Motivating Programming: using storytelling to make computer programming attractive to middle school girls

Caitlin Kelleher

November, 2006
CMU-CS-06-171

School of Computer Science
Carnegie Mellon University
Pittsburgh, PA 15213

Submitted in partial fulfillment of the requirements
for the degree of Doctor of Philosophy.

Also appears as Human Computer Interaction Institute Technical Report
CMU-HCII-06-110

Thesis Committee:
Randy Pausch (chair)
Jessica Hodgins
Sara Kiesler
Alan Kay, Viewpoints Research

This research was sponsored by the Office of Naval Research under grant no. N00140210439 and by the National Science Foundation (NSF) under grant nos. IIS-0329090, DUE-0339734, IIS-0121629, IIS-9812012 and a generous NSF Graduate Fellowship. The views and conclusions contained in this document are those of the author and should not be interpreted as representing the official policies, either expressed or implied, of any sponsoring institution, the U.S. government or any other entity.

Keywords: programming environments, gender, computer science education, motivation, storytelling, Alice, Stencils

Abstract

Women are currently under-represented in computer science. Increasing the numbers of female students who pursue computer science has the potential both to improve the technology we create by diversifying the viewpoints that influence technology design and to help fill projected computing jobs. Numerous studies have found that girls begin to turn away from math and science related disciplines, including computer science, during middle school. By the end of eighth grade, twice as many boys as girls are interested in pursuing science, engineering, or technology based careers.

In this thesis, I describe Storytelling Alice, a programming environment that gives middle school girls a positive first experience with computer programming. Rather than presenting programming as an end in itself, Storytelling Alice presents programming as a means to the end of storytelling, an motivating activity for a broad spectrum of middle school girls. The development of Storytelling Alice was informed by formative user testing with more than 250 middle school aged girls. To determine girls' storytelling needs, I observed girls interacting with Storytelling Alice and analyzed their storyboards and the story programs they developed. To enable and encourage middle school girls to create the kinds of stories they envision, Storytelling Alice includes high-level animations that enable social interaction between characters, a gallery of 3D objects designed to spark story ideas, and a story-based tutorial presented using Stencils, a new tutorial interaction technique.

To determine the impact of the storytelling focus on girls' interest in and success at learning to program, I conducted a study comparing the experiences of girls introduced to programming using Storytelling Alice with those of girls introduced to programming using a version of Alice without storytelling features (Generic Alice). Participants who used Storytelling Alice and Generic Alice were equally

successful at learning basic programming concepts. However, I found that users of Storytelling Alice show more evidence of engagement with programming. Storytelling Alice users spent 42% more time programming and were more than three times as likely to sneak extra time to continue working on their programs (51% of Storytelling Alice users vs. 16% of Generic Alice users snuck extra time).

Acknowledgements

First, I would like to thank my advisor, Randy Pausch, for his frank thoughts and tireless support. He gave me a tremendous amount of freedom to explore what is fundamentally a muddy area while continually encouraging me to find ways to objectively measure success. My committee members Jessica Hodgins, Sara Kiesler, and Alan Kay provided valuable feedback that guided both my research and my presentation of it.

Several organizations helped me to find middle school girls for both the formative and summative evaluations. Thanks to the Carnegie Museum of Natural History, Georgia Tech's TEC camp, the Houston Museum of Natural Science, the PALS homeschooling group, NPHEP, and the Girl Scouts of Western Pennsylvania. I also owe a large debt of gratitude to the many girls (and their parents) who participated in the development and testing of Storytelling Alice.

The undergraduates who participated in the Designing Interactive Narrative Components helped me begin to understand the space of inspiring kids' story ideas. I continued to learn from their work long after the end of the seminar.

I would like to thank Dennis Cosgrove, Adam Fass, Andrew Faulring, Desney Tan, Gabriel Yu, and all of the other Stage 3 folk I have had the good fortune to work and play with. Finally, I am grateful for the support of my parents, William and Denise Kelleher, and my sister Erin.

Contents

List of Figures and Tables

Chapter 1 Introduction

Chapter 2 Introduction to Alice 2

Chapter 3 Formative User Testing

Chapter 4 Enabling Storytelling

Chapter 6 Developing the Storytelling Tutorial

Chapter 7 Evaluation Methodology

Chapter 8 Summative Evaluation Results

Chapter 9 What Girls Create

Chapter 10 Conclusions and Future Work

Chapter 11 Programming Languages and Environments for Novice Programmers

Chapter 1 Introduction

1.1 Introduction

In my thesis work, I have developed a programming system called Storytelling Alice for middle school girls that presents computer programming as a means to the end of storytelling. The development of Storytelling Alice was guided by formative testing with more than 200 girls over a two-year period. The formative testing took place in a variety of formats ranging from 4 hour afternoon workshops to week-long camps with groups of 3 to 20 girls ranging in age from 10 to 17. Participants were recruited from technology camps, home-schooling groups, and the Girl Scouts. During formative testing, girls created storyboards of movies they wanted to create and then tried to implement them in a version of Storytelling Alice. Storytelling Alice includes three types of supports to enable users to create stories: 1) high-level animations that support the use of social characters who can interact with one another 2) a gallery of characters and scene elements that helps girls find story ideas, and 3) a story-based tutorial. Storytelling Alice is based on Alice 2.0, which provides the ability to render 3D animations and a drag and drop interface for constructing programs.

To evaluate the impact of the storytelling focus on girls' motivation to learn and success at learning computer programming, I did a study comparing girls' experiences and behavior using Storytelling Alice and a version of Alice without storytelling support

(Generic Alice). The study took place during a series of one-time, four-hour workshops. Participants were assigned to use either Storytelling Alice or Generic Alice. I collected participants' Alice programs, logs of their actions within Alice, survey responses, quiz performance, participants' workshop behavior. Participants who used Storytelling Alice and Generic Alice were equally successful at learning programming concepts. However, participants who used Storytelling Alice showed more evidence of engagement: they spent more time programming, were more likely to sneak extra time to work on their programs and have a stronger interest in using Alice in the future.

The ability to motivate middle school girls to learn computer programming may encourage more women who choose to pursue computer science. The field of Computer science has a long-standing problem in attracting women. The participation of women in computer science peaked in 1985 when more than 35% percent of CS bachelor's degrees were awarded to women (Vegso 2005). Since that time, the number CS degrees awarded to women has dropped. In 2004, fewer than 20% of CS degrees granted by research universities were awarded to women (Vegso 2005). Several studies have shown that girls begin to turn away from math and science during middle school (AAUW 1998; CAWMSET 2000). A positive first experience with computer programming during middle school may help to increase the number of girls who pursue computer science.

1.2 A Pragmatic Need for Diversity in Computer Science

Despite wide-spread usage of computers-based technologies, only a small, unrepresentative sample of the population is involved in creating new technologies. Broadening and diversifying the group of people who create new computer-based technologies has two potential benefits: 1) a larger, more diverse group will help ensure that computer science attracts the talent that the discipline needs and 2) a more diverse group of people involved in the design of new technologies will help to ensure that new technologies meet the needs of our diverse society.

Advances in computer science enable progress across many disciplines including fields as diverse medicine, education, and predicting natural disasters. Given the broad impact of computer science, it is critical that we ensure that computer science continues to attract

bright minds that will enable the field to continue to make forward progress and support progress in other fields. Recently, there has been a dramatic drop in the numbers of students interested in studying computer science at both the college and high school levels. In the period between 2000 and 2004, the number of college freshman who listed computer science as their probable major dropped by 60% and computer science enrollments at research universities dropped by 39% (Vegso 2005). There is a similar loss of interest in computer science at the high school level. In the year between 2004 and 2005 alone, the number of students who took an AP computer science exam (either the Computer Science A or the Computer Science AB exam) dropped by nearly 6% (College_Board 2004; College_Board 2005). Further, computer science was the **only** AP subject area that saw a decrease in student participation (College_Board 2004; College_Board 2005). Although it is difficult to accurately predict future job openings, decreasing student interest will inevitably result in a smaller selection pool for the future leaders of computer science.

In addition to the need to increase the number of people who enter computer science we need to increase the diversity of people who choose to pursue computer science. Currently, women are under-represented in computer science. According to the 2004 Taulbee survey, 82.3% of bachelor's degrees in computer science were awarded to men. Increasing the diversity of viewpoints in computer science may help to ensure that we design new technologies that meet the needs of our diverse society. Today, technologies created by computer scientists touch the daily lives of a broad segment of our population. Technologies designed by an unrepresentative group may be less likely to take everyone's needs into account. For example, early voice recognition and video conferencing systems did not recognize women's voices (Margolis and Fisher 2002). The failure to recognize women's voices is likely the result of the voice recognition and video conferencing teams testing their programs in-house with their male-colleagues. A more diverse design team decreases the likelihood that this kind of scenario will occur. Further, the problems that we choose to solve and the technologies that we create inevitably reflect our personal beliefs about what kinds of problems are important and how they should be solved. For example, the parent of an autistic child is much more likely to think

about ways that technology can support autistic children and their families than someone who has no experience with autism. As technology continues to become an integral part of daily life, involving a representative sample of people in the design of new technologies can help ensure that our technologies meet everyone's needs.

There is some evidence suggesting that men and women would tend to design different kinds of technologies. A study of 47 preadolescent boys and girls showed that when they were asked to design their "dream" technology, they tended to describe very different things. Boys often described vehicles that could take them anywhere whereas girls often described objects that could help in everyday life (Brunner, Bennett et al. 1998). Similar differences were seen among 24 adult technology users, balanced for gender and profession. The men tended to fantasize about bionic mind implants that grant god-like powers whereas the women in the study tended to fantasize about small flexible technologies that help people stay in touch and adapt to the wearers' current needs (Brunner, Bennett et al. 1998). Because men and women appear to envision different future technologies, it seems likely that men and women will tend to push technology in different directions.

Learning to program is also a valuable part of a general education for all students. In addition to being a nice introduction to structured problem solving, programming also gives students experience with complex systems and provides students with computational thinking skills that can be applied to a broad range of disciplines ranging from Biology to Economics (Wing 2006). The world around us is filled with complex systems whose behavior depends on the behaviors and interactions of smaller parts within the system: cars, weather, and manufacturing plants, to name just a few. Yet our schools do little to prepare students to reason about complex systems. When your car breaks, it is helpful to be able to read about the main components of car engines and eliminate possible problems based on your understanding of the behavior of your car. When we, as a country, make environmental policies that will impact neighboring states, countries, and the rest of our planet over many years, our citizens need to be able to recognize that seemingly simple actions like chopping down trees or drilling for oil can affect our air

quality and food supplies. Programming provides children with some hands-on experience dealing with complex systems that they create themselves. When their programs do not behave as expected, children have to learn to isolate the problems and solve them. They learn to narrow the scope of a problem and that a single malfunctioning program component can cause other program components to malfunction.

In addition to the critical thinking skills students develop through programming, an understanding of computer programming may prove to be a valuable job skill for many students. Few of today's students will be able to avoid working with computers in some capacity. Some research estimates that up to 30% of our computer-using workforce will be required to do some programming activities as part of their job (Scaffidi, Shaw et al. 2005). Further, many students who choose not to pursue computer-related careers may find themselves working with computer scientists, programmers, and engineers in some capacity. Particularly for those students who will eventually work with computer professionals, a basic understanding of computer programming will be helpful in preparing students to communicate and work productively with computer professionals.

1.3 Attracting Women to CS

To get a larger, broader group of people to enter the field of computer science, we need to get a larger, broader group of people to take the first steps towards computer science careers. One of the main entry points for computer-related careers is learning to program. The ACM K-12 task force describes the relationship between computer science and programming in the following way:

> *While programming is a central activity to computer science, it is only a tool that provides a window into a much richer academic and professional field. That is, programming is to the study of computer science as literacy is to the study of literature (Tucker, Deek et al. 2002).*

Learning to program provides students with the basic skills necessary to pursue computer science. If we can increase the number of female students who learn to program and who enjoy programming, we will likely help increase the numbers and diversify the community of people capable of succeeding in computer-related jobs.

However, programming courses at both the high school and college level have traditionally failed to attract a significant number of female students (AAUW 1998). Researchers have suggested a variety of factors that contribute to girls' low enrollments in computer science including disinterest in computers, concerns about the computing culture, lack of encouragement from peers, parents, and educators, and relatively fewer opportunities to interact with computers (Furger 1998; AAUW 2000). It is likely that many of these factors play some part in girls' decisions not to pursue computer science. While it would be difficult to broadly address the cultural factors that influence girls' decisions not to pursue computer science, we can make the process of learning to program more motivating for girls.

One factor that may contribute to students' loss of interest in computer science is that students often find their first experience with computer science uninspiring. Typical assignments in beginning computer science courses like "sort a list of numbers" or "generate the sum of the first 1700 integers" fail to engage many students.

To make the process of learning to program more relevant for students, it is important to introduce programming as a *means to a motivating end*. What constitutes an "interesting end" for a female student may vary considerably with age. To have the greatest potential impact, I chose to focus on designing a programming system for middle school girls. Studies have shown that middle school is a critical age for girls; many girls decide whether or not to seriously pursue the study of math and science during middle school (AAUW 1996). By late high school many girls have already opted out of the math and science classes that would enable them to pursue a mathematical or scientific major in college (AAUW 1998). If girls have a positive experience with computer programming during middle school, they may be more likely to consider enrolling in a high school or college programming class.

1.4 Middle School Girls

To successfully design a programming system for middle school girls, it is important to understand this age group. In this section, I will attempt to provide some insight into what a middle school girl is *like.* Most of the information reported here is based on research that included many girls, so any single girl may not exhibit all or perhaps even any of the characteristics of the broader group. Most girls will have at least some of the characteristics of the broader group.

1.4.1 Gender Expectations

Most girls hit puberty at twelve; middle school children are typically between twelve and fourteen (Collins and Kuczaj 1991). In U.S. culture, as children start to exhibit the physical changes of puberty, many people begin to expect both more adult behavior and more gender-appropriate behavior from them (Collins and Kuczaj 1991). Because there is still some disagreement on how girls and women should behave, girls at this age often receive very confusing messages from society (AAUW 1996). In their report Girls in the Middle, the American Association of University Women described the expectations for adolescent girls this way:

> *Adolescent girls are to be sexy and flirtatious but at the same time remain "good girls." They are to fend off aggressive male attention while simultaneously meeting teachers' expectations of non-aggressive behavior. Females are to put domestic life first at the same time that they prepare for financial independence (AAUW 1996).*

As adolescents start working towards determining how to balance the sometimes conflicting expectations for young men and women, they tend to become much less accepting of gender transgressions (behavior associated with the opposite gender) in their peers than they were just a couple of years before (Collins and Kuczaj 1991). One study revealed that adolescents are hesitant to partake in activities that they perceive as belonging to the opposite gender because they believe that it may cause or at least be indicative of a problem with sexual identity (Collins and Kuczaj 1991).

1.4.2 Academics

During the middle school years, many girls experience a drop in self-esteem and confidence in academics, particularly in math and science (AAUW 1996). In the third grade, approximately the same number of boys and girls believe that they are good at math (64% of girls and 66% of boys) (Dossey, Mullis et al. 1988). By seventh grade, 57% of girls and 64% of boys believe they are good at math (Dossey, Mullis et al. 1988). That number of girls who believe they are good at math drops to 48% by the end of high school (Dossey, Mullis et al. 1988). This confidence drop typically *precedes* a drop in academic performance (Fennema and Sherman 1977). Girls' experiences with science are similar. From elementary school through high school, girls are less likely to do science based activities than boys (AAUW 1996). While girls express interest in science in the elementary years, they have increasingly negative views of science, science classes, and science-based careers as they progress through middle and high school (AAUW 1992). While the exact reasons for the self-esteem drop and later performance drop in math and decreasing interest in science are unclear, there are a couple of known factors that may contribute. Girls tend to attribute their own struggles and failures to a lack of ability, which may cause them put less effort into their schoolwork (AAUW 1992). Boys, on the other hand, tend to attribute failures to insufficient hard work or, bad luck (AAUW 1992). Additionally, excelling in school may lead to social isolation for girls, as their female peers begin to lose academic confidence and downplay the importance of academics (AAUW 1996).

1.4.3 Identity Formation

For girls and for boys, the process of forming an identity is a fundamental activity of the middle school period (Stone and Church 1984). Psychologists define identity as an "internal, self-constructed dynamic organization of drives, abilities, beliefs and individual history" (Marcia 1980). In other words, identity is a person's own ideas about what he or she stands for, what he or she is good at, what he or she enjoys doing, and how his or her past helped to shape who he or she is today. Identity is dynamic, so it can change to adapt to new life experiences, new friends, and new found abilities or failures (Stone and Church 1984).

Psychologists believe that identity formation is a process that must include two elements: a crisis and a commitment (Stone and Church 1984). During the crisis, children become aware that they are expected to fill or begin to anticipate adult roles and start to feel pressure to integrate information and emotions about themselves and their experiences (Marcia 1980). Children who do not experience the crisis often accept the roles that parents, teachers, peers, or other significant figures in their lives want for them (Marcia 1980).The crisis is resolved when a child commits to a role or set of roles that he or she is currently filling or working towards filling as an adult and a set of beliefs about themselves and the world they inhabit (Marcia 1980).

To resolve their identity crises, middle school aged children typically experiment with a wide variety of roles in their interactions with peers, teachers, parents, etc. (Frankel 2002). Before settling on a role or set of roles, middle school aged children may change roles and personae rapidly (Frankel 2002) and may take on different roles when interacting with different people (AAUW 1996). A girl of this age may try to appear daring and independent with her friends, obedient and responsible with her teachers, and childish and playful with her parents as she tries to sort out what she stands for, what her strengths are, and what she enjoys.

1.4.4 Middle School Girls and Computers

Girls are now using computers in large numbers. As of 2000, 9 out of 10 children in the US have access to a computer in school or at home (Newburger 2000). According to data from the 2000 census, over 66% of households with children between 6 and 17 have at least one computer (Newburger 2000). Among teens 13 to 17 years of age, 73% of girls and 70% of boys use the Internet, although their usage patterns tend to differ (GA 2003). Girls tend to do more communication-based activities: 68% of girls report using email "very often" or "pretty often", as compared to 50% of boys (Roper 1999); 56% of girls report using instant messaging "very often" or "pretty often", as compared to 48% of girls. Boys are more likely to report that they play games (50% of boys vs 43% of girls) or gather information about sports (40% of boys vs 15% of girls) "very often" or "pretty often" (Roper 1999). Still, girls are unlikely to move from using computer software to signing up for a computer programming class. Potential reasons for this include the

perception of computer science as a male domain and the fact that traditional presentations of computer science are uninteresting to many girls.

When girls are asked to draw someone who is good with computers (a computer whiz) the computer whiz is male in 71% of responses, female in 18% of responses, and indeterminate in the remaining 11% (Castell and Bryson 1998). Many girls view the heavily technical culture, with its emphasis on clock speeds, megabytes and other performance metrics as a fairly uninteresting domain and one that is most appropriate for boys (AAUW 2000). Because adolescents are less tolerant of cross-gender activities than kids of most other ages, the perception of computer science as a boys' activity can itself be an inhibitor, even for girls who are interested (Collins and Kuczaj 1991).

1.5 Storytelling as a Motivating End

I chose to base my programming system around the activity of creating animated 3D movies. Storytelling is a good context for middle school girls to learn about computer programming:

1. Given a little bit of time, most girls can come up with a story they would like to tell. Storytelling is, at its core, a form of communication which is an important activity to most middle school girls.
2. Stories are naturally sequential, allowing users to begin by creating sequences of instructions and gradually progress to more advanced programming concepts as they gain experience and confidence.
3. Stories are a form of self-expression and provide girls an opportunity to experiment with different roles, a central activity during adolescence.
4. Non-programming friends can readily understand and appreciate an animated story, which provides an opportunity for girls to get positive feedback from their friends.

1.6 Leveraging Alice 2

While learning to program would be valuable for many students, it is also difficult and frustrating for many students. To avoid some of the common problems associated with learning to program, I leveraged an existing system for novice programmers: Alice (2003).

Alice is a programming environment for novice programmers that allows users to create interactive 3D virtual worlds, including movies and games. Alice was designed to make the process of learning to program easier and less frustrating for beginning programmers by addressing two of the common difficulties beginning programmers encounter: syntax errors and invisible state. In Alice, users construct programs by dragging and dropping code elements, which removes the possibility for making syntax errors. Programs in Alice are animations which enables users to see their mistakes. Alice allows students to gain experience with the programming concepts and constructs typically taught in a first computer science course. These include looping, conditional statements, methods, parameters, variables, arrays, and recursion.

The mechanical supports Alice provides can help broaden the pool of CS majors. NSF-sponsored studies have shown that Alice increases the academic success and retention of at-risk college students (freshmen intending to major in CS who enter college with no programming experience and/or who are not prepared to enroll in Calculus as freshmen). At-risk students who enrolled directly into a Java-based CS 1 class earned an average grade of C and only 47% of them continued on to the second course. After a short Alice course, at-risk students performed as well as well-prepared students in the Java-based CS1 course: they earned an average grade of B and 88% of them continued on to the second course (Moskal, Lurie et al. 2004). The Moskal, Lurie, et al. study did not control for the amount of time students' spend in class. Follow-up studies that control for contact-time are currently underway. Alice is currently being used in CS1 courses at more than 100 colleges and universities.

The Alice system removes syntax-based frustration and makes data state visible. While decreasing the frustration associated with learning to program may help us retain those students who are already interested in computer science, it is not sufficient to attract new students into computer science. No matter how easy something is people still need a reason to want to do it. To provide a motivation for middle school girls to learn programming, I created a modified version of Alice that better supports girls in creating animated stories.

1.7 Research and Contributions

My thesis work is composed of three components: 1) formative user testing to determine how to support girls in creating animated stories 2) modification of the Alice programming environment to support storytelling (I will refer to the modified version of Alice as Storytelling Alice) and 3) an evaluation of Storytelling Alice on girls' success and interest in learning to program.

1.7.1 Formative Testing

More than 200 middle school girls (most were Girl Scouts between 11 and 15) participated in the formative evaluation that informed the design of Storytelling Alice. Over 18 months, I created and tested 15 different versions of Storytelling Alice. In early user tests, I asked girls to work in pairs to create an animated story using a version of Alice. Pair-based user testing is a common technique for gathering usability and requirements information (Nielson 1993). However, in the case of a creative task like storytelling, pair-based testing was ineffective because pairs often had difficulties negotiating the storyline and frequently stopped talking with each other. As an alternative method for capturing girls' visions for their stories, I developed a three-step storyboarding process in which girls write: 1) the "back of the dvd box" description of their story, 2) break the story into scenes and describe the setting, action, and purpose for each scene, and 3) create a storyboard of 6-9 frame drawings with accompanying textual descriptions for each scene.

In trying to improve Storytelling Alice, I was guided by girls' storyboards of what they wanted to build, observed problems and questions that came up during user testing, and recorded logs of the actions that girls took in interacting with Storytelling Alice.

1.7.2 Development of Storytelling Alice

Based on user testing, I made three major changes to the Alice system:

1) I added a set of high-level animations and support for creating multiple scenes.

The animations in Alice 2.0 allow users to perform simple graphical transformations like moving and rotating a character or one of its parts. Using simple transformations, it is often tedious and frustrating to create the kinds of animations that girls needed for their stories such as walking or having two characters hug each other. By analyzing the storyboards girls created for their movies, I identified a high-level set of animations that enables girls to make more rapid progress on their stories without removing the motivation to learn a variety of programming constructs.

Figure 1.1: A scene created in Storytelling Alice.

Most of stories that girls imagined creating take place in several different scenes. While it is possible to create the appearance of multiple scenes in Alice 2.0, it is an involved

process that is unsuitable for novice Alice users. To enable girls to create multi-scene stories, I added scene support to Storytelling Alice. In addition to allowing girls to more easily create the stories they envision, the need for multiple scenes provides a nice opportunity for introducing the concept of subroutines.

Table 1.1: A comparison of the animations in Storytelling Alice and Generic Alice.

