Jobs: A Look at National, State, and Congressional District Data*. National Center for Women & Informaton Technology.

[39] 1987. *Gender Differences on the California Statewide Assessment of Attitudes and Achievement in Science*. Proceedings of the Annual Meeting of the American Educational Research Association.

[40] *Taulbee Survey Report 2010-2011*. Computing Research Association.

[41] 2000. *Tech-Savvy: Educating Girls in the New Computer Age*. American Association of University Women Educational Foundation.