Storytelling Alice	**Generic Alice**
Say, think	Move
Play sound	Turn
Walk to, walk offscreen, walk	Roll
Move	Resize
Sit on, lie on	Play sound
Kneel	Move to
Fall down	Move toward
Stand up	Move away from
Straighten	Orient to
Look at, Look	Point at
Turn to face, turn away from	Set point of view to
Turn	Set pose
Touch, Keep Touching	Move at speed, turn at speed, roll at speed

2) I created a library of 3D characters and scenery that helps to spark story ideas.

One of the determining factors in girls' motivation to learn to program in Storytelling Alice is whether or not they can find a story that they want to tell. I found that the gallery of 3D characters and scenery can be a source of inspiration for girls' stories. In particular, highly caricatured characters with clear roles and giving characters animations that require some explanation within the story (e.g. what made a robot character go crazy) can help spark ideas.

3) I created a story-based tutorial that introduces users to the mechanics of creating programs in Alice in the context of stories similar to the ones girls envision creating. I found that placing the tutorial within a story-context was necessary to engage girls. However, story-based examples tend to be more complex and contain greater potential for user error than traditional tutorial examples which are often chosen to illustrate a concept as simply as possible. To moderate the additional complexity of story-based tutorials, I created a new interaction technique called "Stencils." The Stencils technique overlays a transparent blue screen that catches mouse and keyboard events over the active Alice interface. User instructions and explanatory information are presented using sticky-style notes drawn on the blue screen. Holes in the Stencils allow users to interact only with the interface components necessary for the current step. Stencils is an adaptation of a related (unpublished) overlay-with-holes technique that I co-developed with Cliff Forlines while interning for Alan Kay's group at Walt Disney Imagineering.

The Stencils technique prevents most user errors and enables the presentation of more complex tutorials. This technique allowed me to create substantially richer, detailed tutorials, which helped underscore that the system can be used for storytelling.

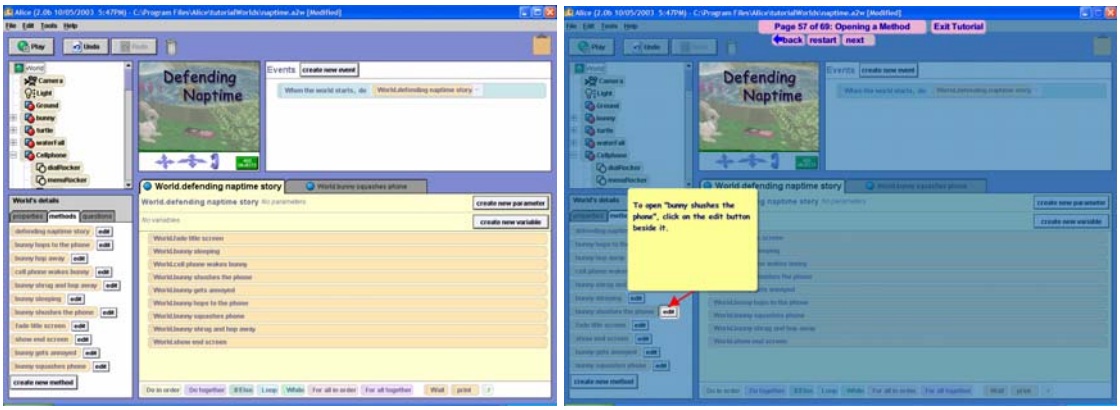


Figure 1.2: Views of the Alice interface without and with a Stencils-based tutorial.

1.7.3 Development of Storytelling Alice

To evaluate how successful Storytelling Alice is at motivating girls to learn programming, I conducted a between-subjects study comparing girls' motivation and

learning when they were introduced to programming using Storytelling Alice and a version of Alice without storytelling support (Generic Alice). Study participants recruited from the Girl Scouts were randomly assigned to use either Storytelling Alice or Generic Alice. I used a programming quiz given after the study participants had finished working with their assigned version of Alice to measure their mastery of programming concepts. To measure motivation, I collected several kinds of data including participants programs and actions within Alice (what girls built), their opinions of Alice attitude survey responses (what girls said), and behavioral measures that indicated engagement such as the percentage of girls in either condition who snuck extra time to work on their programs (what girls did).

Study results suggest storytelling is a highly promising approach for motivating more girls to learn computer programming. Girls who used Storytelling Alice and Generic Alice performed statistically similarly on the programming quiz. Girls who used Storytelling Alice show significantly more evidence of motivation than girls who used Generic Alice.

What girls built:

Participants who used Storytelling Alice spent 42% more time on the programming aspects of Alice than participants who used Generic Alice, suggesting that storytelling helps to make the activity of programming more appealing ($p < .001$). Given the short period of time users spent working with Alice, users of Storytelling Alice did not learn more than users of Generic Alice.

What girls said:

Although girls in the Storytelling Alice and Generic Alice conditions found their assigned version of Alice similarly easy to use and entertaining, girls who used Storytelling Alice expressed stronger interest in future use of Alice, either as part of a class or on their own ($p = .05$).

What girls did:

Further, users of Storytelling Alice were nearly three times more likely to sneak extra time to continue working on their programs after "time was up;" 16% of Generic Alice users and 51% of Storytelling Alice users snuck extra time ($p < 0.001$).

Participants who used Storytelling Alice wrote stories about a wide variety of topics including whether or not you should abandon your friends if you are given a chance to hang out with the popular crowd, how to deal with a cheating boyfriend, the difficulties of moving to a new town, how to find a kidnapped dog, and a father with no sense of direction. Approximately half of the stories the girls created addressed deep issues that middle school girls face. It seems clear this approach can provide a vehicle for girls to think about issues they are facing.

Storytelling can provide a gentle, motivating introduction to programming concepts. Girls often begin by creating sequences of instructions and, as they gain confidence, create new scenes and new actions for their characters, tasks which often require more complex programming constructs. Girls' storyboards commonly included motivation to use methods, parameters, loops, and parallel execution.

1.7.4 Contributions

This thesis makes several contributions:

1. **Results of the study comparing girls' learning and motivation using Storytelling Alice and Generic Alice demonstrate that storytelling is a promising approach for motivating more girls to learn computer programming.** As we continue to search for ways to attract a larger and more diverse group of students to study computer science, storytelling is an activity we should consider. Informal testing with other demographic groups suggests that storytelling has a broad appeal and is approachable for beginning programmers.
2. **Through formative user testing, I found that the commonly used pair-based usability testing approach is poorly suited for creative tasks such as storytelling.** In user testing creative software in which users set their own goals, it

is important to find ways to capture users' plans and intentions before they begin to change them based on their interaction with the software. For the activity of storytelling, a 3-step, gradual refinement, storyboarding process can help to capture users' goals.

3. **Through analysis of girls' storyboards and user testing, I developed a set of high-level animations that enable girls to create the kinds of animated stories they envision.** The set of high-level animations for humans can be used as a starting point for other systems that are designed to allow non-expert users to animate human beings.
4. **I developed the Stencils interaction technique to guide users through tutorials and prevent most user-errors, enabling the presentation of more complex, story based examples in the Storytelling Alice tutorial.** The Stencils technique is a general interaction technique that can be used to present tutorial and help information in user interfaces. It is particularly well suited for interfaces which make heavy use of point-and-click or click-and-drag interactions.

Chapter 2 Introduction to Alice 2

I chose to base my programming system for middle school girls on the Alice system, a programming environment in which users construct programs via drag and drop to control the behaviors of objects in a 3D virtual world. Alice 2.0, the most recent version of the system, was designed for college-level introductory computer science students without prior programming experience. In this chapter, I will describe Alice 2.0, which served as my starting point in creating a version of Alice for middle school girls.

2.1 Why Alice?

Alice provides support for two problems that beginning programmers often struggle with: 1) typing syntactically correct programs and 2) finding and fixing logic bugs (Cooper, Dann et al. 2003). Users construct Alice programs by dragging and dropping program tiles and selecting parameters from a list of valid choices, a method or program construction that prevents syntax errors. Alice allows users to master programming logic and control structures independently of learning to type syntactically correct program statements. While other systems have prevented users from making syntax errors (Kay; Teitelbaum and Reps 1981; Miller, Pane et al. 1994; Smith, Cypher et al. 1994; Begel 1996; Kahn 1996; Repenning and Ambach 1996; Goldman 2003; Maloney, Burd et al. 2005), most limit users to a small subset of the control structures typically found in general-purpose programming languages. Alice allows users to gain experience with all

of the standard programming constructs taught in introductory programming classes including loops, if-then-else statements, recursion, variables, methods, functions, and parameters.

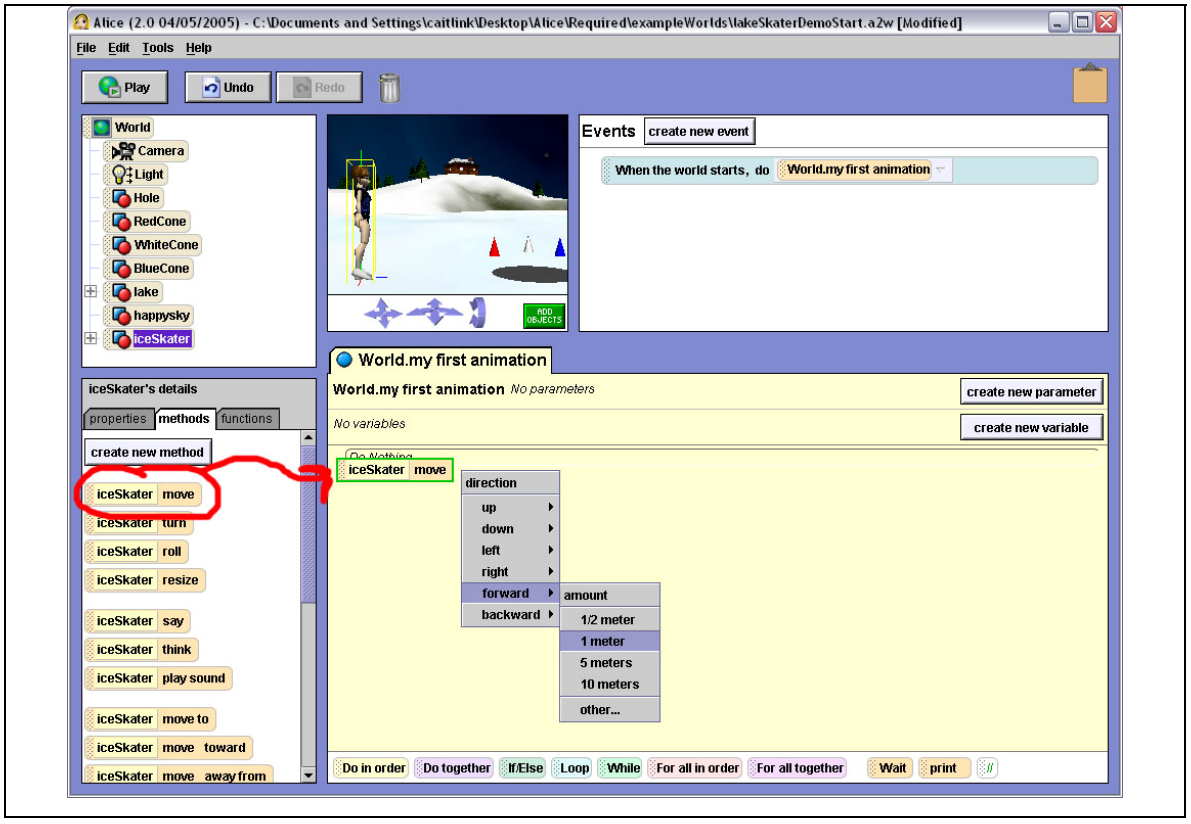


Figure 2.1:To call the IceSkater's move method, the user drags the tile "IceSkater move" into the method editor, drops it, and selects parameters from the pop-up menus.

In Alice, running programs are animated, so students are able to see where they have made mistakes in their programs. Consider, for example a program in which a dragon is supposed to leave a castle (by crossing over a moat) and then fly to a knight standing to the left of the castle. Students' ability to watch the dragon's motions can ease the debugging process. Seeing, for example, that the dragon turned the wrong way after crossing the moat helps students understand why the dragon is not finishing in the correct location. In many introductions to programming, students detect errors through incorrect printed program output. This is similar to being able to see only the dragon's final position in the previous example, but not what motions he made to arrive in his location. Several other novice programming environments animate programs to provide immediate

feedback to users (Kay; Papert 1980; Pattis 1981; Kahn 1996; Pane 2002; Maloney, Burd et al. 2005).

The activity of storytelling is a good match for Alice; Alice is designed to allow users to create interactive 3D virtual worlds. Animated stories are essentially non-interactive virtual worlds, a natural subset of Alice's design space. In Alice, students create programs that control the behaviors of graphical objects in a 3D virtual world, often short animations or simple games. In Alice 2.0, users can combine simple animations (e.g. move and turn) to create more complex behaviors like dances and gestures for the characters and objects in their Alice 2.0 worlds. It is possible, although frequently difficult, to create stories in Alice 2.0.

An early study indicates that using Alice 2.0 helps at-risk CS majors to succeed in introductory programming and increases the chances that they will continue to pursue a Computer Science degree (Moskal, Lurie et al. 2004). At-risk CS majors are incoming CS majors who lack programming experience and/or who do not have sufficient grounding in mathematics to enroll in Calculus. Typically, at-risk students, who are disproportionately female and minority students, perform poorly in their first Computer Science course (CS1) and less than half enroll in the second Computer Science course (Cooper, Dann et al 2003). Cooper, Dann, et al's study demonstrated that students who take an Alice programming class either prior to or concurrent with a Java-based CS1 course perform a letter grade better than students not exposed to Alice and 88% of the students exposed to Alice (vs. 47% of students not exposed to Alice) continue to CS2 (Cooper, Dann, et al 2003). A larger scale study is in progress to validate Cooper, Dann et al's results with a larger group of students at a variety of institutions. However, Alice 2.0's early successes at the college level, particularly among female and minority students, made Alice 2.0 a strong starting point for my work.

Alice is also beginning to have a real-world impact. Prentice-Hall published a textbook for college-level introductory programming classes based on Alice in August of 2005: Learning to Program with Alice by Wanda Dann, Stephen Cooper, and Randy Pausch. As

of spring 2006, Alice 2.0 is in being used in classrooms at more than 100 universities and 100 high schools across the United States. Prentice-Hall believes that number will continue to grow rapidly in coming years.

2.2 Creating Alice Worlds

In this section, I will describe how users build worlds in Alice. In later chapters, I will describe the modifications I have made to Alice 2.0 to support storytelling. In both versions of Alice, there are two basic steps in creating a program: 1) selecting and laying out 3D objects within the virtual world and 2) adding animations to control the behavior of those 3D objects. Figure 2.2 shows a screenshot of the Alice interface.

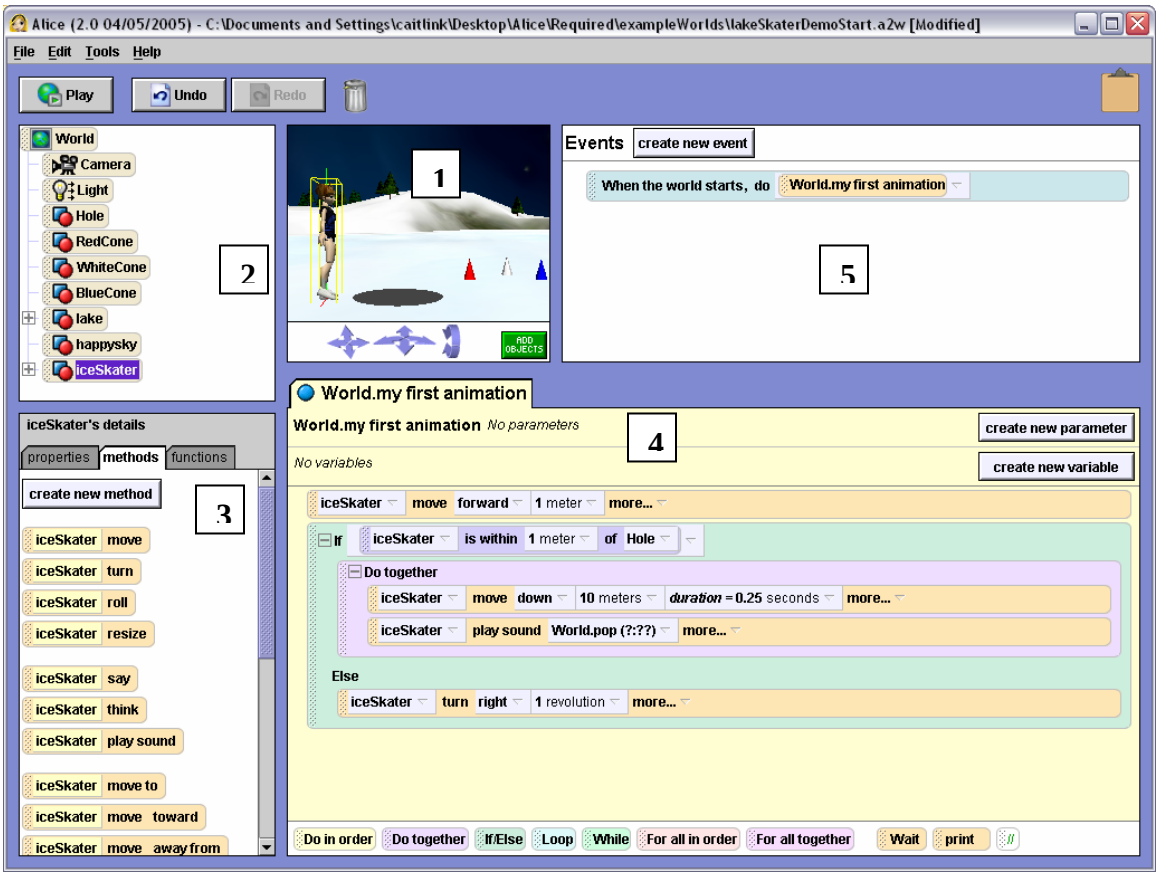


Figure 2.2: A screenshot of the Alice interface. 1) The world window provides a view of the virtual world that a students' program will control. 2) The object tree contains a list of the 3D objects in the virtual world. 3) The details area shows the properties, methods, and functions for the object selected in the object tree. 4) The methods editor shows the code that defines a method a student is working on. 5) The events area allows students to call methods based on events in the world, such as mouse clicks or changes in the value of a variable.

2.2.1 Adding and Positioning 3D Objects

Alice 2.0 includes a gallery of more than 350 3D objects that students can use in their programs organized into groups including animals, people, amusement park models, buildings, 3D controls such as button and switches, musical instruments, space models, and many more. A larger gallery with more than 400 additional models is accessible through the web. While some of these objects were specifically created to support the Alice textbook or in response to user testing, most of them were created by undergraduates as part of course projects. Consequently, models in the gallery vary widely in both quality and utility.

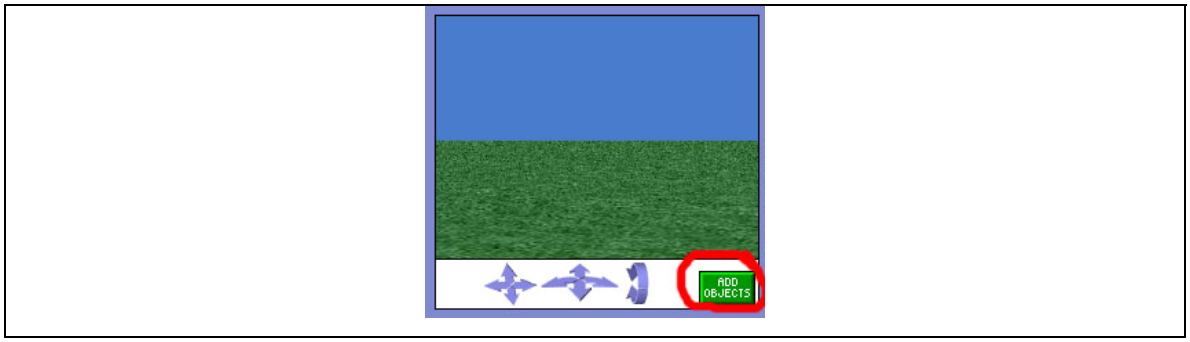


Figure 2.3: Users press the "Add Objects" button to access the Alice gallery.

To access the Alice gallery, students press the “Add Objects” button underneath the world window (see Figure 2.3). This causes the world view to expand and the gallery is displayed underneath the world view. Groups of objects are stored in folders. By clicking on a folder, students can see the objects inside that folder. Figure 2.4 shows the objects in the medieval folder. Students can add an object to their virtual world by dragging the object into the 3D scene and/or by clicking on the picture of the object and pressing the “Add instance to world” button on the dialog box that appears.

In Alice, when users drag an object into the world, a bounding box appears in the world so users can position the new object. When they release the mouse button, the object is added at the final position of the bounding box.

Users can move any object in their virtual world by clicking on the object and dragging it along the ground plane. A series of buttons to the right of the world view in both versions

of Alice allows users to change the dragging behavior of the mouse so that the mouse moves objects up and down, rotates objects, resizes objects, or makes copies of objects. To move the position of the camera in their virtual world, users can click and drag on the arrows beneath the world view. When users are ready to begin animating their virtual worlds, they can press the "Done" button to the right of the world view.

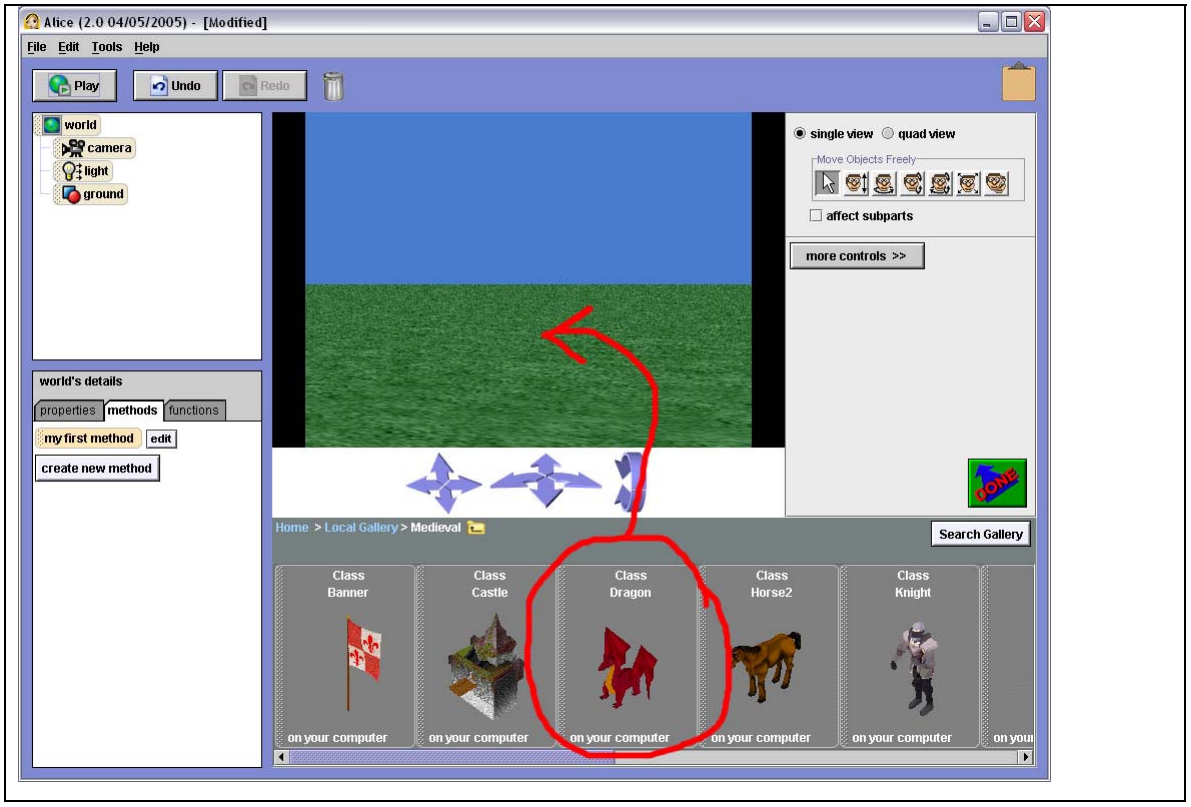


Figure 2.4: A view of the objects in the Medieval folder. Users can add 3D objects like the Dragon by dragging them into the 3D scene.

2.2.2 Creating Sequential Programs

By default, Alice worlds contain a pre-created method called "my first method" that is called when the world starts (e.g. when the user presses the play button). This allows Alice users to simply add method calls to "my first method" and press the play button to run their program. Typically, beginning Alice users assemble their first programs within "my first method."

Users begin to create programs in Alice by dragging command tiles into the method editor for "my first method." To see what methods a particular object can do, a user can

select that object in one of two ways: 1) by clicking on the object's tile in the object tree or 2) by clicking on the object itself in the 3D world window. Alice will display the methods that the selected object can perform in the details area. To call a method on an object (as part of a program), the user can drag the method tile into the method editor, drop it and select the parameter values from a pop-up menu (see Figure 2.5).

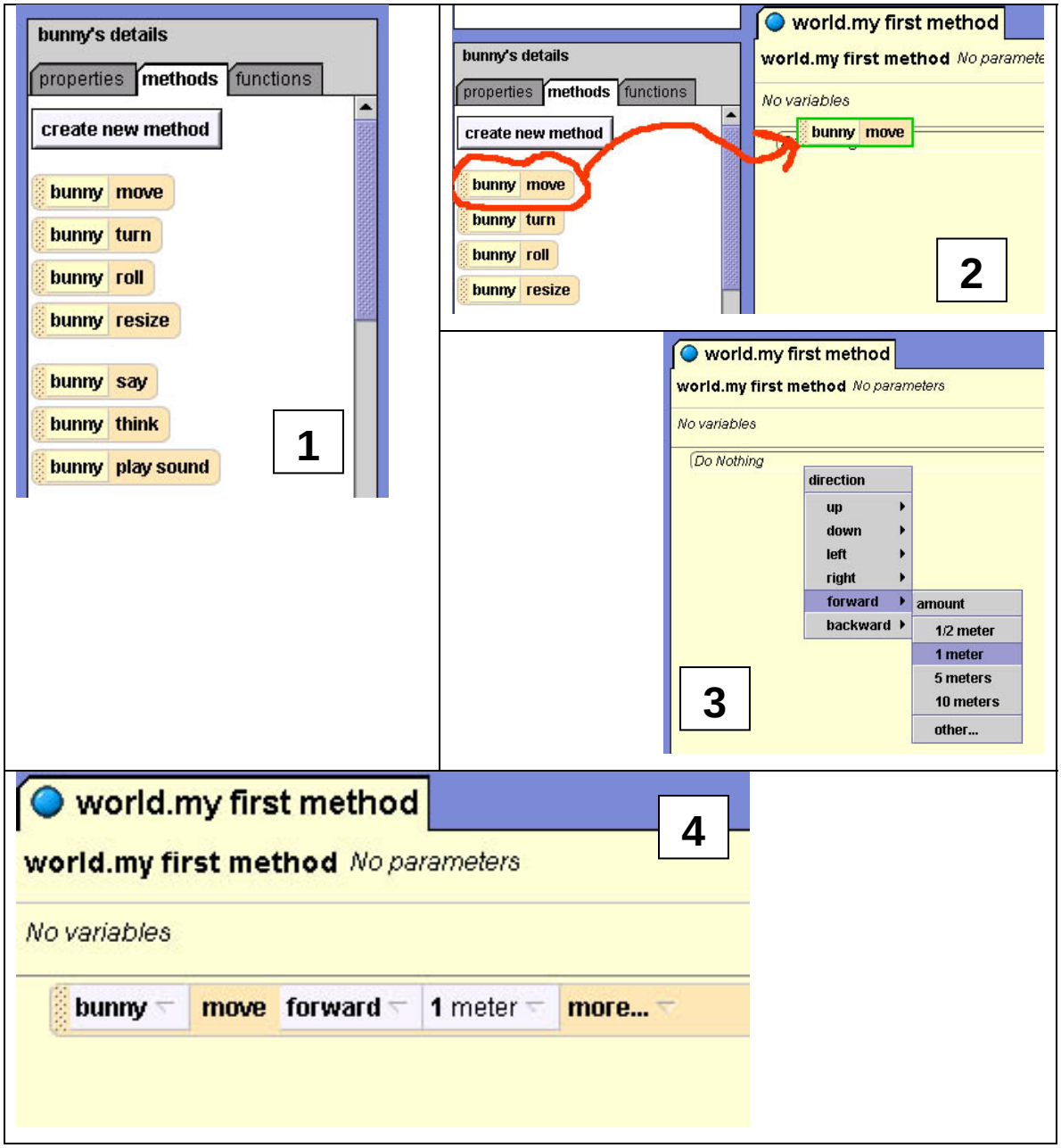


Figure 2.5: To animate the bunny in Alice 2, the user 1) selects a method from the list of methods the bunny can perform, 2) drags the method into the editor for "my first method" and drops it, 3) and selects parameters from the pop-up menus. 4 shows a completed method call that tells the bunny object to move forward by 1 meter.

Method editors can contain as many method calls (or lines of code) as the user desires. Scroll bars appear when the method length exceeds one screen; expert users have written 3000 line programs using Alice. As in typical programming languages, method calls are executed in order. Users can rearrange the lines of code by dragging and dropping them.

In Alice 2.0, all 3D objects perform methods drawn from the thesis work of Matthew Conway on an earlier version of Alice (Alice 98). In his thesis, Conway studied how to make 3D graphics transformations such as translate, rotate, and scale understandable to and usable by undergraduates in non-engineering majors. By observing users completing a paper-based tutorial for Alice 98, Conway determined how users expected simple 3D animations to behave and modified the naming and behavior of the animations in Alice to match users' expectations.

In addition to performing animations on whole 3D objects (e.g. moving a cow through space), users can also animate an object's subparts (e.g. the cow's legs).

In Alice 2.0, objects can perform the following methods:

object.move(direction, amount)
The move animation slides an object some number of meters in a direction (forward, backward, left, right, up, down).

object.turn(direction, amount)
The turn animation rotates an object by some number of revolutions in a direction (left, right, forward, backward). Turning an object left or right rotates the object around its up-down axis. Turning an object forward or backward rotates the object around its left-right axis.

object.roll(direction, amount)
The roll animation rotates an object by some number of revolutions in a direction (left, right). Rolling an object left or right rotates the object around its forward-backward axis.

object.resize(amount)

The resize animation resizes an object by a multiplicative factor. Resizing an object by 2 makes the object twice as large in every dimension.

object.say(message)

The say animation displays a cartoon speech bubble containing the message over the object. Although this is included in Alice 2.0, it was one of the first additions in support of storytelling that I made to the Alice system.

object.think(message)

The think animation displays a cartoon thought bubble containing the message over the object. Like say, think was one of my early modifications to Alice in support of storytelling.

object.playsound(sound)

The playsound animation plays a sound in wav or mp3 format.

object.move to(target)

The move to animation moves one object to another object's position. In Alice, an object's position is defined to be the position of the object's origin, which is specified by the artist when the 3D object is created. The most common origin locations are the center of the object and the center of the object's bottom face. The move to animation moves an object so that its origin is in the same location in the Alice world as the target object's origin. From a user's perspective, the move to animation may cause objects to appear to be on top of one another or cause the moving object to end in a position either above or below the Alice ground plane.

object.move toward(amount, target)

The move toward animation moves the object toward the target by some number of meters along the line connecting the object's origin with the target's origin. The line

between the object's origin and target's origin may not be parallel to the ground plane. This can result in an object moving either into the ground or off of the ground.

object.move away from(amount, target)

The move away from animation moves the object away from the target by some number of meters along the line connecting the object's origin with the target's origin.

object.orient to(target)

The orient to animation rotates the object so that its axes are parallel to the target's axes.

object.point at (target)

The point at animation rotates the object so that its forward axis is pointing toward the target's origin. This can result in an object appearing to lean forward or backward.

object.turn to face(target)

The turn to face animation rotates the object around its up-down axis so that its forward axis is coplanar with a vector beginning at the target's origin and pointing up.

object.set point of view to(target)

The set point of view animation moves and rotates the object so that it has the same position and orientation in the Alice world as the target.

object.set pose(pose)

In Alice, users can create poses for objects by moving their subparts. For example, a person might have arms and legs as subparts. A checkbox in the scene layout controls allows users to use mouse dragging to manipulate objects' subparts. Once a user has created a pose he or she wants to keep, he or she can press the "capture pose" button on the properties panel in the details area. The pose stores the positions and orientations of all of the objects subparts. Set animates an object and all of its subparts from their current positions and orientations to the positions and orientations stored in the pose.

object.stand up()

The stand up animation rotates an object so that its up-down axis is aligned with the Alice world's up-down axis.

object.move at speed(direction, speed)

The move at speed animation moves an object at a given speed in a direction (forward, backward, up, down, left, or right).

object.turn at speed(direction, speed)

The turn at speed animation rotates an object at a given speed in a direction (forward, backward, left or right). The rotation directions are the same as for the turn animation.

object.roll at speed(direction, speed)

The roll at speed animation rotates an object at a given speed in a direction (left or right). The rotation directions are the same as for the roll animation.

object.constrain to point at (target)

The constrain to point at animation rotates the object to point at the target (like the point at animation) during every frame of the animation. This animation can be used to have a character's head track a moving object.

object.constrain to face(target)

The constrain to face animation rotates the object so that it is facing the target (like the turn to face animation) during every frame of the animation. This animation can be used to have a character constantly face a moving object.

By default, all Alice 2.0 animations animate over 1 second using slow-in and slow-out. Further, when the animation depends on a coordinate system, Alice 2.0 uses the object's coordinate system by default. For example, if a user tells a person to move forward 1 meter, the person will move 1 meter in the person's forward direction rather than the world's or the camera's forward directions.

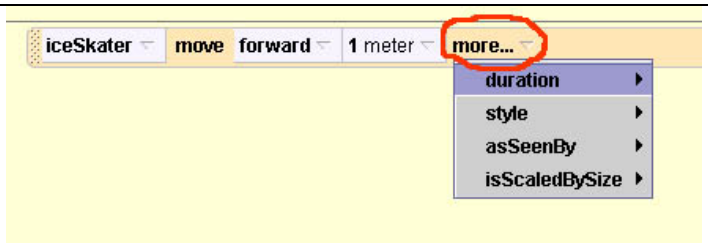


Figure 2.6: Users can specify values for optional parameters for a method call using the "more…" pop-up menu.

In designing a system for novice programmers, it is crucial not to overwhelm beginning users with details. Consequently, most methods in Alice only require users to specify one or two parameters. For example, when adding a move animation, a user must specify which direction the object should move and how many meters. To provide users with the option of greater control over their animations, Alice presents infrequently needed parameters as optional parameters. Optional parameters are assigned a default value when the animation is created. If users want to change the values for any optional parameters, they can click on the "more…" at the end of the method call (see Figure 2.6).

Users can also create new methods for specific objects in their worlds. For example, a user might want to teach a ballerina to pirouette or a kangaroo to hop. Most students intuitively understand that a ballerina should be able to do different things than a kangaroo. Working with object-methods in Alice gives students a concrete space to begin to think about the concept of objects in programming. Alice is an object-based (not object-oriented) system. When a user creates a pirouette animation for a ballerina, it applies only to the selected ballerina, not all ballerinas. When students transition into a language like Java or C++, they have to tackle the concepts of classes, instances, and inheritance.

2.2.3 Using Programming Constructs

Alice provides several programming constructs. Tiles representing programming control structures such as if-statements and loops are displayed along the bottom of the method editor. To add a control structure, users can drag the tile for that control structure into the method editor, drop it and select any necessary options. Then, the user drags any statements that they want to be affected by that control structure into it. Each of the

control structures available in Alice is discussed briefly below. Both Alice 2.0 and Storytelling Alice provide the same control structures.

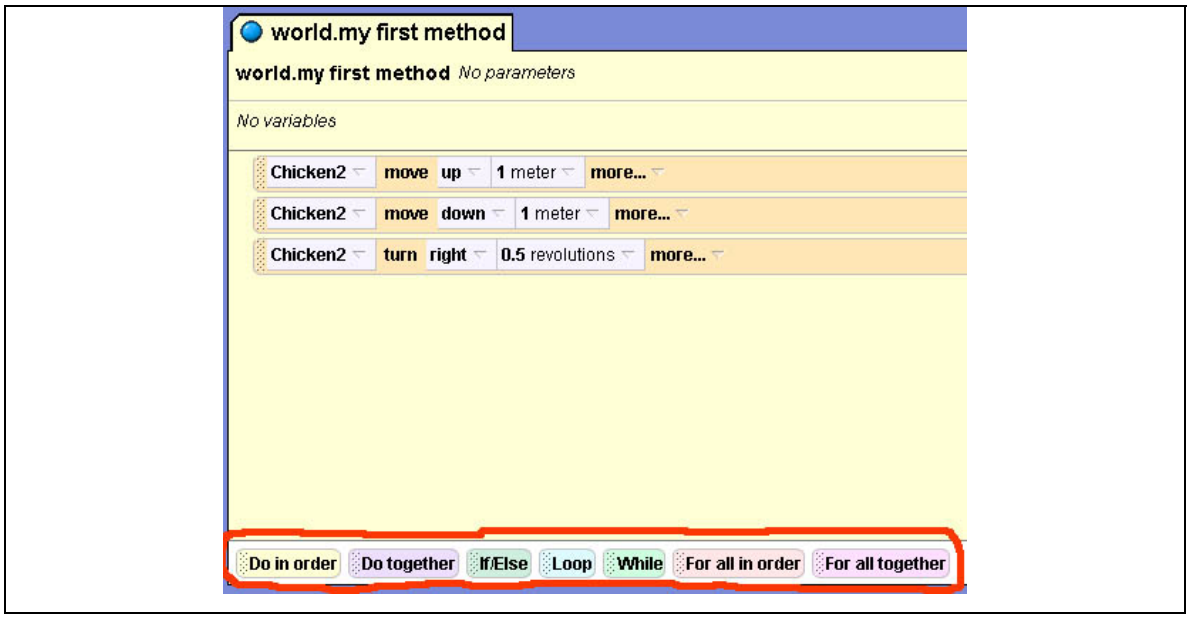


Figure 2.7: Users can add control structures to their programs by dragging and dropping the control structure tiles from the bottom of the method editor.

Loops

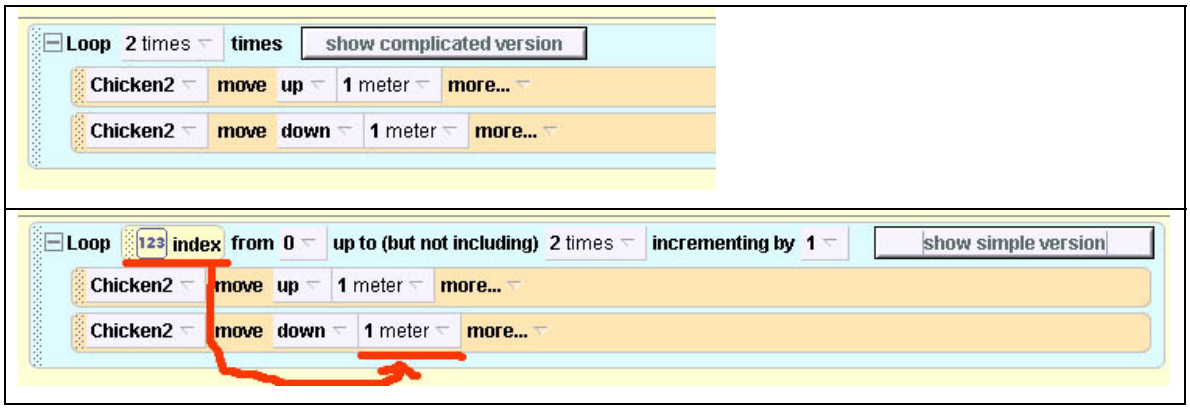


Figure 2.8: By default, Alice displays a simplified for loop but users can press the "show complicated version" to gain access to the loop count variable (i.e. index).

To create a loop, the user drags the "Loop" tile, drops it into the method editor and selects the number of times he or she wants the loop to execute. By default, the Alice loop is presented as a count loop. However, a "show complicated version" button displays the Loop in the style of a for-loop and allows users to access the loop count variable (see Figure 2.8).

If/Else Statements

To create an If/Else statement, the user drags the "If/Else" tile into the method editor and selects "true," "false," or a Boolean expression, such as a variable or a parameter. Users can replace the "true" or "false" value with a function that returns a Boolean value. In addition to methods, all objects in Alice have functions, essentially questions that they can ask about the state of the world. These are displayed on a tab marked "Functions" in the details area. The user can drag a Boolean function and drop it onto top of an If-statement's conditional to replace it (see Figure 2.9).

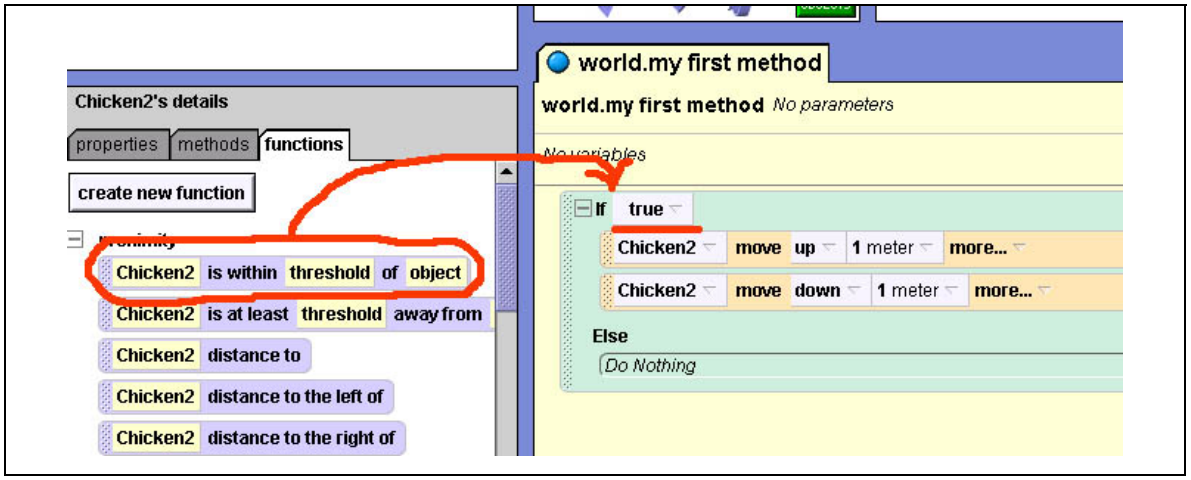


Figure 2.9: Users can drag in functions that return Boolean values onto the condition (i.e. "true") for an If-statement to replace it.

While Loops

To create a while loop, the user drags the "While" tile into the method editor, drops it, and selects true, false, or a Boolean expression as the while condition. As with If-statements, the user can replace the condition with a function that returns a Boolean value.

Do Together

Unlike many programming languages, Alice provides a simple construct called a Do Together for executing multiple methods simultaneously. To create a Do Together, the user drags the "Do Together" tile into the method editor and drops it. The user can drag any lines of code that he or she wants to execute in parallel into the Do Together.

Do In Order

By default, all methods execute the instructions inside them in order. However, when creating animations, it is sometimes necessary to have a sequential list of actions happen in parallel with another action or list of actions. For example to create a jump forward animation, a user might want an object to move up and then down while moving forward the whole time. To create a Do In Order, the user drags the Do In Order tile into the method editor and drops it. Do In Orders are most commonly found inside Do Togethers.

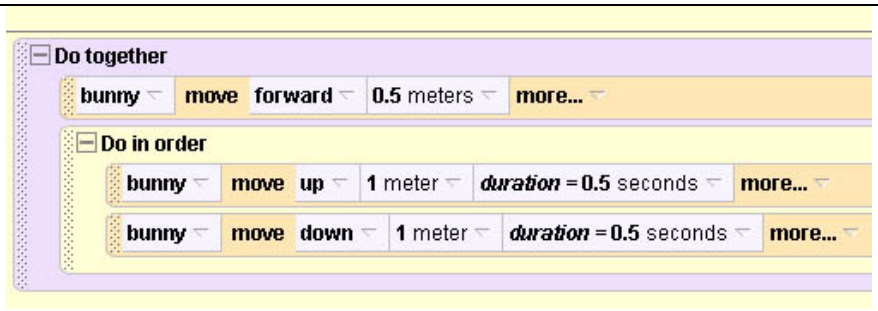


Figure 2.10: An example of a nested Do in order inside of a Do together which causes the bunny to jump forward by moving forward while moving up and down.

For All In Order

Sometimes, it is useful to iterate over a list of objects and do something with them, one at a time. To create a For All In Order, the user drags the "For All In Order" tile into the method editor and drops it. The user can then select a pre-existing list or choose to create a new one. Lists can contain numbers, strings, objects, colors, etc. The For All In Order construct creates a tile that represents the current element in the list that users can drag in to the For All In Order to use.

For All Together

This construct is the parallel equivalent of For All In Order. Rather than iterating through the items in the list one at a time, For All Together performs the actions inside it for all of the list items simultaneously.

2.2.4 Creating New Methods

One of the challenges in learning to program is in learning how to structure programs so that as they grow in length the programmer can quickly navigate through the code. If a user wants to change some aspect of their program, it is important that they be able to locate the relevant code for that aspect of the program. In object-based and object-oriented languages, programs are organized into methods associated with particular objects. A collection of lines of code that instructs a clown character to perform a back flip would be placed inside a method called "back flip"

To create a new method for an object in Alice, a user clicks on that object's tile in the object tree. In the details area, at the top of the methods panel, there is a button marked "Create new method". Users can press the "Create new method" button and type in a name for their new method. Alice will open a method editor for their new method and create a new tile in the object's list of methods that users can drag into programs to call their new method. Figure 2.11 below shows the Alice interface after the user has created a new method called "triple jump" for the iceSkater. In the editor area, Alice has opened a new method editor for "triple jump." Initially, "triple jump" will do nothing; users can drag command tiles into the triple jump's editor to define what it means for the iceskater to perform a triple jump. Users can call the method "triple jump" by dragging the "triple jump" tile into the method editor.

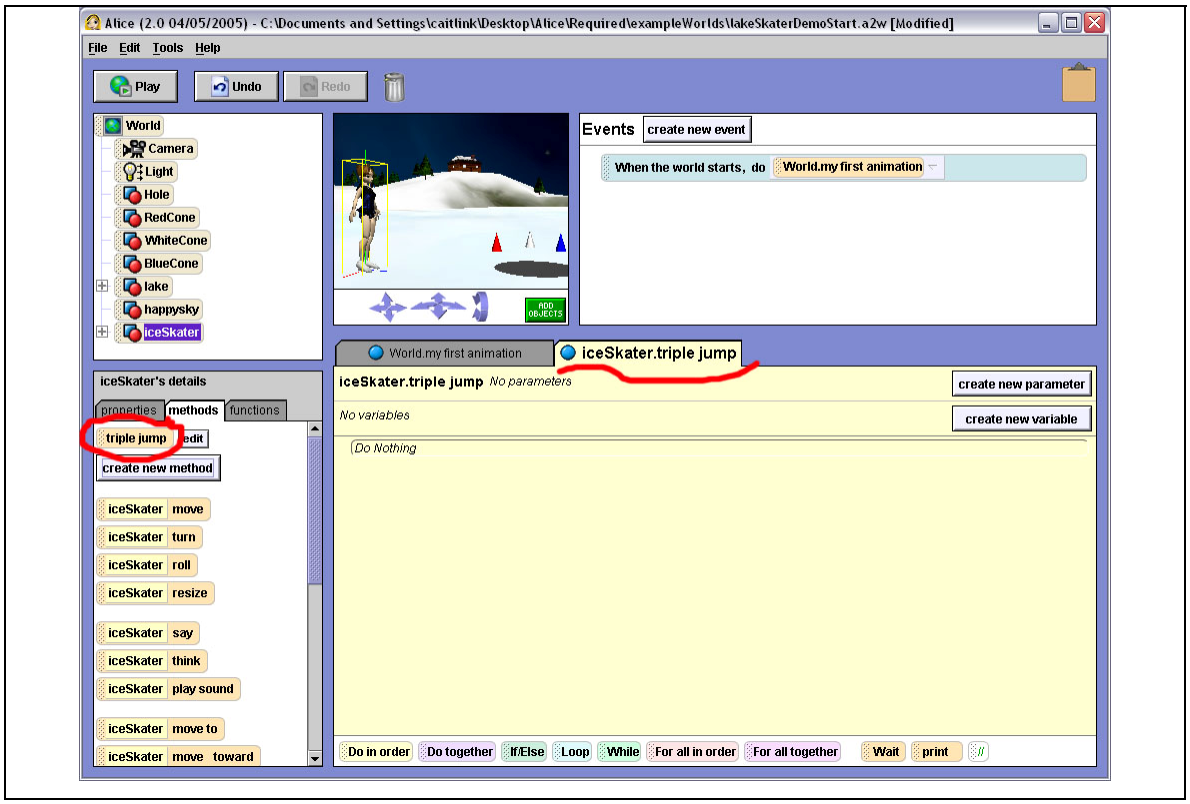


Figure 2.11: The interface after the user has created a "triple jump" method for the iceSkater. Alice has created a tile the user can drag into their program to call "triple jump" and opened a method editor where the user can define the behavior for "triple jump."

2.2.5 Creating methods with parameters

Alice allows users to create methods that take parameters. Parameters in Alice can be numbers, strings, objects, colors, etc. For example, a user might want to create "jump and spin" method for the iceskater and pass in how many times the iceskater should spin in the air as a parameter. Below is a simple version of "jump and spin" without a parameter. To add a parameter to this method, the user can click on the "create new parameter" button. A dialog box will appear which asks the user to choose a type for the parameter and give it a name. For the "jump and spin" method, the user might create a number parameter called "how many times."

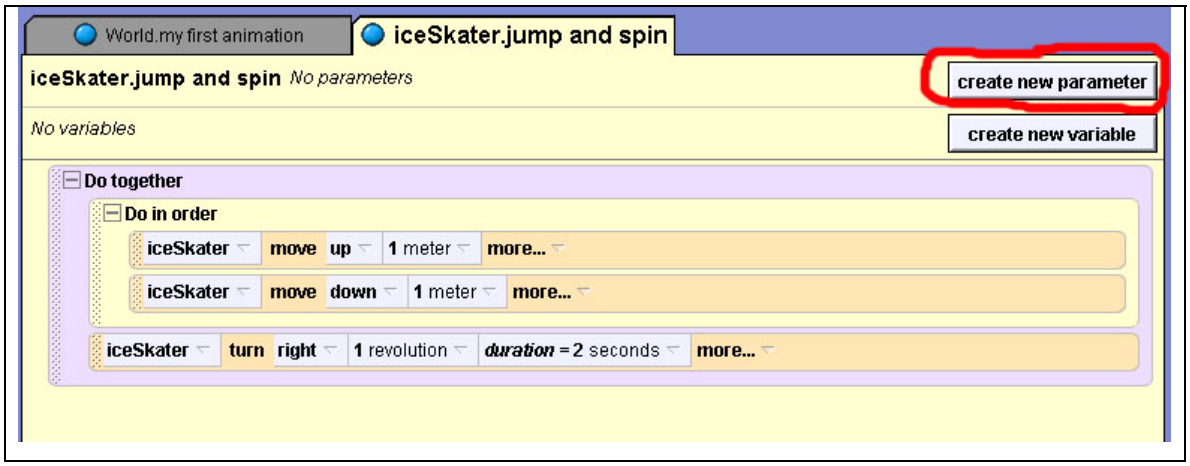


Figure 2.12: To add a parameter to a method, users can click on the "create new parameter" button in the method editor.

When the user clicks ok, Alice will add a tile which represents the new parameter to the method editor. The user can then drag the parameter tile and drop it where he or she would like to use the parameter. In the case of the "jump and spin" method, the user would drag the "how many times" tile and replace the "1 revolution" in the line "IceSkater turn right 1 revolution."

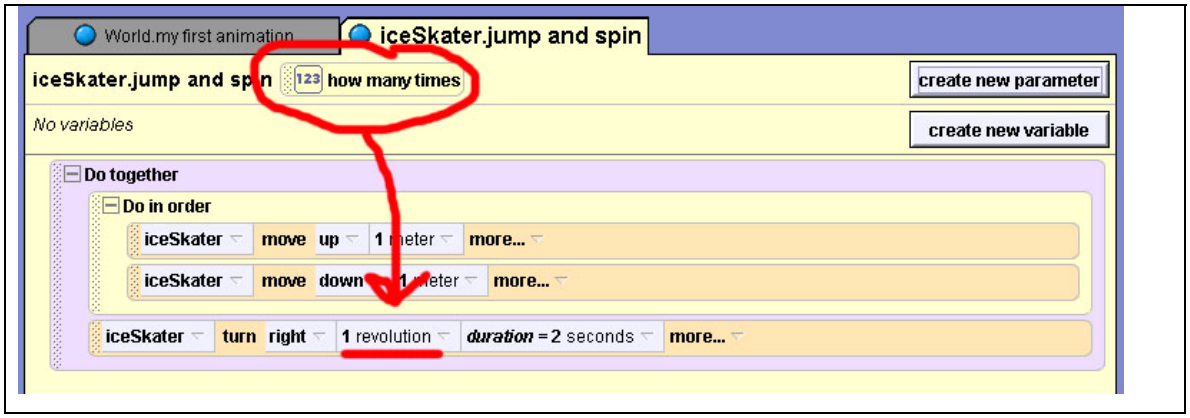


Figure 2.13: To make the iceSkater turn right "how many times," the user can drag the "how many times" tile and drop it on top of "1 revolution" to replace it.

When Alice adds a parameter to a method, it automatically updates everywhere that method is called. Before the user adds a parameter, a call to IceSkater's "jump and spin" method looks like Figure 2.14. When the user adds the "how many times" parameter to the "jump and spin" method, Alice also adds it to any places that the method is called.

Also, when users create new calls to "jump and spin", Alice will automatically create a pop-up menu where users can select values for any parameters.

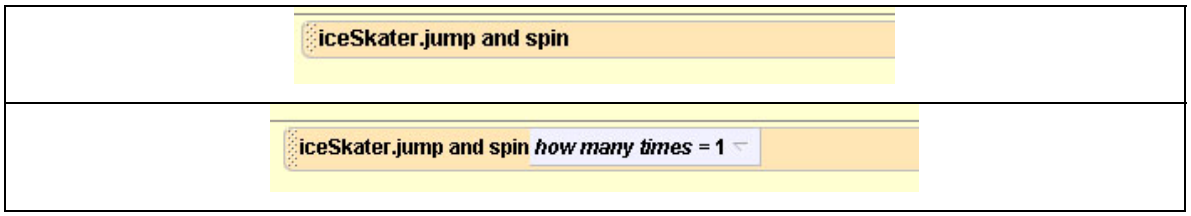


Figure 2.14: The method call to "iceSkater.jump and spin" without (above) and with (below) a parameter that controls how many times the iceSkater should spin.

Alice 2.0 provides mechanical supports that ease the process of learning to program. By constructing programs via drag and drop, users can concentrate on understanding how programming constructs work rather than learning syntactic rules. Because Alice programs are animations, users can see the program execute, making it easier for users to find and fix errors.

Alice provides most of the programming constructs that are typically covered in an introductory programming class. In Alice, users can create programs that use loops, if-then-else statements, methods, parameters, and parallelism.

2.3 Storytelling Alice

Alice is a good environment for learning how to program. Unfortunately, most middle school girls are not intrinsically interested in learning to program a computer. If you walk into a classroom of middle school girls and ask how many want to learn to program, only a few hands go up. However, I found that most girls are interested in learning how to create animated movies. Guided by user tests with more than 200 middle school girls, I created Storytelling Alice to introduce computer programming to girls as a means to the end of creating an animated movie. Storytelling Alice has:

1. a set of high-level animations that more closely match the kinds of actions girls envision using in their stories.
2. a gallery that is designed to help girls find story ideas.
3. a tutorial that introduces girls to the process of creating programs in Alice using story examples.

Subsequent chapters describe my user testing methods and the changes that I made to Alice 2.0 in support of storytelling.

2.4 Motivating Novice Programming Environments

There is a long and rich history of research on designing programming languages and environments to broaden the pool of people who learn to program computers. The majority have focused on simplifying the process of learning to program using a variety of techniques from improving textual programming languages to creating environments that allow users to author programs without making syntax errors (see Chapter 11: Programming Languages and Environments for Novice Programmers)(Kelleher and Pausch 2005). A smaller number of systems try to provide a motivating context for learning program.

Three systems have attempted to motivate computer programming through competitions: in AlgoArena (Kato and Ide 1995) users program sumo-wrestlers to fight tournaments; in Robocode (Nelson 2001) users program battletanks for a "fight to the finish", in Rapunsel (Flanagan, Nissenbaum et al. 2005) users program competitive dancers.

Other systems enable users to create animations and video games. Although it is possible to create animated stories in these systems, most provide users only general graphics capabilities. It may be difficult to create stories from basic graphics commands because of the number of steps required. Several systems allow users to create 2D animations by moving sprites and changing the graphical image associated with the sprite. Stagecast Creator enables users to create 2D games and simulations by specifying graphical before and after conditions (Smith, Cypher et al. 2000). HANDS allows children to create 2D games using a programming language designed to more closely match the ways in which non-programmers describe the solutions to programming problems(Pane, Myers et al. 2002). In Toontalk, users can create 2D video games through demonstrating their programs in an animated 3D virtual environment (Kahn 1996). Scratch (Maloney, Burd et al. 2005) and Squeak EToys (Kay) enable children to create 2D animations and games through dragging and dropping graphical tiles. StarLogo TNG (Kloper and Begel 2006) allows users to create games and simulations by moving 3D models in a virtual world;

animations that require a character to move body parts such as walking or waving hello must be created using external software (Kloper and Begel 2006).

Storytelling Alice is one of few novice programming systems to focus on making the process of learning to program more motivating. While other systems have attempted to enable novice programmers to use general-purpose graphics and animations in their programs, I am not aware of any systems that focus on enabling the creation of 3D animated stories.

Chapter 3 Formative User Testing

3.1 Introduction

The development of the storytelling version was guided by extensive user testing with more than 200 middle school aged girls (and more than 50 boys). This chapter describes the participants in the formative user testing of Storytelling Alice as well as the tools and techniques I used in gathering information about users' storytelling needs and their interaction with Storytelling Alice. I estimate that I created and tested 15 different versions of Alice in the process of developing Storytelling Alice.

3.2 Participants

One of the difficult aspects of research involving children is finding enough children who are representative of typical middle school children. Participants in the formative testing of Storytelling Alice were largely from Western Pennsylvania (with some from Atlanta, GA and Houston, TX). Typical middle school children may vary a little bit from one region of the country to another. Through the course of my thesis work I worked with children recruited from three main sources: 1) Science, Technology, Engineering, and Math (STEM) summer camps 2) home-schooling groups and 3) the Girl Scouts.

3.2.1 Girls in STEM Camps

Girls from two different STEM camps participated in the formative testing of the storytelling version of Alice.

32 girls aged 12-14 from Georgia Tech's Technology, Engineering and Computer (TEC) Camp participated in an Alice class as part of a week-long camp during the summer of 2003. The Alice class met for four 1.5 hour long sessions. To participate in the TEC camp, girls were required to complete an application that included an academic transcript, an essay on a technology-related topic, and teacher recommendations. I do not have specific demographic information about these students, but the application process was designed to identify bright girls who had an interest in technology and engineering.

Table 3.1: Academic Demographics for the Houston Museum of Natural Science Summer Alice Workshop Participants

		Total Number
Number of Participants		19
Ages	High:	15
	Low:	10
	Mean:	11.4
	Standard Deviation:	1.1
Grade in School	High:	9
	Low:	6
	Mean:	6.4
	Standard Deviation:	0.8
School Type	Public:	10
	Private:	8
	Home-school:	1
Academic Performance	Mostly A's:	9
	A's and B's	7
	Mostly B's:	1
	B's and C's:	1
	Mostly C's:	0
	C's and D's:	0
	Mostly D's and below:	0
	No Answer:	1

During the summer of 2004, 19 girls participated in a 3-day Alice workshop offered through the Houston Museum of Natural Science (HMNS). HMNS workshops and camps have open enrollment. Consequently, the HMNS girls represented a broader spectrum of abilities than the TEC camp girls. Academic and computer-related demographic information is presented in the tables below. Girls ranged in age from 10 to 15, with most between 10 and 11. All but 3 of the students reported their grades as being either all A's or A's and B's. Slightly more than half of the students attended public schools, one was home-schooled, and the rest attended private schools.

More than half of the girls who participated in the HMNS camp ranked themselves as "very good" or "excellent" at using computers. More than 1/3 of the students had made a web page and 2 of the 19 students had previously written a computer program.

Table 3.2: Computer-related Demographics for the Houston Museum of Natural Science Summer Alice Workshop Participants

		Total Number	Percentage
Number of Participants		19	
During the last week, how often did you use a computer for any purpose?	High:	25	
	Low:	0	
	Mean:	7	
	Standard Deviation:	6.3	
What do you use computers for?	Only schoolwork:	0	0.0%
	Mostly schoolwork, some fun:	1	5.3%
	Equally for schoolwork and fun:	4	21.1%
	Mostly fun, some schoolwork	8	42.1%
	Only fun:	2	10.5%
	No answer	4	21.1%
Have you ever written a computer program?	No:	6	31.6%
	Yes:	2	10.5%
	Don't know:	7	36.8%
	No answer:	4	21.1%
Have you ever made your own web page?	No:	8	42.1%
	Yes:	7	36.8%
	Don't know:	0	0.0%
	No answer:	4	21.1%
What is your skill level at using computers?	Poor:	0	0.0%
	Fair:	3	15.8%
	Good:	2	10.5%
	Very good:	8	42.1%
	Excellent:	2	10.5%
	No answer:	4	21.1%

Recruiting from STEM camps has several problems: STEM campers tend to be brighter (they have chosen to attend an educational camp and may have gone through an explicit application processes) and more receptive to STEM related activities than average. However, STEM campers were valuable early testers of Storytelling Alice. They tended to show more persistence than I have seen in my groups overall. Where a typical middle school girl might give up after encountering a few difficulties, STEM campers were more willing to push through problems they encountered. Further, they were able to reflect on

ways to improve Alice. In fact, some of the most insightful comments about Alice came from STEM camp kids. For example, one girl pointed out that we should carefully select the set of 3D objects that come with Alice to maximize their expressive potential and minimize the need for repetitive work; she stated that adding a lot of trees to create a forest was boring and unexpressive, but selecting characters or adding details to a character's bedroom had more potential for self-expression.

3.2.2 Home-schooled Students

In addition to girls attending STEM camps, I worked with a variety of home-schooled students.

10 students, 7 girls and 3 boys, recruited by a colleague who home-schools his children, came to Carnegie Mellon for 1 hour per week for a semester. The children ranged in age from 10 to 17 and 6 were African-American students.

31 students, 13 boys and 18 girls, from two Pittsburgh home-schooling groups participated in two classes organized through local home-schooling organizations. The classes met for a total of 8 hours, divided into 1 or 1.5 hour sessions. The students ranged in age from 10 to 16. Most said that math and/or science were their favorite subjects. A larger proportion of girls listed language arts, arts, or history/government related subjects as their favorite subject than boys. Girls also tended to have different hobbies than boys. Where 9 of the 13 boys listed computers as a hobby, only 2 of the 18 girls did. Girls most commonly listed arts (13 of 18) and sports (13 of 18) as hobbies. According to self-reports, the home-schooled boys were more frequent computer users than the girls. All but 1 of the boys reported using their computer daily. Among the girls, fewer than half of the girls reported using the computer everyday.

Table 3.3: Academic Demographics for Home-schooled Participants.

	Boys	Girls	All Users
Number of Participants	13	18	31

Ages	High:	16	16	16
	Low:	10	10	10
	Mean:	12.23	12.74	12.45
	Standard Deviation:	0.66	0.55	1.59
Hobbies	arts:	2	13	15
	computers:	9	2	11
	construction:	3	0	3
	sports:	5	13	18
	media:	1	4	5
Favorite Subjects	math/science:	11	8	19
	language arts:	1	5	6
	arts:	1	4	5
	history/government:	1	4	5
Computer Usage	everyday	13	6	19
	every other day	0	6	6
	weekly	1	3	4
	occasionally	0	3	3

Although I did learn some valuable lessons from the home-schoolers who worked with Alice, I moved away from working with home-schooled kids for a variety of reasons. The home-schooling community in Pittsburgh is not large enough to serve as a user-testing pool. Both home-schooling groups struggled and, ultimately, failed to find 20 kids (with at least half girls) between the ages of 11 and 15 to participate in the Alice classes. However, I also found that the home-school community includes a larger than normal proportion of students I would describe as atypical: several of the students in the two Alice classes had learning disabilities or special needs. Further, the high degree of parental involvement in the home schooling community made user testing difficult. For example, I tried to pair girls with girls and boys with boys. The parent of a shy girl told me that her daughter would only participate if she could be paired with her brother. In another case, the mother of a girl with a learning disability insisted that her daughter be paired with a friend in the class, causing me to have to reshuffle groups after user-testing had started. Due to the small numbers of girls in these classes, I felt it was necessary to comply with the requests. The home-school classes were fairly early in my process; both were held in the spring of 2004. At the time, I was still identifying areas in which Alice could be improved for girls so parents' demands did not seriously impact what I was able to learn from these classes. However, for the later formative evaluations and the summative evaluations which required randomization, the home-schooling community would have difficult to work with.

3.2.3 Girl Scouts

The majority of users who participated in formative user testing were drawn from local Girl Scout troops. I recruited troops by placing a short paragraph advertising 4-hr computer workshops in which Cadette troops (typically composed of students in grades 6-9) could help test the Alice program and learn a little bit about computer programming in the Community Bulletin, a newsletter the western Pennsylvania Girl Scout office sends out to troop leaders and activities coordinators in the local area. Interested troop leaders contacted me to arrange a time for their Girl Scout troops to participate in a workshop. To encourage a broad spectrum of girls to participate, I offered a $10 donation to the troop for each girl who participated. The financial incentive encouraged the participation of girls who did not see themselves as "computer people." Although I did not formally ask people their reasons for signing up, my impression is that the majority of troops participated for fundraising purposes. Many of the troops were explicitly saving money towards an activity or trip; troops mentioned goals including trips to New York and Washington, museum tours with docents, camping and ski trips. Girl Scout troops earn $0.50 per box of cookies sold, so $10 per girl for an afternoon was appealing to a lot of troops. Some troops participated because the workshop provided an opportunity to visit a college campus or to give girls educational experience that they might not get in school.

The girls who participated were largely between the ages of 11 and 15. However, I allowed troops that had a small number of girls slightly older or younger than my target age range to participate. Consequently, my participant pool includes a small number of girls either slightly younger or slightly older than my target age range of 11 to 15. The average girl who participated was between 12 and 13, in the 7^{th} grade, and attended a public school. Most girls self-reported that their grades were either "mostly A's" or "A's and B's." This raises a question about whether or not my participant pool contained a representative sample of academic abilities. While I cannot answer this question definitively, based on my interactions with girls who participated in my user testing sessions, I can say that my participant pool included a wide spectrum of girls with varying ability levels. Girls do not have to be academically gifted to participate in Girl Scouts. In fact, several troops included members who were autistic or had other learning

disabilities. Based on the nature of self-reports, it seems likely that many girls were "rounding up" when describing their grades.

Table 3.4: Age and Academic Demographics for Girl Scout Participants.

		Total Number
Number of Participants		165
Ages	High:	16
	Low:	10
	Mean:	12.6
	Standard Deviation:	1.2
Grade in School	High:	11
	Low:	5
	Mean:	7.3
	Standard Deviation:	1.3
School Type	Public:	155
	Private:	10
	Home-school:	0
Academic Performance	Mostly A's:	70
	A's and B's	65
	Mostly B's:	10
	B's and C's:	17
	Mostly C's:	1
	C's and D's:	1
	Mostly D's and below:	0
	No Answer:	1

Most of the girls who participated in the formative testing of the storytelling version of Alice ranked themselves as either "good" or "very good" at using computers on a scale that ranged from "poor" to "excellent". Approximately one third of my participants had made their own web page and 11 of the 165 girl scouts had previously written a computer program. Most girls reported that they used computers either "equally for schoolwork and fun" or "mostly for fun and sometimes for schoolwork". When asked how often girls had used a computer in the last week, there was a wide variation in answers. 10 of the 165 girls had not used a computer at all in the last week. 12 of the 165 girls reported using the computer for more than 30 hours over the last week. 97 of the 165 girls reported using the computer between 1 hour and 10 hours over the last week.

Table 3.5: Computer-related Demographics for Girl Scout Participants.

		Total Number	Percentage
Number of Participants		165	

During the last week, how often did you use a computer for any purpose?	High:	75	
	Low:	0	
	Mean:	9.35	
	Standard Deviation:	11.47	
What do you use computers for?	Only schoolwork:	3	1.8%
	Mostly schoolwork, some fun:	14	8.5%
	Equally for schoolwork and fun:	60	36.4%
	Mostly fun, some schoolwork	76	46.1%
	Only fun:	4	2.4%
	No answer	8	4.8%
Have you ever written a computer program?	No:	112	67.9%
	Yes:	11	6.7%
	Don't know:	34	20.6%
	No answer:	8	4.8%
Have you ever made your own web page?	No:	95	57.6%
	Yes:	55	33.3%
	Don't know:	7	4.2%
	No answer:	8	4.8%
What is your skill level at using computers?	Poor:	2	1.2%
	Fair:	23	13.9%
	Good:	62	37.6%
	Very good:	57	34.5%
	Excellent:	13	7.9%
	No answer:	8	4.8%

3.2.4 Why not schools?

One of the obvious potential sources for recruiting girls is local public schools, as students drawn from a broad collection of local public schools would be likely to be a representative sample. I approached a few schools, but found that most of the schools were focused on preparing their students for the Standards of Learning (SOL) tests required by the No Child Left Behind Act (NCLB 2002). As computer science is not a required part of the middle school curriculum, educators were hesitant to devote classroom or extra-curricular time to anything not covered in the SOL. Even among interested schools (mostly schools where I had an inside contact), I found it difficult to set up user tests. In one such school, the principal and computer lab coordinator were interested in offering an Alice-based activity after school, but the activity coordinator did not want another activity to supervise.

I elected not to approach private schools because I felt it was important to work with a representative sample of girls. The students who attend most of the private schools locally are not a representative sample.

3.3 Types of Data

The design of the storytelling version of Alice was based primarily on two sources of information from formative evaluation: classroom observations of girls using Alice and storyboards that girls created prior to seeing or using Alice. To supplement my primary information sources, I used short-answer survey questions, logs of the actions that users performed in Alice, and classroom discussions.

3.3.1 Classroom observations

Classroom observations were one of the main sources of information guiding the development of the storytelling version of Alice. I took notes on users' questions and comments, as well as things I noticed while watching them work. During user testing, I answered users' questions. When not answering questions, I looked for users who were clearly having a very positive experience (typically evidenced by laughter or attempts to get the attention of a peer to show something) or a very negative experience (typically evidenced by sighing or muttering) to observe and tried to get a sense for the causes of their positive or negative experience. At other times, I simply watched users interact with Alice. Classes ranged from as few as 3 students to as many as 20. However, I had ten laptops available for user testing, so there were no more than 10 Alice sessions to observe in each user testing session.

I performed my initial user tests with pairs of girls working together to create stories in Alice. By pairing the girls together, I hoped to gain more insight into both the problems they encountered in attempting to create stories in Alice and their approaches to solving those problems. Observing the actions and listening to the conversation between a pair of users working on a task (e.g. two-person talk aloud protocol) is a common and often successful technique for gathering usability data (Nielson 1993). However, I found that the two-person, talk-aloud protocol is ill-suited for creative tasks like storytelling because it creates a string of goals related to negotiating a creative vision and largely unrelated to

the software program being evaluated. I found that most of the conversation between pairs was about the story rather than the software, making it more difficult to capture problems with the software. The process of negotiation between pairs in developing the story also has the potential to significantly change the kinds of stories that users try to build, making it harder to determine what kinds of stories girls envision creating. Further, I found that girls had a difficult time negotiating a shared vision. While there were some instances in which working together appeared to result in a better story, there were more instances in which working in pairs was problematic. I observed cases in which one member of the pair essentially took over the process and the other half of the pair contributed little or nothing to the story or its implementation. In other pairs, both members of the pair were so concerned about each other's feelings that they were hesitant to state an opinion. Often these pairs created stories that neither one felt motivated to build. For a creative process like storytelling, collaboration is difficult. Even among professional writers, we do not see many jointly authored stories.

One alternative to observing pairs of users is to allow users to work on their stories alone. However, users working alone do not naturally verbalize their current goals and strategies for solving a problem. Asking users to talk aloud as they work through a task is awkward for many users and reminders to verbalize their thought process can distract users from their task. Despite the problems associated with gathering usability information from users working individually, I moved from having users work in pairs to having them work individually. This removed the necessity for girls to negotiate a shared vision for their story. However, it was important that girls still be able to talk to each other as their conversations can provide information. To encourage girls to talk to each other, I configured their workspace such that they could easily interact with each other. I placed the laptop computers close together, such that girls could see their neighbor's screen. The fact that our user testing was done using laptop computers was important because the size and form factor of the laptops enabled girls to easily turn their computers or adjust the screens to show what they were working on to another girl at the table. During my user testing sessions, girls frequently showed each other their animations. Girls asked how to create actions they saw in other's worlds. Girls' conversations with each other as they

created their stories in Alice provided valuable insight into what they wanted to create and their assumptions about how to go about implementing their visions in Alice.

In addition to girls' conversations with each other, their questions proved to be another valuable source of information. In user tests, researchers often assign users specific tasks to complete (Gomoll). In my user testing sessions, I asked girls to create a story, but they decided what specific story they would like to create. By giving girls control over their end-goal, I was giving them the power to change that end-goal as they worked. Often, this occurred when a girl decided that some aspect of her original story idea was going to be too hard or too frustrating to create in Alice. Girls often asked questions shortly before giving up on a current goal and forming a new goal. I encouraged girls to ask questions during user testing sessions.

In my user testing sessions, there were sometimes a small number of shy girls who seemed hesitant to ask for help. I found that for these girls, if I could offer a quick but useful pointer in a case where their goals were clear from their actions (for example, in a case where a girl was dragging in a lot of copies of the same object from the gallery, I might show her the copy tool) that the girls seemed to be more likely to ask questions or request help if they encountered problems later on in the session.

3.3.2 Storyboards

Unlike a lot of user testing situations in which a researcher provides users with an explicit list of tasks to complete, the nature of my research required that girls set their own goals for the stories they would like to create in Alice. One of the problems with allowing users to set their own goals is that users can *change* their goals at any time. In my user testing sessions I found that while girls often started with lofty goals for the stories that they wanted to create, they quickly began to adapt those goals based on their perceptions of what kinds of tasks are difficult in Alice. Consequently, the girls' initial goals were particularly valuable. To capture those goals, I experimented with having girls create storyboards before they ever saw Alice. The storyboards that girls created were an invaluable source of information to me in determining what the Alice system needed to support in order to enable girls to create the kinds of stories that they envisioned.

However, it took several iterations for me to find a storyboard creation process that resulted in storyboards from which I could extract actionable information.

3.3.2.1 Attempt 1: Handout on Storyboarding for Movies

As part of the TEC camp held at Georgia Tech, I gave girls a handout which explained the purpose and process for creating storyboards in the form of a storyboard (see Appendix A). Originally designed to educate middle and high school students about creating their own film-based movies, it discussed everything from the level of detail that a single frame of a storyboard should contain to where one should position the camera to in order to elicit different kinds of emotional responses from the viewer. I asked girls to read the handout on storyboarding and then create a storyboard (using an unlimited supply of blank paper) for the movie that they wanted to create.

In analyzing these storyboards, only one thing was clear. The users expected to be able to create multiple scenes within Alice. 9 of 11 of the storyboards contained multiple scenes. Unfortunately, most of the storyboards did not provide sufficient detail about what girls expected their characters to do. In some storyboards, girls drew a single storyboard frame to represent an entire scene. In others, girls simply wrote a sentence or two about what happened in the scene. Girls tended to only provide a general description of what happened in the scene. For example, a girl might write "Jeremy finds out that Julie is dating someone else" but omit important details like where Jeremy is, who tells him, what Jeremy does when he hears the news, etc.

3.3.2.2 Attempt 2: Storyboarding an Example Story

I created a second handout in which I provided an example story in text format, discussed what information the movie needed to show a viewer in order to communicate what was happening in the story, and then showed the corresponding storyboard (see Appendix A). As with the previous handout, I gave users (in this case home-schooled children taking a class at a local museum) the example story handout and asked them to read the handout and create a storyboard for their own story. They had an unlimited supply of blank paper to use.

My results were roughly the same as before. Some children drew and others wrote but few of the storyboards provided detailed information about the kinds of actions that children expected their characters to be able to perform.

3.3.2.3 Attempt 3: Guided Storyboard Creation Worksheet

Based on the two prior attempts with creating storyboards, it seemed clear that I needed a more structured process for storyboard creation. I created a worksheet packet that guided children through creating storyboards in 3 steps (see Appendix C). In the first step, they wrote a single-paragraph description of the story they were planning to create. In the packet, I encouraged them to think of this paragraph as being similar to the description one might find on the back of the DVD box. In the second step, the worksheet directed girls to break their story into 3-5 separate scenes. For each scene, girls wrote a description of the setting, what happens during the scene (in 1-2 sentences), and the purpose for the scene (what the audience should learn from the scene). Finally, girls created a series of storyboard frames for each of their scenes. The worksheet provided 9 frames per scene and directed girls to both draw the frame and provide a short textual description of the action in that frame beneath it. In practice, most scenes contained 4 to 6 frames and accompanying textual descriptions.

The more structured approach to creating storyboards helped kids to flesh out their story ideas and provided some redundant information sources that made the storyboards easier to interpret. The storyboards contained three sources for actions that girls wanted to incorporate into their stories: 1) changes in the drawings between one frame of the storyboard and the next 2) the description of the action for each frame and 3) the textual description for the scene.

The drawings of the frames were a good source of information about the motions of characters around the set. In one frame a character might be talking with a friend and absent from the following frame. Based on this, it is reasonable to conclude that the character must have left the scene.

The textual descriptions often contained information about gestures that the characters should do. For example, one character might wave to another or raise his arms to indicate a victory. In looking through the storyboards, most of the information about what girls wanted their characters to do come from either the frame drawings or their textual descriptions. There were a small number of actions in scene descriptions which did not actually appear in the storyboards.

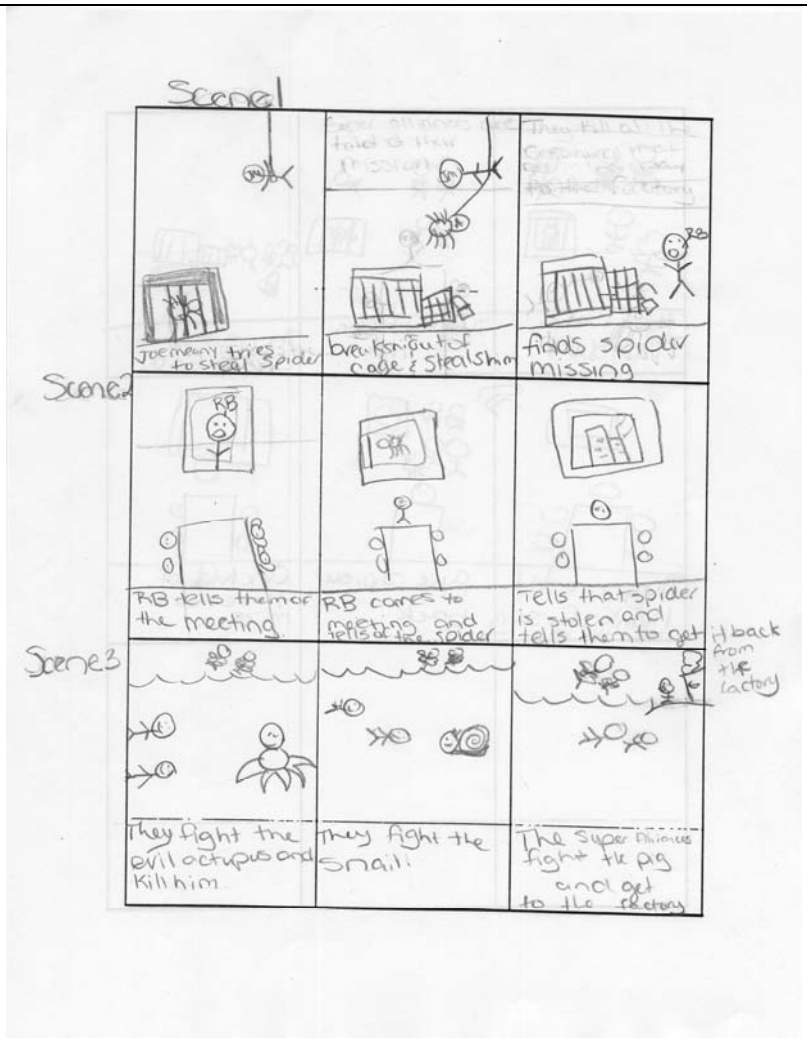


Figure 3.1: An example story-board created by a Girl Scout during formative testing of Storytelling Alice.

3.3.3 Alice logs

I instrumented Alice to record and time-stamp all of the changes that users make to their Alice worlds in a text file. These logs include information such as

- when a 3D model is added, deleted, or moved

- when a line of code is added, deleted, or moved
- when a parameter is changed
- when a new method, variable, or parameter is created
- when the world is played

The Alice logs form a detailed picture of everything the user did in the process of creating their Alice world. While I did not go through the logs for all of the users who participated in formative testing of the storytelling version of Alice, I did study the logs in two types of cases 1) girls who did not have obvious problems but seemed to have a less positive experience than their peers and 2) in cases where a girl had a clear goal she had asked for help in realizing which she later gave up on.

3.3.4 Classroom discussions

I experimented with having classroom discussions in which I asked user testing groups for suggestions on how we might improve Alice. I did this in two ways: 1) opening the floor to comments and suggestions and 2) breaking girls into small groups to discuss potential improvements to Storytelling Alice. I found that writing down girls' suggestions demonstrated to girls that their ideas were being taken seriously. Girls were more likely to offer suggestions when I was actively recording them. In general, girls' suggestions were only occasionally useful because asking how to improve Alice requires girls to think about the process of creating a story at a meta-level. Mostly I got comments about specific pieces of content that girls liked (a particular animation or character) or didn't like (the storyline in a tutorial or their favorite animal missing from the gallery). One girl in a group discussion talked at length about how she thought the second tutorial, which at the time was a story about a bunny being woken up by a cell phone, was too young for her. Yet, in another group a girl talked about how cute the bunny was and several other girls listed the same tutorial as one of the best things about Alice in their survey response.

3.3.5 Surveys

At the end of each session, I asked girls to complete a survey (see Appendix A). The survey questions varied. I used the end survey as an opportunity to develop and refine attitude and programming achievement measures for the summative evaluation. In addition to the attitude survey and programming quiz, I asked users to answer several

open-ended questions. The actual questions changed over the course my evaluations. Questions included:

- What are the 3 best things about Alice?
- What are the 3 worst things about Alice?
- If you plan to talk about your experience today (e.g. the Alice workshop), what will you say?
- What should we do to improve Alice?
- What are the 3 most frustrating things you tried to animate today?

I found that in the early user tests, asking users to list the 3 worst things about Alice helped to identify the worst usability problems; users often listed bugs in the system or processes that were frustrating. As I addressed those problems, girls' answers to these questions became less useful. Towards the end of the user testing, I got a fair number of students who would list "nothing" or talk about a particular character animation that they did not like (e.g. [the character] Joe Meanie walks funny). One student complained about the dialog box that reminds students to save their work every 15 minutes. Users rarely listed missing content or functionality as one of the worst things.

Asking users about functionality or content they wanted in Alice was hit or miss. Occasionally, users would provide an insightful comment ("You should make simple stuff like walking/running easier") or identify a particular usability problem that frustrated them ("You should make the animation area scroll when someone drags a new animation to the bottom"). Typically, the suggestions that appeared in surveys also appeared in observing users working with Alice. The majority of functionality and content suggestions were very general ("provide more stuff" or "make it less frustrating") and hard to act on without additional information.

3.4 Methods

Over the course of developing Storytelling Alice, I focused on one big problem at a time and adapted the methodology to get at the focal questions for that problem. In all user testing sessions users completed a version of the tutorial and built a story in Alice. Most of the sessions included users creating storyboards either before seeing Alice or after completing the tutorial and before building an Alice world, and most users completed a

Broadly speaking, my work on the storytelling version of Alice was divided into three areas:

1) Creating a tutorial that demonstrates how Alice can be used for storytelling while introducing new users to the process of creating Alice worlds

2) Identifying and developing storytelling supports that enable girls to create the kinds of stories they envision in a reasonable period of time.

3) Identifying ways in which the collection of characters and scenery that comes with Alice (the gallery) can help girls find a story idea.

I found that it was easier to make progress by focusing on one of the three areas at a time. The table below shows a schedule of the development of the story version of Alice.

Table 3.6: Development Schedule for Storytelling Alice

Tutorial	Prior to January 2003	Develop and test on-line tutorials that demonstrate that Alice can be used for storytelling using Generic Alice animations.
Content	January – May 2003	Working with an ETC (ETC) Master's student, a team of undergraduates, and 10 home-schooled kids I worked to develop themed sets of 3D models (i.e. Story Kits) containing scenery and characters supported by character-specific animations to help kids find and successfully create a story.
Content	July, 2003 January-May 2004	I created and modified a story-gallery based on experiences with a larger group of students including campers from Georgia Tech's TEC Camp and local home-students who had a longer period of time in which to work on their stories. In these sessions, students were not restricted to using content from a single Story Kit in their stories.
Higher-Level Animations and Scene Support	July, 2004 September – May 2005	I analyzed what supports girls needed to create stories and then developed, tested and refined scene support and higher-level animations within Alice.
Tutorial	June 2005	The new storytelling animations made the tutorial out-of-date. I created a new second tutorial showcasing the animations users commonly needed to create the kinds of stories they envisioned

Content	July, August 2005	The new storytelling supports made it possible to provide better animations with characters. I took a pass through the story gallery, removed animations no longer necessary because of the new animation set, improved the remaining animations, and creating new ones where appropriate.

3.5 Usability Changes

In developing and testing Storytelling Alice, I identified several usability problems that were unrelated to the story aspects of the system. These problems fell into two basic categories: 1) basic usability problems and 2) usability problems related to programming in Alice. I made small changes to address the non-programming related usability problems. Changing the programming model that Alice presents was outside the scope of my work, but I document my observations here so that they can be considered in the development of future versions of Alice.

3.5.1 Switching Between the Scene View and the Programming View

In designing Alice 2.0, the Alice team (of which I was a member) decided to create a modal interface in which the user is either programming or laying out the scene. When Alice starts, the interface is in the programming view. To get from the programming view to the scene layout view, the user presses a small button labeled “add objects” that is displayed underneath the view of the 3D scene (the world area). To return to the programming view from the scene layout view, the user presses a button labeled “done”. Despite the fact that the tutorial introduces both of these buttons and uses them multiple times, new Alice users often have difficulty remembering where they are. Users get used to manipulating the list of objects in the object tree to select their current object of interest. Consequently, many users tend to gravitate towards the object tree for tasks involving selecting or adding new objects. To help users who search the object tree looking for a way to add new objects, I added an “add new objects” button to the area just above the object tree. Because the object tree is always visible, I change the text of this button to read “done adding objects” when the user is in the scene view. The “add new objects” button does appear to help people, because it is proximate to the object tree and

provides a single button for switching modes. Some users scroll down to the bottom of the object tree looking for a way to add objects (rather than looking at the space above the object tree). Placing the "add new objects" button at the bottom of the visible part object tree might provide further help to new users.

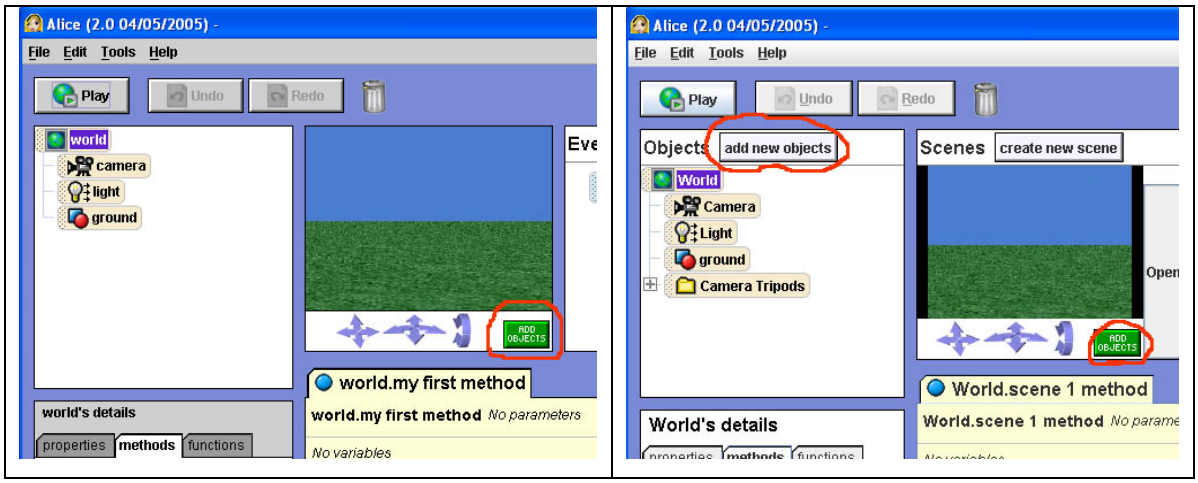


Figure 3.2: The "Add Objects" button in Generic Alice (left) and the two "Add Objects" buttons in Storytelling Alice (right).

3.5.2 Too Many Methods Displayed

In Alice 2.0, we ordered the methods for each object by their expected use; frequently used methods like "move" and "turn" were displayed at the top of the list while less frequently used methods like "orient to" were near the bottom of the list of methods.

In watching users creating programs in Alice, I found that many users will scroll through the list of methods and get lost at the bottom of the list. When they later need a common animation, they will only see the less common ones and frequently end up experimenting with something animations that are not well suited for their intended task because they have forgotten about the more common animations. To handle this, I added a collapsible pane of "infrequently used methods". This allowed me to shorten the list of methods that were in the list of available animations for each object, increasing the probability that the method most appropriate for a user's current task was either visible or nearby.

3.5.3 Losing Objects in the Scene

In Alice 2.0, there are two ways for a user to add a new object to their Alice world: 1) they can drag it into the scene or 2) they can click on the object and click the "add instance to world" button. When a user drags an object into their Alice world, Alice

displays a bounding box for the object, allowing the user to place the new object at an appropriate location in their world. When the user adds the object with the "add instance" button, Alice adds the object at a location saved within the model. These saved locations are hand-selected such that the object is easily visible from the camera's opening position. In practice, users frequently move the camera, creating situations in which the model's saved location is far from the user's current context. To ensure that the user knows approximately where the new object is, Alice 2.0 animates the camera to a position where the new object is visible and then animates the camera back to its last position.

I observed numerous sessions in which users were forced to drive the camera long distances to find add a new object they had added using the "add instance" button. To move the object to its intended position within the world, users would repeatedly drag the object to the edge of the camera's view and then move the camera a little bit. Driving the camera to the new object's location could sometimes take as much as 1 minute. Moving the new object from where Alice placed it to where the user wanted it often required several minutes. Users understandably found it extremely frustrating. To prevent this situation, I modified Alice such that the "add instance" button calculated the offset of the model's saved position to the camera's opening position. Then, rather than adding the model at its saved position in the world, Alice adds the new model at the calculated offset from the camera's current position. With this simple change, Alice places the vast majority of objects in the Alice gallery at a visible location. Users can click and drag the model to adjust its final placement in the scene but it is rare that they need to move the camera to accomplish this.

3.6 Issues about Programming

I encountered a small number of frequently occurring problems relating to how Alice presents programming. These are likely to be more problematic when users are self-taught than when Alice is used as part of a class and with the guidance of a teacher.

3.6.1 Editing methods is tempting and dangerous for new users

For my user population (and, I suspect for most user populations), the action of clicking on something is more natural than dragging. In my user testing sessions, users who were unsure of what to do often clicked on the edit buttons next to character-specific methods. By making it possible to edit a method by clicking a button, we tempt beginning users to do something that is likely to cripple them as they begin to explore Alice. By clicking the edit button, users are changing which method is selected in the method editor. This creates two problems 1) users often do not know how to get back to the previous method and 2) the method they have opened is probably not called in their program so code they add to it will not execute.

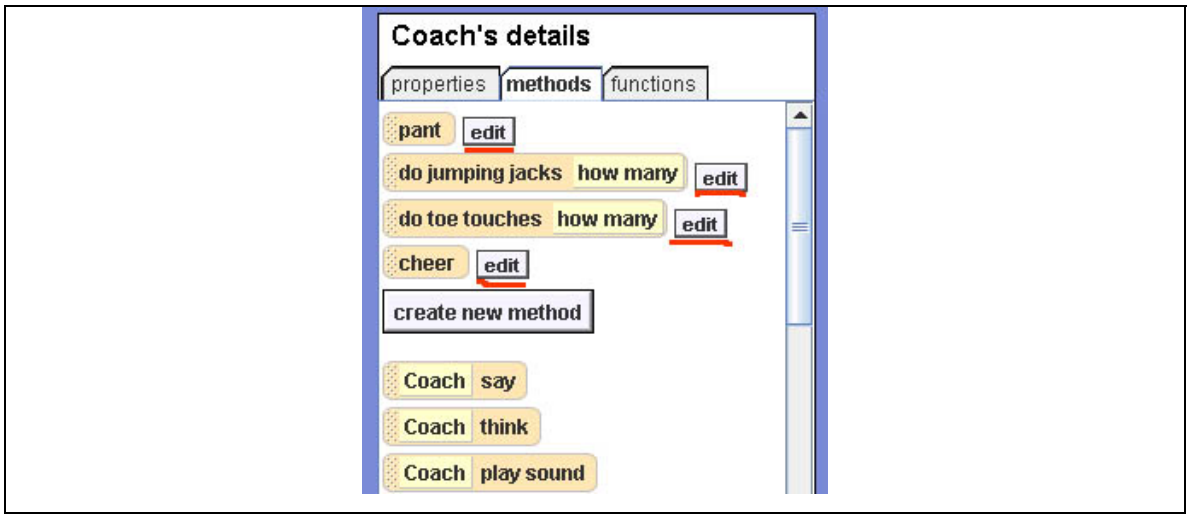


Figure 3.3: Many participants found it more natural to click on the edit buttons than to drag the method tiles.

3.6.2 New users are often forced to tackle events quickly

When users create a new method or edit an existing method without first understanding the concept of a method, they often fail to call that method within their program. After adding one or more method calls to the method they are currently editing, users press the play button and are frequently confused that the code they have just created does not execute. When a user presses the play button and the currently selected method is not called, Alice displays a message informing users that the method is not called in the program. Users nearly always dismiss the dialog box without reading it. Most users seem to mistakenly expect that Alice will play whatever method is selected in the method editor. I see two potential ways this problem can be addressed 1) provide a better

explanation of what is going on when the user hits play or 2) change the model so that in a default world, Alice will play the currently selected method.

When a user hits play and their currently selected method is not called by anything, they are almost always lacking information about how Alice decides what method to play. This provides a nice opportunity for the Alice system to introduce the user to the events area and point out which methods will be played. This could be done using the Stencils the interaction technique.

Typically, users who are going to open or create a method do so fairly early in the process of starting to use Alice independently. By trying to use this opportunity as a teaching experience, there is some risk that users will feel overwhelmed. We may lose users by trying to force them to understand too much too quickly. One alternative is to make the default Alice worlds come with a "when the world starts event" that plays whatever method is currently selected in the method editor. This would allow beginning users to more freely explore the system. This allows users to gain experience with Alice before they need to begin using events.

3.6.3 New users frequently make recursive calls.

New users often read through the list of available methods for a character and almost immediately try to create a new one without realizing that simply typing in the name for a new method does not result in the characters performing the actions the name implies. Having typed in the name for this new method, the next step is to drag it into their program. But, having created a new method, their new method is now open method editor. So, when users drag the new method in, they create a recursive call. Alice does put up a dialog box that asks users if they intend to make a recursive call. For the users who click the 'X' to close the dialog box or answer "no", this stops them from adding a recursive call. But, a non-trivial number of users will answer "yes." For the most part, these users do not yet understand the concept of methods, so explaining recursion to them is nearly impossible.

At least for middle school users, I think providing users with the ability to easily and mistakenly create recursive calls in their first minutes using Alice outside the tutorial has no up side. Recursion is often a concept that even students who are several weeks (as opposed to a few minutes) into learning to program have trouble mastering.

Chapter 4 Enabling Storytelling

4.1 Introduction

Basic animations like move, turn, and resize are not sufficient to enable girls to create the kinds of animated stories they typically envision, both because it is difficult to create human-like motions by moving and turning individual joints (users need to turn a character's hips, knees, and ankles individually to make it walk) and because girls are limited to communicating their stories through motion alone. Based on analyzing the storyboards girls created and observing the difficulties they encountered in building stories in Alice, I have added higher-level animations and multiple scene support to Storytelling Alice. This chapter summarizes my findings about the technical supports necessary to enable girls to create animated stories and describes how I have implemented these supports within the storytelling version of Alice.

4.2 Problems with Generic Alice Animations

In early user testing, I observed a mismatch between the kinds of basic animations that Generic Alice supplies (move, turn, resize, etc) and the kinds of stories that girls wanted to create, stories in which characters walk around and interact with each other. While middle school girls appeared to readily understand what the Generic Alice animations do, many found the process of combining basic animations to create more complex behaviors like sitting or hugging too difficult. Further, most found the sheer number of lines of code needed to realize their stories daunting and quickly began to scale back their goals. Often

their completed animations bore little or no resemblance to the stories they described as they began working. The kinds of problems that I observed users having in creating their stories suggested properties for Storytelling Alice.

4.2.1 Supply animations that better match common actions

In Generic Alice, all animations more complex than sliding through space or rotating around an axis involve directly controlling the character's body parts. To create a basic walk animation for a character, a user must combine at least eight different turn animations to animate the characters' upper and lower legs, a move animation to move the character forward while his legs are animating, and a loop to enable him to take more than one step. There are simply too many steps involved in creating animations like walk for a beginning user. With a cast of several characters, the task of teaching them to walk alone becomes daunting. And walking is unlikely to be the only animation that girls will need to create for their stories. While girls may be willing to put special effort into an animation that is particularly important within their story, they need to be able to get the basics of their story in place without Herculean effort. In designing Storytelling Alice, it was important to determine what actions are commonly used in stories and create animations to match those.

4.2.2 Reduce the need for trial and error

Building stories using the Generic Alice animations requires a lot of number tuning via trial and error. Even seemingly simple actions like moving one character into a position where they can talk to another character (e.g. the two characters should be fairly close to each other and facing one another) can involve a lot of number tweaking: the user might start by having the character move forward 3 meters and discover that 3 is too big. After several tries, they might find that the 1.7 seems to give them a reasonable result. Storytelling Alice provides animations that allow users to tell their characters where to move relative to a target (e.g. in front of a sofa or to the right side of a chair), eliminating much of the need for number tuning when moving characters in a scene.

The need to tweak numbers becomes even more pronounced when users try to create complex actions like having a character touch an object. Inverse Kinematics is a

commonly technique for calculating an appropriate set of joint rotation angles to get an end-effector (e.g. a hand) to touch a target (e.g. a doorknob). However, Generic Alice does not include inverse-kinematics animations, so users must find appropriate joint rotation angles through trial and error. This can be quite difficult in practice; moving a character's arms such that he or she appears to be touching an object using Generic Alice can involve tuning the values for rotations around three axes for both the upper and lower arms. Further, the hand-tuned rotations are extremely brittle. If the user needs to move either the object or the character attempting to touch the object, the user will have to find a new set of joint rotation angles using trial and error. Storytelling Alice includes two animations based on simple, two-joint inverse kinematics to enable users to have characters touch targets and keep touching those targets as the targets move.

4.2.3 Beginning users should not need to understand graphics concepts like insertion points

The goal of my work is to motivate girls to learn a little bit about computer programming through creating animated stories. Consequently, I would like to avoid having users spend time learning about the internal workings of Alice or 3D graphics systems in order to understand the behavior of an animation they are using. For example, Generic Alice has animations like **`move to`** and **`move toward`** that depend on the origin of an object, a value that is assigned by the artist who created that object. If a user creates a move to animation in which the moving character's origin is at waist level and the target object's origin is at ground level, the move to animation causes the moving character to sink into the ground. Users do not need to learn about the origins of 3D objects in order to become competent programmers. By focusing on humanoids and animals and performing most calculations based on bounding boxes, I have hidden many of the inner workings of the graphics engine; users rarely need to understand what an object's origin is or how a pivot point can influence the rotation of an object.

4.3 Determining Users' Needs for Storytelling

Observing girls trying to create stories using the Generic Alice animations can provide insight into why animations like move and turn are not sufficient for storytelling, but they provide little insight into what supports Storytelling Alice should provide to enable girls

to create stories. To capture girls' intentions, I asked them to create paper-based storyboards of their story ideas. I then used those storyboards in combination with classroom observations, logs of the actions girls took within Alice, and the Alice worlds they created to identify problems. In between sessions, I modified Storytelling Alice to address the most serious problems I witnessed.

4.3.1 Analyzing Storyboards

My participants created their storyboards using a 3-step process. First, girls wrote a paragraph length summary of the story they intended to create. Then they broke the story into scenes and described the setting, action, and purpose for each scene. Finally they drew and annotated 6-9 frames for each scene in their story. The process girls used in creating their storyboards is more fully described in Chapter 4: User Testing.

In analyzing the storyboards girls created, I looked at 3 sources of information:

1) Changes that occur between one frame of the storyboard and the next.
In one frame of the storyboard a character might be standing next to door and in the following frame, he is sitting on a couch. In between those two frames, the character must have walked over to the couch and sat down.

2) Actions described in the text under a storyboard frame.
For each frame of their storyboards, girls must describe the action taking place in that frame. Sometimes the descriptions underneath a storyboard frame reinforce action we can extract from the pictures themselves. In the case of the character walking to the couch, an annotation stating that the character walks over to the couch and sits down does not provide any new information. Often, the frame annotations describe actions that either are not pictured or are unclear in the storyboard frame. For example, an annotation for a picture showing two characters might indicate that one of the characters waves her arm and says hello.

3) Actions mentioned in the scene description

Occasionally, a user will mention an action in the scene description that does not appear in either the frame drawings or their annotations. Because these were part of the user's vision of their story at some point, I have included them.

4.3.2 Insights from Storyboards

I used storyboards as a qualitative data source. By reading through the storyboards, one gets a fairly clear sense of the kinds of things that girls expect to be able to do. In particular, I read through the storyboards looking for things that girls expected to be able to do that I knew to be difficult to implement within Alice. In addressing problems that I identified, I prioritized common actions and actions that played a critical role in stories above other actions.

4.3.2.1 Example Storyboard Data

To provide the reader with a feel for the kinds of information that I was able to extract from the storyboards, I describe the information extracted from the storyboards created by one troop of six Girl Scouts. For each storyboard, I wrote down all of the actions that appear in storyboard frames, frame descriptions or scene descriptions. I then classified these actions into six categories: changes in body position, camera motions, dialog, actions intended to convey emotional expression, locomotion, and object manipulation (see table below). The six storyboards contained a total of 280 actions.

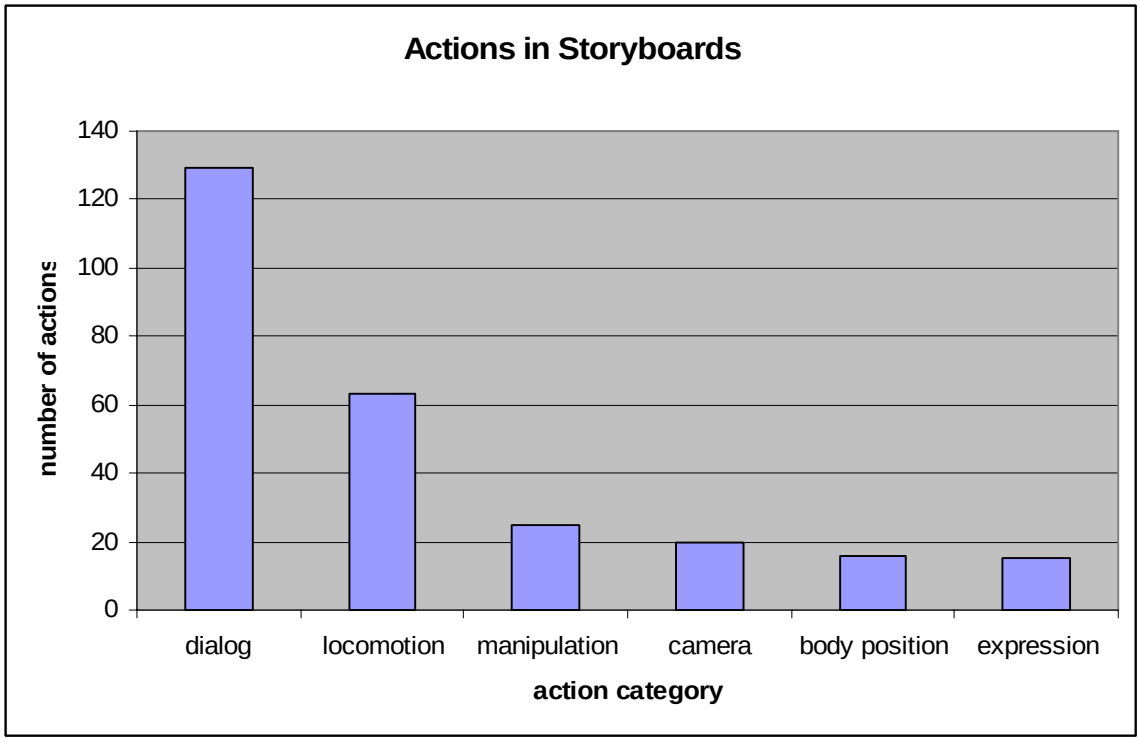


Figure 4.1: Counts of storyboard actions by category.

4.3.2.2 Dialog

Almost half of the actions in the storyboards were communication based: (i.e. characters speaking or thinking). Users' worlds make heavy use of the ability for characters to speak and think to communicate character's intentions, histories, feelings, etc.

4.3.2.3 Locomotion

Locomotion actions are actions in which the characters move to a new location within the scene. Characters most commonly walked up to other characters or targets within the scene, entered the scene or left the scene. Again, the storyboard plans match fairly well with what girls actually create in their Alice worlds.

4.3.2.4 Manipulation

Manipulation animations involve one character interacting either with an object or another character. Examples from the storyboards include a character petting a dog, dribbling a basketball, pointing at another character, or hugging another character. Often manipulation actions involve having a character touch an object and possibly maintain contact with that object as it moves through space. For example, in animating a character

opening a door, a user would probably want the character to touch the doorknob and then maintain contact with the doorknob as the door opens

4.3.2.5 Camera

Camera actions involve moving the camera to get views of particular characters or the whole scene. In the storyboards for this group, girls wanted to move the camera to get a good view of characters, to show the outside of a building the characters would be entering, and to move to difference settings (e.g. the city, inside the living room, etc).

4.3.2.6 Body Posture

Body posture actions involve a character changing the orientations of their limbs without moving a significant distance. In these storyboards, body posture actions include characters sitting or lying down, standing up, or turning to look at another character. However this category might also include a character performing a ballet pose.

4.3.2.7 Expression

Expression actions are largely intended to express a particular emotion such as anger or sadness. Based on the storyboards, the expression actions are united more by the fact that they are expressing emotions than by the means users chose to express the emotion. Examples include characters crying with cartoon-style tears, pouting, and stomping out of the room.

4.4 Requirements for Storytelling Alice

Through analyzing the storyboards my users created and observing users trying to realize their stories as animated movies, I have learned what primitives are necessary to allow girls to tell the stories that they envision and what tradeoffs are involved in adding these primitives into systems. I present my lessons learned to inform the design of future systems for creating animated stories.

4.4.1 First and foremost, characters need to be able to express themselves

Simple animations that allow characters to speak and think simple text (in our case through cartoon-style speech and thought "bubbles") can go a surprisingly long way towards enabling storytelling, both in terms of helping to communicate the story and

allowing characters to express emotions. Speech and thought animations seem like an obvious addition to any animation system. Yet, versions of the Alice system existed for more than five years without a way for characters to speak or think (Conway 1997; Conway, Audia et al. 2000).

Figure 4.2: An example "say" animation in Storytelling Alice.

4.4.2 Users Animate People and Characters

It is extremely rare for users to want to animate chairs and tables, but extremely common for users to want to animate people and other characters. Further, users have expectations for how objects should be animated based on their appearance. For example, when users added a move animation for a person, users expected the animated human figure to walk, not slide, forward. And these expectations went beyond the method by which a particular object moves. Users expected people to be able to perform the kinds of basic actions that most people can do including standing, sitting, and touching objects.

In Generic Alice, all 3D objects can perform the same animations and, from a programming perspective, are of the same type. As I began to add animations specifically for people and characters to Storytelling Alice, it no longer made sense to have only one type of object. In Storytelling Alice, there are 3 types of objects: "things," "humanoids", and "characters". Things are objects like chairs and tables that users would not ordinarily want to animate. Things can perform simple animations like move and turn. Humanoids are all bipedal characters that can walk, talk, change their body posture, and perform a

variety of gestures. Characters are animals and monsters that do not have a bipedal body structure. Characters can move by sliding around the scene, talk, and perform some gestures like looking at objects in their 3D world.

Table 4.1: Object types and their animations

Object Type	Animations
Humanoid	Say(message string) Think (message string) Play sound (sound) Walk to (object or character) Walk offscreen Walk (distance) Move (direction, distance) Sit on (object or character) Lie down (object or character) Kneel Fall down Stand up Straighten up Look at (object or character) Look (direction) Turn to face (object or character) Turn away from (object or character) Turn (direction, amount) Touch (character or object) Keep touching (character or object)
Character	Say(message string) Think (message string) Play sound (sound) Move to (distance, direction to, object or character) Move (direction, distance) Look at (object or character) Look (direction) Stand up Straighten up Turn to face (object or character) Turn away from (object or character) Turn (direction, amount) Roll (direction, amount)
Object	Turn (direction, amount) Roll (direction, amount) Straighten up Move (direction, amount) Resize (amount)

4.4.3 Most stories require multiple scenes

For the purposes of this discussion, I use the term scene in the way that a play might: a scene takes place in one setting over a continuous block of time. In our first user testing

session, I gave users a handout on storyboarding and asked them to draw storyboards on paper. 9 of 11 storyboards clearly revealed that users expected to be able to create multiple scenes. This is perhaps less surprising when you consider that even movies and plays that take place in a single setting, which are often considered unique, show action in non-continuous blocks of time and therefore use multiple scenes. While having users create free-form storyboards made the need for multiple scene support clear, the storyboards were not detailed enough to inform the design of higher-level animations. After some experimentation with different methods for creating storyboards, I settled on the 3-step process described in Chapter 1Formative User Testing.

While it is technically possible to create the appearance of having multiple scenes within Generic Alice, there is no explicit support for it, and it is outside the scope of something we could reasonably expect a beginning Alice user to master. Users who tried to create scenes in Generic Alice were largely unsuccessful. One pair of girls commented to the observer that whenever they needed to start a new scene, they went ahead and called the observer because it was "just too confusing" to try themselves. And, of the 9 stories that required scenes, only 3 were actually able to implement them, even with extensive help.

4.4.4 Scenes can ground and motivate the use of programming subroutines.

Even though the process of creating and maintaining scenes was confusing for users in Generic Alice, I found that scenes provide a wonderful way to ground and motivate the concept of programming subroutines (called "methods" in some languages). Girls seemed to find the notion of separating the action for scene one from the action for scene two and being able to call each scene's subroutine when needed to be fairly natural. In adding scene support to Storytelling Alice, it was important to us to be able to continue to use scenes to introduce girls to the concept of subroutines.

4.4.5 Basic changes in posture go a long way

Despite the amazing number of positions the human body is capable of assuming, I found that there were really only three that showed up regularly in stories: sitting, standing, and

lying down. In Storytelling Alice, these animations are: person sit on (target), person stand up, and person lie down on (target).

However, I did add animations for two additional body position animations: kneel and fall down. While kneel and fall down are not used with the same frequency as sit, stand up, and lie down, I found that they both play significant roles in the kinds of stories that middle school aged girls tend to tell. I had two groups in a row of Girl Scouts where marriage proposals played a significant role in most of the stories. Kneel was important not because it was a commonly occurring action, but because it had great significance in their stories. Users also used kneel for tasks like petting dogs and picking up items on the ground. Fall down is used in stories either as a humiliating moment for a character or as a way to indicate that someone has gotten hurt. Both uses tend to come at important points in girls' stories.

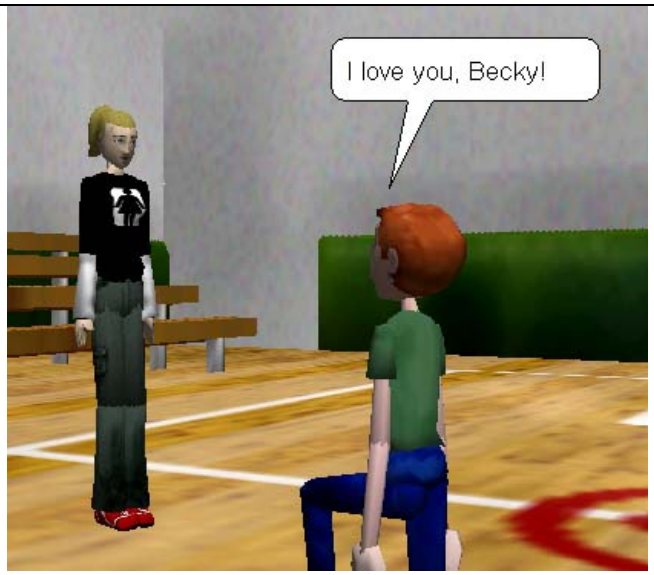


Figure 4.3: Although kneel is not as commonly used as sit on, stand up, and lie down, I added it to Storytelling Alice because it played an important role in many of the love stories middle school girls envisioned creating.

4.4.6 For the most part, locomotion is targeted

When characters move, mostly they move to a position relative to some other object or character in the world. For example, a person might walk over to a sofa and sit down or walk to another person to start a conversation. Characters also frequently need to leave a scene. Moving a specific distance, one of the most commonly used animations in Generic Alice (Conway 1997; Conway, Audia et al. 2000), is useful largely as a band-aid for

situations in which there is not an easy way for a user to describe where they want a character to go relative to some target.

In our storytelling version of Alice, there are three kinds of locomotion animations: person walk to (target), person walk off screen, and person walk (distance).

4.4.7 Many gestures and special-purpose animations are targeted touching

While there are many gestures that occur freely in space, I found that about half of the special-purpose animations that girls wanted to create can be captured by a character touching a target with his or her hand (or foot) and sometimes following that target as it moves through space. For example, a user could animate one character pushing another by having the first character touch the second and continue touching him as he falls down. As another example, a user could animate dribbling a basketball by touching the top of it and continuing to touch it as it moves up and down. The ability to have a character touch something and continue touching it as it moves has a great deal of expressive power.

In Storytelling Alice, I added two animations: person touch (target) and person keep touching (target). Users can specify additional parameters to control which limb (right arm, left arm, right leg or left leg) they would like to touch the target with and which side of the target they would like to touch.

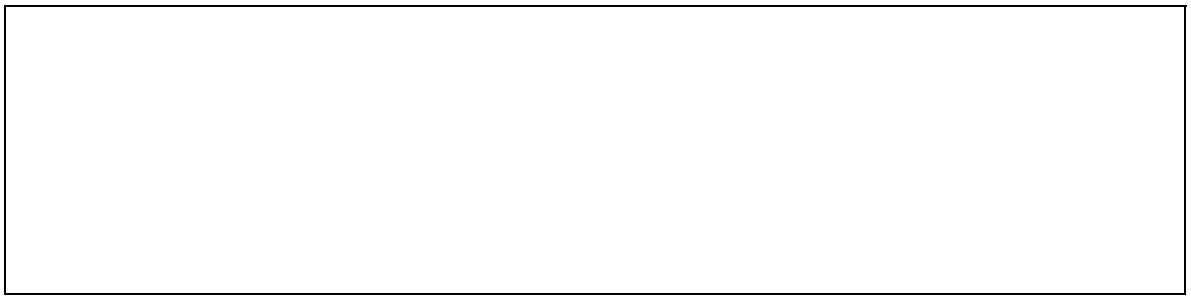


Figure 4.4: An example of a push animation created with the touch and keep touching animations (above) and a series of images showing the push animation in action (below).

4.4.8 Users need an easy way to get characters back to a normal position

In Generic Alice, users specify joint rotations numerically. For example, to have a character kick their leg, a user might use the animation "person.rightLeg turn backward 0.25 revolutions". One advantage of numerically-based animations is that they have obvious inverses; to return the character's leg to its normal position, the user turns the leg forward by 0.25 revolutions. When using animations like touch (target) and look at (target), it is harder for users to return characters to their default postures (i.e. standing straight with arms at sides. In Storytelling Alice, I added the "straighten up" animation which returns all the body parts to their normal positions. It is sometimes possible for straighten up to generate motions in which body parts pass through other body parts. Although users sometimes ask if there is some simple way to avoid unrealistic motions, few are willing to add extra animations to generate more realistic motion.

4.4.9 It is sometimes necessary to annotate 3D models with target information

Because most locomotion is targeted, it is important that users can reference the targets that they need. It is important to annotate scenery objects with target information so that users can easily direct their characters to perform actions like walking over to the painting on the wall. In the absence of appropriate targets, users often rely on trial and error to position and orient their character.

4.5 Changes to Alice

Based on user testing, I made two large changes to Alice: 1) I added scene support and 2) I added a set of high-level animations to enable girls to construct the stories they envision.

4.5.1 Scenes

In Storytelling Alice a scene includes a collection of 3D objects (scenery and characters) and a method (the action that takes place).

To create a new scene, Alice needs to support the user in:

1. finding a location within the 3D world for the new scene
2. organizing the characters and objects for the scene
3. creating a new method in which the user can specify the action for the scene
4. moving the camera when setting up or modifying scenes
5. moving the camera as part of their scene action

4.5.1.1 Finding a location for the new scene

In selecting a location for a new scene, it is important to avoid the possibility of viewing multiple scenes at once. In Storytelling Alice, I chose to accomplish this by stacking the scenes on top of one another, but spaced far enough apart (i.e. at twice the far clipping plane) that objects in one scene would not be visible from other scenes. Even if a user decided to point the camera straight up, they would not see any of the other scenes. Another strategy for implementing scenes might have been to show and hide 3D objects for each scene on demand. I chose to place scenes out of visual range to minimize the changes to the core system and ease the process of incorporating changes to Alice 2.0 into Storytelling Alice as both developed.

4.5.1.2 Helping the user to organize the characters and objects needed in the scene

Allowing users to create worlds with multiple scenes creates two problems: 1) users can easily get overwhelmed by the number of objects in each world 2) main characters often appear in multiple scenes requiring that users either have multiple copies of their main characters or be able to easily move characters from scene to scene.

To make the number of objects more manageable, Storytelling Alice automatically creates a new folder in the object tree for each scene. All the objects and characters for a

given scene are placed inside the folder. Objects added to the second and all subsequent scenes are named “Scene X <object name> ”.

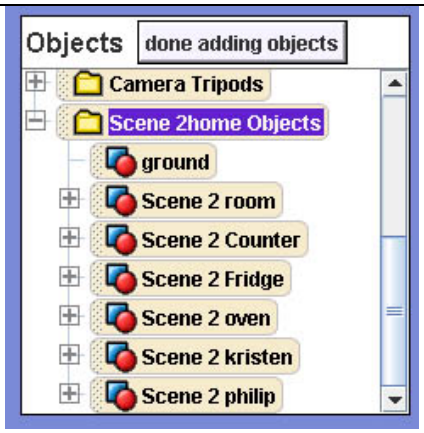


Figure 4.5: Objects for the home scene (scene 2) are added to a folder called “Scene 2 home objects.”

When users change which scene they are viewing in the world window, Storytelling Alice opens the folder corresponding to the new scene, scrolls the object tree such that the new scene is at the top of the viewable area, and closes all other scene folders. When users drag and drop commands into the method editor, Alice displays menus to allow users to select appropriate parameters for their method calls. In the pop-up menus, Storytelling Alice displays the objects and characters for the current scene in the top-level menu.

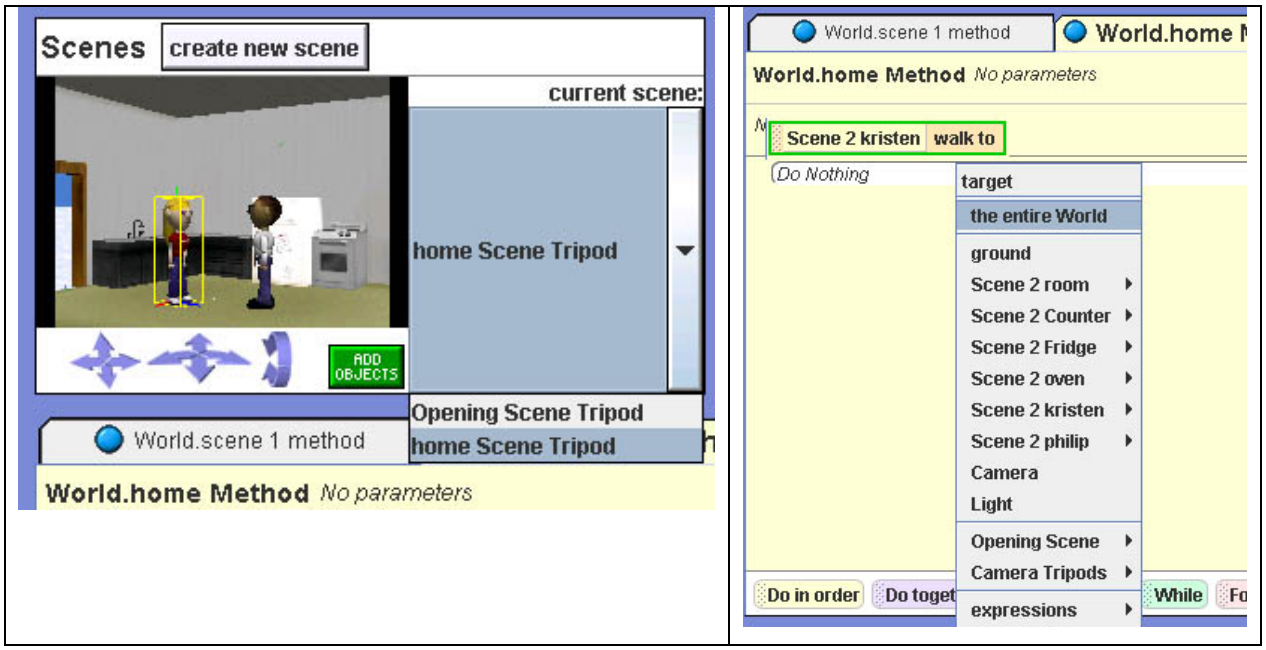


Figure 4.6: A drop down menu allows users to move from one scene to another (left). When the user drops "Scene 2 kristen.walk to" in the method editor, Storytelling Alice presents a list of the characters and objects in Scene 2 as potential targets (right).

Rather than moving characters that appear in multiple scenes from scene to scene, users simply add all of the objects and characters that are needed for a given scene from the gallery. While this does create the possibility for multiple copies of the same character in one Alice world (most commonly in different scenes), it simplifies the process of adding characters and laying out a scene. Each copy of an object has a different name (Scene X <object name>) to avoid naming collisions within Alice. However, having multiple copies of the same character creates a potential scope issue: users may expect methods and data they add to a character in one scene to apply to all instances of that character in their Alice world. In practice, I have not seen this issue arise in the first couple of hours interacting with Storytelling Alice.

4.5.1.3 Creating a new method in which the user can specify the action for the scene

When the user creates a new scene, storytelling Alice creates a new method named "<scene_name> method" where users can specify the action that occurs in that scene. All new worlds in Alice come with a "when the world starts" event that is set up to play "scene 1 method." When users create a new method, Storytelling Alice does not change

which method will play. This creates an opportunity for users to learn about the "when the world starts" event which they can use initially to play one scene at a time and later to call a method that will play their whole story.

4.5.1.4 Helping the user manage moving the camera when setting up or modifying scenes

Through user testing, I found that when users switch from one scene to another, they expect that the camera will return to its last position in the scene they switch to. To implement this behavior, Storytelling Alice automatically saves the camera position for each scene. When a user switches scenes in the authoring tool or hits the play button, Storytelling Alice updates the saved camera with the current position of the camera. Surprisingly, the ability to move the camera from scene to scene is sufficient for most users and most users do not create specialized camera shots. Those who do experiment with different camera shots tend to do so as a finishing touch on their story.

To enable users to save arbitrary positions and orientations for an object, Generic Alice provides a "dummy object," essentially a 3D model of an axis that holds the camera's place in the Alice world. In user testing, I found that most middle school girls found the concept of dummy objects difficult to understand. Describing dummy objects as camera tripods made the concept accessible to middle school users. In the physical world, tripods are frequently used to hold a camera in a specific location. The metaphor of a tripod is less appropriate for other 3D models, users rarely want to save the position and orientation of an object. By describing dummy objects as tripods within Storytelling Alice, the most common usage for dummy objects is readily understandable.

4.5.1.5 Helping the user manage moving the camera as part of their animation

The camera positions for each scene are stored as tripods that are automatically updated to reflect the most recent position of the camera in each scene. To allow users to easily move between scenes in their animated stories, I added an animation "move to scene tripod <scene name tripod>." A pop-up menu allows users to select the scene tripod corresponding to the next scene in their story.

4.6 High-Level Animations

In Generic Alice, all 3D objects are of the same type and perform the same set of animations. After analyzing the storyboards our users created, I modified Alice to create three types of objects: humanoids, non-humanoid characters, and objects (i.e. props and scenery). Each of these three types of objects can perform a different set of animations within Storytelling Alice.

4.6.1 Animating People

In creating stories, girls most frequently want to animate humanoid characters. In Storytelling Alice, humanoid characters can perform the following animations:

Say: <person> say <string>

The say animation displays a cartoon-style speech bubble containing a text message over the chosen character's head. Optional parameters enable users to set the background color of the speech bubble, the text color, font size, font, and duration of the animation.

Figure 4.7: An example "say" animation in Storytelling Alice.

Think: <character> think <string>

The think animation displays a cartoon-style thought bubble containing a text message over the chosen character's head. Optional parameters enable users to set the background color of the thought bubble, the text color, font size, font, and duration of the animation.

Figure 4.8: An example "think" animation in Storytelling Alice.

Play sound: <person> play sound <sound>

The play sound animation allows users to play a wav or mp3 sound. Alice comes with a library of 10 sounds. Additionally, users can choose to either record a new sound or select a sound they have stored on their hard drive. Optional parameters enable users set the volume level and duration of the sound.

Walk to: <person> walk to <target>

The walk to animation has the selected person walk to the specified target (another person, character, or object in the Alice world). By default, the selected person walks to a position 1 meter in front of and facing the target. Using optional parameters, users can change the end-distance between the person and the target and which side of the target (front, left, right, back, etc) the person approaches. Users can also change the style of the walk by changing the step size, the amount of vertical "bounce" in the person's step, and the extent to which the person swings their arms. To change how quickly a person reaches their target, users can change either the person's walking speed (e.g. steps per second) or the overall duration of the animation.

Walk animations naturally lend themselves to being controlled with speed (e.g. steps per second) rather than duration. If one asks a person to walk to two different targets, most people expect that the person will walk at a comfortable pace to both targets rather than choosing their speed based on how the distance to their target. By default, all the walk

animations have a set speed (1.5 steps per second) and calculate their duration. However, I found that users expected to be able to set the duration of walk animations; users control the pacing of other Alice animations by setting the duration. To accommodate this, all of the walk animations have both a speed and property and a duration property. Setting either the speed or the duration disables the other property. For example, if a user sets the duration for a walk animation, Alice disables the speed property and calculates an appropriate step speed using the duration.

Walk offscreen: <person> walk offscreen

The walk offscreen animation has the selected person turn to face stage right (as defined by the camera's current position) and walk just far enough that they are no longer visible to the camera. As with the walk to animation, users can change the style of the walk by modifying the step size, amount of bounce, and amount of arm swing the person uses. Users can set the number of steps per second or the duration to control how quickly the person walks. Users can also set the exit direction to control whether the person exits the scene to stage right or stage left.

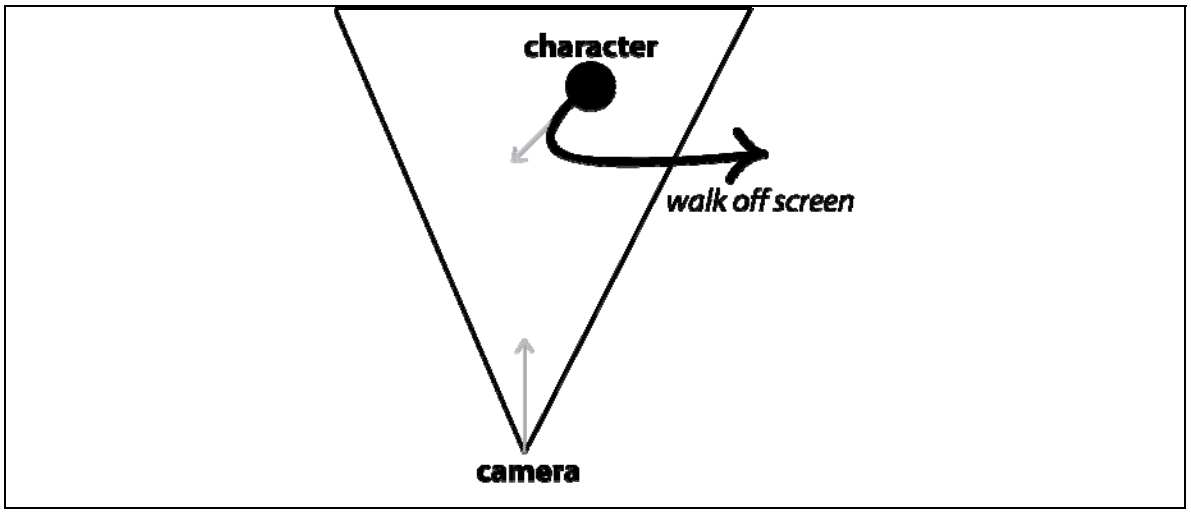


Figure 4.9: When a character walks offscreen, the character will turn so that its forward vector is parallel to the camera's right or left vector and walk forward enough distance to be out of view of the camera.

Walk: <person> walk <distance in meters>

The walk animation has the selected person walk forward a given number of meters. Users can use optional parameters to set step size, amount of bounce, amount of arm swing, steps per second, and duration.

Move: <person> move <direction> <distance>

The move animation slides the characters the specified distance in the selected direction without animating any of the person's body parts. By default, Alice uses object-centric directions. Each object is a coordinate system; if a user tells a person to move forward, the person will slide in his or her own forward direction. Optional parameters allow the user to specify the duration of the move animation and provide the ability to tell a person to move using someone else's coordinate system.

Sit on: <person> sit on <target>

Sit on animates the selected person to a sitting position on the front-center of the target's bounding box. If the target object is the ground, then the person will sit in place on the ground. Optional parameters allow the user to specify a different side of the object to sit on (e.g. the right side of the bed) and change the duration of the animation.

Lie down: <person> lie down on <target>

Lie down on animates the selected person to a lying position on the top center of the target's bounding box. If the target object is the ground, then as a special case the person will lie down in place on the ground rather than moving to the large ground plane's origin. Optional parameters allow the user to specify which direction the person's feet should be facing (e.g the right side of the couch) and change the duration of the animation.

Kneel: <person> kneel

Kneel animates the selected person into a kneeling position on the ground. By default, the selected person kneels on one knee but users can change an optional parameter to have the selected person kneel on both knees, as they might during a church service.

Fall down: <person> fall down

Fall down animates the selected person to a lying down position on the ground with their arms and legs in random positions. By default, fall down causes people to fall forward, but users can set an optional parameter to make people fall backward, left, or right.

Stand up: <person> stand up

Stand up animates the selected person to a standing position on the ground with all limbs in their normal positions (e.g. legs together and arms by the person's side). To avoid collisions with chairs and other objects a seated character might be on, stand up slides the character forward a little bit. An optional parameter allows users to have the character stand up in place.

Look at: <person> look at <target>

Look at animates a person's head to point at the target. If the target has a head, the selected person will look at the target's head.

Look: <person> look <direction>

Look animates a person's head to look in a particular direction: up, down, left, right, or forward (e.g. straight ahead).

Straighten up: <person> straighten up

Straighten up animates a person's body parts to their default positions (e.g. looking straight ahead with legs together and arms by his or her sides). Straighten up does not change the location of the person's body in the world. Users can apply straighten up to any part of the body. For example if the selected person was seated and looking at another character in the scene, the user could call person.torso straighten up to return the person's head to its normal position.

Turn to face: <person> turn to face <target>

Turn to face turns the selected person such that they are standing straight up and facing the target.

Turn away from: <person> turn away from <target>

Turn away from turns the selected person such that they are standing straight up and facing directly away from the target.

Turn: <person> turn <direction> <amount>

The turn animation rotates the selected person in a given direction (forward, backward, right or left) a given number of revolutions.

Touch: <person> touch <target>

The touch animation uses a very simple version of inverse kinematics to animate a person's right arm such that their hand is touching the front center of the target object's bounding box. If the person is not close enough to touch the object, the touch animation will move the arm such that it appears to be reaching towards the target object. Optional parameters allow the user to set which limb to touch the target with (right arm, left arm, right leg, or left leg) and which side of the object to touch (front, back, right, left, top, bottom). Because touch is based on bounding boxes, sometimes users need to be able to adjust the point that the hand or foot touches. To allow users to adjust the touch point, there is an offset which allows users to move the touch point along the line normal to the plane the person is touching (e.g. the front, back, right, left, top, or bottom of the target object).

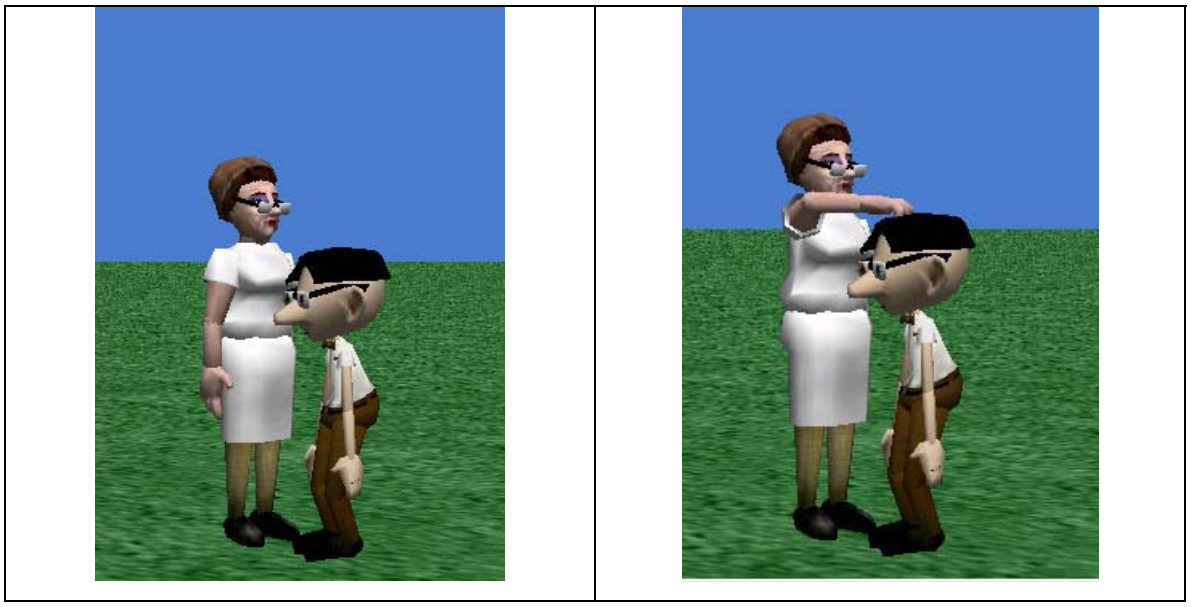


Figure 4.10: The image on the right shows the result of "LunchLady.touch Geoffrey side=up" from the starting position shown at the left.

Keep touching: <person> keep touching <target>

Keep touching is intended to allow users to create animations in which a person continues to touch an object as it moves through space. Keep touching positions the selected person's arm or leg in the same way as touch. However, where the touch animation calculates the target positions and orientations for the selected person's limb and animates the parts of the person's limb to those positions and orientations over the duration of the animation, the keep touching animation calculates the target positions and orientations each frame of the animation and sets the selected person's limbs to those calculated values. By default a person will keep touching their target for one second, but users can set a different duration for the animation using the more menu.

4.6.2 Animating Other Characters

Sometimes girls' stories include non-humanoid characters such as a dog or a fish. These characters perform a subset of the animations that humanoid characters do. Characters can perform the following animations:

Say: <character> say <string> [see Animating People]

Think: <character> think <string> [see Animating People]

Play sound: <character> play sound <sound> [see Animating People]

Move to: <character> move to <amount> <direction> <target>

Move to animates the selected character from its current position and orientation to the selected distance away from the selected side (front, back, right, left, top, or bottom) of the target's bounding box and facing target. Move to does not animate any of the character's body parts.

Move: <character> move <direction> <distance> [see Animating People]

Look at: <character> look at <target> [see Animating People]

Look: <character> look <direction> [see Animating People]

Stand up: <character> stand up [see Animating People]

Straighten up: <character> straighten up [see Animating People]

Turn to face: <character> turn to face <target> [see Animating People]

Turn away from: <character> turn away from <target> [see Animating People]

Turn: <character> turn <direction> <amount> [see Animating People]

Roll: <character> roll <direction> <amount>

Roll rotates the selected character around its forwards axis by a given number of revolutions in the selected direction (left or right).

4.6.3 Animating Objects

Objects perform a set of animations inspired by the commonly used animations in Generic Alice, which were tailored for moving objects in 3D space. These animations include:

Turn: <object> turn <direction> <amount> [see Animating People]

Roll: <object> roll <direction> <amount> [see Animating Characters]

Straighten up: <object> straighten up [see Animating People]

Move: <object> move <direction> <amount> [see Animating People]

Resize: <object> resize <amount>

The resize animation changes the size of the character by a multiplicative factor. For example, if the user specifies an amount of ½, the resize animation will animate the character such that each of its dimensions (height, width, and depth) is half as large as its original dimensions. Optional parameters allow the user to specify a single dimension to resize along and whether or not the object should resize like rubber (i.e. it should maintain a constant overall volume).

4.6.4 Animating Cameras

In both Generic Alice and Storytelling Alice, cameras are distinct from other 3D objects. However, in Generic Alice, cameras and 3D objects have an almost identical list of animations. In early user testing, one of the consistent messages that I heard from users was that cameras were too hard to control. In Storytelling Alice, I have added animations to enable users to create scene transitions, camera shots, titles and subtitles. Based on my user testing, scene transitions are frequently needed. Some users create specialized camera shots.

Move to scene tripod: <camera> move to scene tripod <scene tripod>
Move to scene tripod animates the camera to the selected scene tripod's position and orientation.

Get close up of: <camera> get close up of <target>
Get close up of moves the camera to a position far enough in front of the selected target that the entire target is in the camera's viewing area. It does not check for other objects that may be occluding the camera's view of the selected target. Users can change which side of the target object the camera is viewing (e.g. front, back, left, right, etc).

Get two shot of: <camera> get two shot of <target1> and <target2>
Get two shot of moves the camera to a position where both characters are in the camera's viewing area. By default the camera is positioned to the right of target1, but users can change which side of target1 the camera is positioned at.

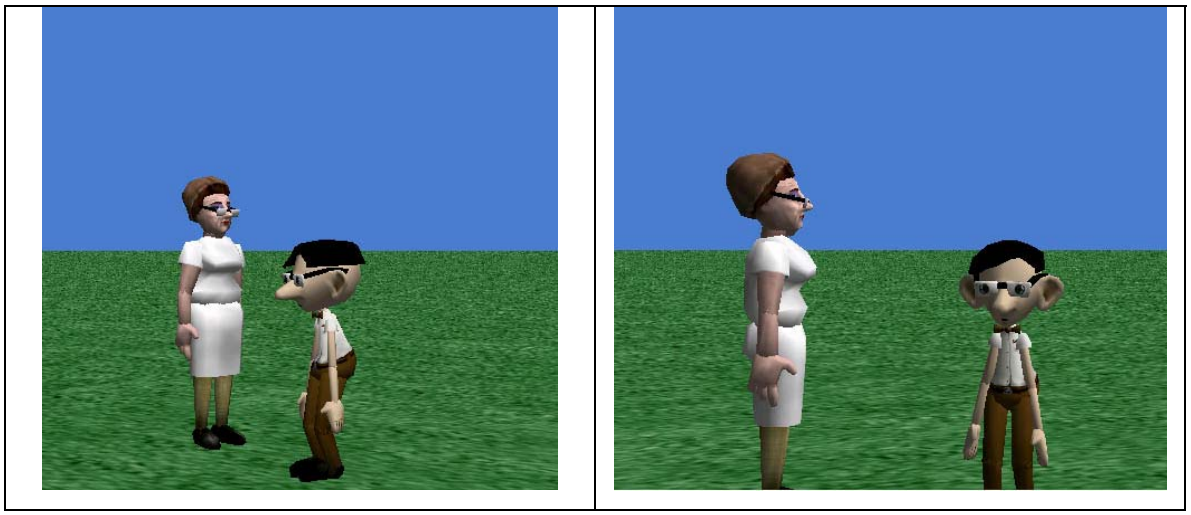


Figure 4.11: The image on the right shows the result of "Camera.get two shot of LunchLady and Geoffrey" from the starting point shown at the left.

Get character's view: <camera> get character's view <character>

Get character's view is designed to move the camera to a position showing what the selected character is looking at. It moves the camera to a position slightly in front of the selected character's head and an orientation matching that of the character's head.

Show subtitle: <camera> show subtitle <string>

Show subtitle displays the specified string at the bottom of the screen similar to movie subtitles. As with speech and thought bubbles, users can change the background color that the subtitle is displayed on and the text color of the message. By default, subtitles display for one second. They do not scroll.

Show title: <camera> show title <string>

Show title displays the specified string on a billboard placed in front of the camera. Users can change the background color and the text color. Like subtitles, titles are displayed for one second.

Fade to black: <camera> fade to black

Fade to black animates the lighting and background color of the scene to black.

Fade up from black: <camera> fade up from black

Fade up from black animates the lighting and background color of the scene from black to the standard lighting.

Point at: <camera> point at <target>

The point at animation orients the camera such that its forward axis is pointing towards the target.

Move to: <camera> move to <distance> <amount> <target> [see Animating Characters]

Move: <camera> move <distance> <amount> [see Animating People]

Turn to face: <camera> turn to face <target> [see Animating People]

Turn away from: <camera> turn away from <target> [see Animating People]

Turn: <camera> turn <direction> <amount> [see Animating People]

Stand up: <camera> stand up [see Animating People]

Roll: <camera> roll <direction> <amount> [see Animating Characters]

Chapter 5 Developing the Storytelling Gallery

5.1 Introduction

In my user testing sessions, one of the attributes that girls who had a positive experience with Alice tended to share was a vision for a story that they actively wanted to pursue. Further, the characters and scenery that girls added to their Alice worlds often had a substantial impact on their ability to find a story idea, their success in creating a program, and on their continuing interest in using Alice. The potential impact of girls' choices of 3D objects was illustrated by a pair of girls who came in to user test an early version of Alice. One of the two girls chose to add a dinosaur and a person to her world. She then proceeded to build a simple story in which the dinosaur scared the person and the person ran away in fear. Having accomplished that, she added a mouse from the gallery and continued her story by having the dinosaur be frightened of the mouse and run away. In this case, the dinosaur's potential to be a frightening character provided the inspiration for a simple story.

The other girl was drawn to a collection of amusement park models, in part because it was one of the only cohesive spaces available in the gallery at the time. She spent a long time carefully arranging the rides in her amusement park and then added a man into the park. She began by having the man ride the merry-go-round but quickly ran out of ideas she wanted to pursue.

Both of these stories are simple cases in which the stories were largely based on cues in the visual appearance of the models that girls chose from the gallery. These and other similar user tests illustrated the role of Alice's gallery of 3D characters and scenery in girls' success at finding of story ideas.

To find techniques for inspiring stories through the Alice gallery, I explored Story Kits. I define a Story Kit as a small, themed collection of characters (in which each character has a small set of animations that only that character can perform) and scenery, intended to be used as a starter-set for constructing a story. I chose to focus on Story Kits for two reasons:

First, girls in user tests were often attracted to coherent sets of objects within the gallery. For example, they frequently selected the models and characters from Egypt, Japan, and the Amusement Park, the only coherent sets in the original Alice gallery.

Secondly, Story Kits provide a low-cost way to experiment with different ideas. Making rapid, large-scale changes to the full Alice gallery of more than 350 models was not feasible. Story Kits provided the opportunity to identify promising approaches by quickly developing and testing smaller set of models.

5.2 Approach

My investigations with Story Kits took place within the context of a seminar for undergraduate students that I co-taught with Entertainment Technology Center Master's student Jessica Trybus. 13 undergraduates who had prior experience with Alice participated in the Story Kits seminar. The undergraduate students worked in teams of 3-4 students to create and test a series of Story Kits. To create a Story Kit, each team had to create 3D geometry and textures for all characters and scenery elements and animate their models in Alice. Teams had two weeks to create each Story Kit. Over the course of the semester, we created and tested a total of 16 Story Kits in four rounds with each round taking two weeks. At the end of each round, the Story Kit creation teams were shuffled so that the undergraduate students were working with a different team on each Story Kit

that they built. The approach of two-week long projects and shuffling teams for each project was inspired by the Building Virtual Worlds course (Pausch).

Throughout the semester, a group of 10 local children came to Carnegie Mellon once a week to build stories in Alice using Story Kits. The children ranged in age from 10 to 15, 7 were female, and 6 were African-American. 4 attended public or private school and 6 were home-schooled. The weekly sessions with the children were 1.5 hours long. During the first session, we introduced the children to Alice, concentrating on the features that we felt would be useful in creating stories. During the subsequent Friday sessions, we asked the children to work in pairs to create a story using one of our Story Kits. Occasionally, because of absences or disagreements, children created stories individually. The Story Kits work occurred fairly early in the process of developing Storytelling Alice, before I had concluded that asking users to work individually on stories was preferable to having them work in pairs.

While children created their stories in Alice, undergraduates were required to observe students using a Story Kit they were not involved in creating, and take notes about what the children did and said while creating their stories. These observations were used to guide subsequent rounds of design.

5.3 Design Process

To provide some insight into the process, I will briefly describe the Story Kits produced during each of the four rounds. Our goal in these four rounds was to explore the space of Story Kits and identify promising techniques for inspiring stories through the gallery.

5.3.1 Round 1

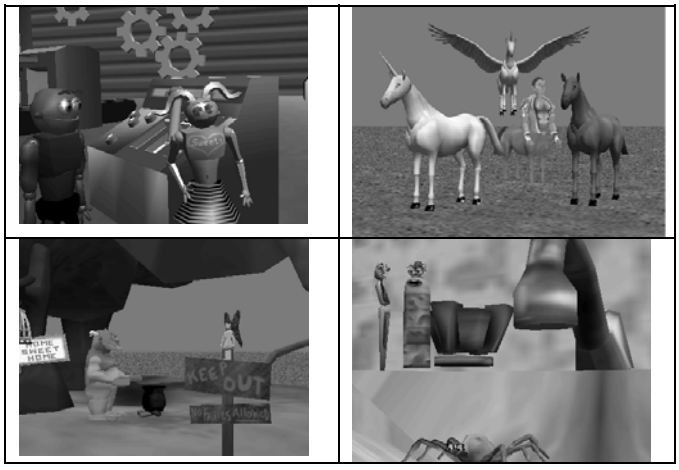


Figure 5.1: Clockwise from left: Robot StoryKit, Mythology StoryKit, Spider in the Sink, and Faeries StoryKit

5.3.1.1 Robots:

The Robots toolkit provided two potential storylines with overlapping characters: 1) a mad scientist story and 2) romantic relationship story. The characters, a mad scientist and four stereotypical teenage robots, two boys and two girls, were designed to have exaggerated and easily identifiable personalities. The setting for this kit was a factory with several machines that could turn on and off and break.

Character animations proved to be one of the strongest story motivators within the Robots Story Kit. Two of the three worlds were motivated by an animation: the "crazy go nuts" animation prompted a story that led up to one of the robots going crazy; the slap animation prompted our testers to explain through dialogue why one character had hit another.

5.3.1.2 Mythology:

The Mythology Story Kit intended to enable testers to use the rich space of Greek mythology as a basis and inspiration for their stories. Unfortunately, the designers of this kit were unable to finish all of the elements they intended to include in the Story Kit. As presented to our testers, the Mythology kit included a centaur, a horse, a Pegasus, and a unicorn. Our testers struggled in creating stories with this Story Kit, probably in large part due to its incompleteness.

5.3.1.3 Faeries:

The Faeries Story Kit used magic as a basis for their toolkit. This Story Kit contained an ogre, a talking tree, and two faeries: a boy and a girl. The setting is the ogre's home in a swamp, which is decorated with a sign that reads, "No faeries allowed." The ogre, the faeries, and the talking tree could all cast a variety of spells. The Faeries Story Kit demonstrated the use of environmental cues as a way to inspire stories. In this case, a "No faeries allowed" sign helped our testers to understand that the faeries and the ogre were not supposed to like each other. However, users were not completely successful in moving from the knowledge that the faeries and ogre disliked each other to a clear story idea. Two groups disliking each other is a very general conflict. We found that users tend to be more successful at creating stories based on concrete conflicts. For example, if the Faeries Story Kit had communicated the reasons behind the ogre's dislike of faeries, users could have used knowledge of those reasons in creating their story scenarios.

5.3.1.4 Spider in the Sink:

The Spider in the Sink Story Kit used an introductory animation in which a spider lowers herself into a sink and realizes that she is trapped to communicate the central conflict for a story. The Story Kit provided several anthropomorphized toiletries that could help the spider and animations that turned on the hot or cold water to drown the spider. Although the introductory animation introduced a clear goal (create a story in which the spider gets rescued), none of the pairs using this Story Kit chose to create a rescue story.

5.3.2 Round 2

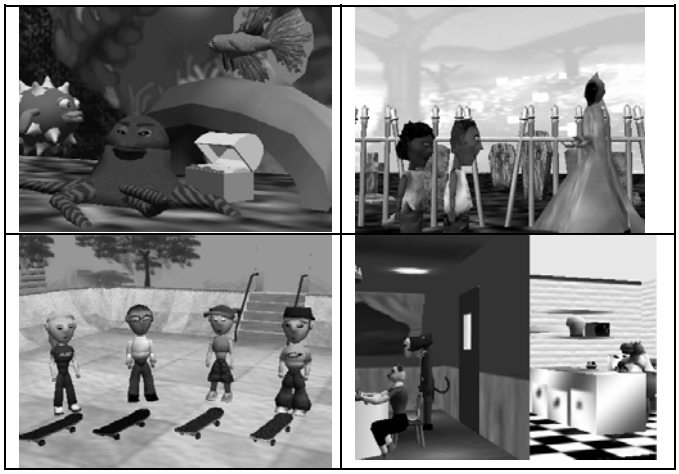


Figure 5.2: Clockwise from left: Aquarium Story Kit, Graveyard Story Kit, Restaurant Story Kit, and Skate Park Story Kit

5.3.2.1 Aquarium

Because the exaggerated characters in the Robots kit seemed to work so well, the designers of this kit decided to see whether creating a set of characters with exaggerated personalities would be as successful without considering potential storylines. The Aquarium Story Kit included six anthropomorphized sea creatures, an aquarium and several objects commonly found in aquariums such as a cave and a treasure chest. Even though the treasure chest was included merely to reinforce the aquarium environment, both pairs of testers focused a fair amount of their attention on the treasure chest because it helped to suggest a potential story line: one character attempting to guard or steal another's treasure. Neither of the pairs seemed to focus on personalities of the sea creatures.

5.3.2.2 Graveyard

In the previous round, we noticed that animations requiring explanation within the story can be pivotal in helping middle school students to find a story idea. The Graveyard Story Kit includes a young girl, a young boy, a ghost and a black cat in a graveyard setting. The girl and boy both have animations that represent extreme emotions. For example, the boy can cry and wail and both can pop their eyes out in fear. The pair that tested this Story Kit seemed to particularly enjoy the crying and screaming sounds associated with the boy's animations.

5.3.2.3 Skate Park

In the previous round, tensions between boys and girls seemed to be a popular theme in the stories created with the Robots. This Story Kit provides the support for similar boy-girl tension stories in a more familiar setting. The Story Kit provides four teenage skateboarders, two girls and two boys and a schoolyard setting. Knowing that sophisticated skateboarding tricks can be difficult to animate in Alice, the designers of this kit created an assortment of skateboarding trick methods that users could call. In addition to performing skateboard tricks, the characters can hug, kiss, shove, and punch each other.

The story created with the Skate Park Story Kit centered around three of the children ganging up on the fourth and punching him, for no discernable reason. The pair who created the story (both boys) seemed to particularly enjoy the cartoon-style punch animation.

5.3.2.4 Restaurant

Some of the most successful worlds from Round 1 included a lot of dialogue. In this Story Kit, the designers wanted to create a setting that encouraged the use of dialogue. The Restaurant Story Kit includes two restaurant patrons (a dog and a cat) and restaurant staff members (a panther waiter and a hippo cook) in a two-room dollhouse style restaurant. All the characters could perform "Rockout" animations. The "Rockout" animation seems like an odd choice for a restaurant-themed Story Kit. However, in previous rounds, we had observed that animations that were unexpected could often help to spark stories. "Rockout" was an attempt to further explore the space of unexpected animations and determine what kinds of unexpected actions can be successful story motivators.

All three pairs that used this Story Kit initially focused on a single animation: one pair focused on the "choke" animation; one pair on the "Rockout" animation that causes the animals to start playing musical instruments; the final pair focused on the "spill" animation. The users who focused on the "Rockout" animation had difficulty finding a sequence of actions that motivated the characters to begin playing musical instruments.

The most successful pair used the spill animation. The spill animation caused the object being spilled to move (by flying) to the hippo cook's hand and then the hippo spilled the object on the floor. The pair found the beginning of the animation in which the object to be spilled flies to the hippo's hand amusing and explained this strange behavior by having the hippo complain about having a strange magnetic force in his hands.

5.3.3 Round 3

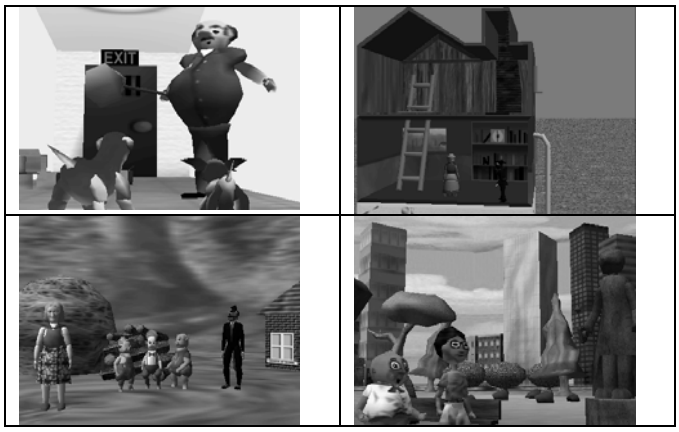


Figure 5.3: Clockwise from left: Kennel Story Kit, Jewel Thief Story Kit, Mosquito Man Story Kit, and Mixed Fairy Tales Story Kit

5.3.3.1 Dog Kennel

The designers of this kit wanted to create a kit that provided characters with exaggerated personalities and obvious character motivations. The Dog Kennel Story Kit contains several dogs, each with distinct personalities, and a dogcatcher. It is set in a kennel. The testers who worked with this kit seemed to understand the potential for conflict between the dogs and the dogcatcher. However, we did not see any evidence that they noticed the dogs' distinct personalities. One potential reason for this is that the animations that were designed to convey personality were hard to interpret unless the camera was close to the dog. However, the Story Kit did not provide camera animations to enable close-ups on particular dogs. Users' difficulties with moving the camera to reasonable positions in this kit and others provided the motivation to add camera primitives for common movie camera shots like close-ups and two-shots. The stories created with this kit seem to focus largely on having the dogs throw different objects at the dogcatcher.

5.3.3.2 Jewel Thief

The designers of the Jewel Thief Story Kit wanted to provide strong character motivations for each character within the Story Kit and enable our middle school users to more readily identify interesting animations. To accomplish this, each character in the Jewel Thief Story Kit has a fantasize animation, which displays a pictorial thought bubble intended to convey each character's dreams (for example, a grandmother character dreams about being a ninja), and a "show all methods" animation, which plays through all of the characters animations to help students identify interesting animations for use in their stories. The Jewel Thief Story Kit contains an old lady, a butler, a cook, a dog, and a ninja. The Story Kit is set in the old lady's mansion. The pairs using this Story Kit struggled with moving the camera and the characters to different floors within the mansion. In essence, each room in the house is a different setting. Users' difficulty in moving from one setting to another motivated the addition of scene support in Storytelling Alice. Despite their difficulties with the camera, one of the two pairs began to develop dialog that elaborated on the characters in their story, especially the old lady.

5.3.3.3 Mixed Fairy Tales

The Mixed Fairy Tales Story Kit provides characters from two common fairy tales: The Three Little Pigs and Little Red Riding Hood. The designers of this kit hoped that providing characters from two known fairy tales would spark interesting stories that combined different elements from the two tales. Three of the four pairs that worked with this kit clearly recognized the reference to The Three Little Pigs and Little Red Riding Hood. Two of the pairs attempted to create a variation of The Three Little Pigs story. However, once pairs discovered that Little Red Riding Hood had a "matrix kick" animation (a kick and coordinating camera motion that is similar in style to fight sequences in a movie called "The Matrix"), all 3 pairs switched their focus to creating something that used the "matrix kick".

5.3.3.4 Mosquito Man

The Story Kit contains two superheroes and a villain in a park setting. The designers of this kit intended for users to create a story in which the superheroes defeat the villain.

The Mosquito Man Story Kit incorporates a puzzle in the character animations; the superheroes must combine their powers in a certain way (through calling character-methods in a specific order) in order to disable the villain. One pair of middle school students tested this Story Kit. The pair was distracted early on by the discovery of a road in the park setting. Rather than focusing on attempting to create a story, they used the fact users can "drive" the camera around Alice scenes as a makeshift driving game.

5.3.4 Round 4

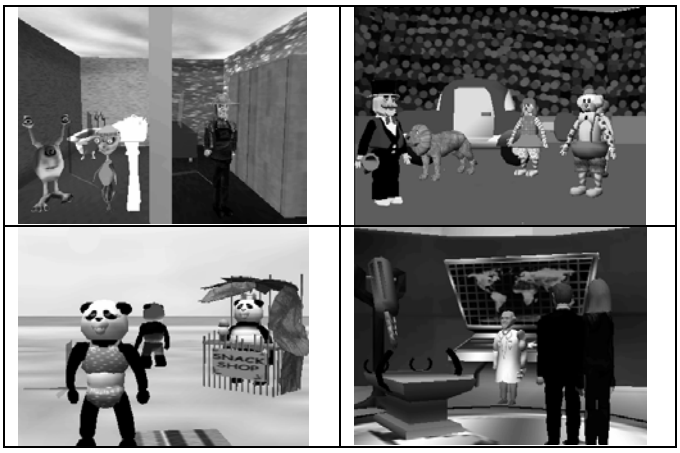


Figure 5.4: Clockwise from left: Aliens Story Kit, Wacky Circus Story Kit, Secret Agents Story Kit, and Panda Beach Party Story Kit

5.3.4.1 Aliens

The idea behind this Story Kit is that having two disjoint sets of characters that do not belong together can help generate story ideas. The Aliens Story Kit contains two Aliens and a farmer and is set in the farmer's house. Two pairs of Story Kit testers built stories with the Aliens Story Kit. One story was inspired by the Aliens' steal animations; the Aliens state that their mission is to steal everything and then return to their home planet. The second story was based around the farmer's household appliances deciding that they no longer wanted to work for him and disappearing. The pair creating this story seemed uninterested in the Aliens.

5.3.4.2 Wacky Circus

Over the course of the semester, we noticed that our middle school testers seem to strive for humor in their stories. This Story Kit represents an attempt to focus on humor. The Story Kit contains a variety of circus performers, a circus tent, and an audience that can

make a variety of sounds. One of the stories created with this kit plays directly into the theme the designers intended. In the story, a lion declares that he is hungry and proceeds to eat everything around him, including the clown car, the audience, and finally the circus tent. The pair creating the other story was not as successful. While they seemed to value the humor aspect of the kit and discussed having a crazy circus show, their progress was stalled by their difficulties moving the camera and the characters within the 3D environment.

5.3.4.3 Panda Beach Party

The Panda Beach Party Story Kit is based around the concept of providing recognizable characters and multiple story lines. The Story Kit provides four Pandas with animations that support romance and drowning stories.

The pair that used this Story Kit seemed to have trouble coming up with a beach-based story and chose to focus on an ice cream cone as the basis for a story. They discussed having one character steal an ice cream cone from another, but this did not happen in their final world.

5.3.4.4 Secret Agent

This Story Kit used a familiar genre to help middle school students generate stories. It provided two secret agents, an evil doctor, and the evil doctor's henchmen. The Story Kit is set in the evil doctor's observatory, which is equipped with a death laser and a piranha tank.

Of the Story Kits that attempted to help kids generate stories through the visual appearance of the characters and scenery, this was by far the most successful. Both pairs immediately recognized the genre and developed simple stories in which the agents defeat the evil doctor.

5.4 Lessons Learned

The Story Kits seminar experimented with a wide variety of strategies for inspiring stories including:

- Giving characters animations that require explanation (e.g. Harold the robot's "crazy go nuts")
- Environmental cues (e.g. the "no faeries allowed" sign)
- Providing an Alice world in which has the beginning of a story and introduces a conflict for the user to resolve (by creating the rest of the story)
- Embedding a puzzle in the story that users need to unravel (e.g. the Mosquito Man Story Kit in which users must use the superheroes' powers in a specific way to defeat the villain.)
- Giving characters animations that reveal a motivation for them (e.g the characters in the Jewel Thief Story Kit each had a fantasize animation that suggested a motive; the butler dreamt of being a king and the old lady wanted to be a ninja.)
- Using characters from familiar stories (e.g. a few Story Kits provided characters drawn from mythology and folk lore)
- Giving characters animations that suggest a personality (e.g. Sammy the Snail in the Aquarium Story Kit who was intended to be shy has animations for hiding in his shell and looking embarrassed)
- Providing character animations that create interaction with other characters (e.g. a kiss or slap animation)
- Creating Story Kits that bring together unexpected sets of characters (e.g. one of the Story Kits brought together a farming family and aliens; another paired characters from multiple fairy tales).

The Story Kits seminar was extremely helpful in identifying promising directions for creating 3D models and associated animations to help girls find story ideas. However, to solicit feedback on all of the Story Kits, we required Story Kit testers to use a particular Story Kit during each session and instructed testers not to add 3D content from other Story Kits, restrictions that users do not typically have in interacting with Alice. After the end of the Story Kits seminar, I removed the constraints on users' content choices and continued to test Story Kits.

5.4.1 The Need for Story Inspiration

Some middle school children seem to come into our Alice workshops with a fully formed idea for a story: some recount scenes from their lives and others use current events, often holidays, as a spring board. One girl used Valentine's Day as inspiration to write a story about an ogre waiting for years and years for the return of his true love, a fairy. In the end, the ogre's love does return. While some kids easily find story ideas, many of the kids who have participated in testing Storytelling Alice benefit from story-idea supports within the gallery. Through the Story Kits seminar and later user testing, I have identified the following techniques which have high potential for sparking story ideas within the Alice gallery framework.

5.4.2 Animations that Require Explanation

One of the most powerful sources of story inspiration is through animations that require explanation in the story. Typically these are animations that cause characters to perform behaviors that are more extreme than would be typically considered socially acceptable. For example, in the Robots Story Kit a robot character had an animation entitled "crazy go nuts." In using this Story Kit with a variety of users including the Story Kit testers, in classes taught to groups of home-schooled students, and in other informal testing, I have seen a wide variety of stories that culminate with the robot going crazy. Students have created stories that dealt with parental authority struggles, relationship issues, social status, academic difficulty, and more. For a summary of the kinds of stories that girls create in Storytelling Alice, see Chapter 10.

Initially, I attributed the success of animations like "crazy go nuts" to their unexpectedness. However, further user testing forced me to refine the explanation; "unexpected" animations like having an ogre spin his horns or animals randomly start to play instruments were not attractive to users. Although, people do not expect a character to randomly start playing a musical instrument, users had trouble finding interesting narratives to motivate that behavior. The power of animations like crazy go nuts is that they were expressing valid emotions in a more extreme way than would be considered socially acceptable in real life, in some ways similar to the over-the-top antics of the Looney Tunes characters created by Tex Avery (Wikipedia). Other examples that proved

useful in sparking story ideas included one character kissing another, a red riding hood character doing a Matrix-style kick, and dogs throwing bananas at a dogcatcher. These slightly extreme reactions to things are very appealing to children. It was not uncommon to hear children say things like "Little Red can matrix kick. I've got to use this." Initially, the attraction is often both the humor and the fact that the characters are behaving in ways that the children themselves cannot behave (e.g. "They can throw bananas?!"). Further, these kinds of actions ask an implicit question (e.g. what did the dogcatcher do to anger the dogs into throwing bananas at him) which helps children begin to piece together a narrative.

5.4.3 Character Roles

Characters with clear roles suggested either by their appearance (e.g. the knight in his armor who needs a quest or a cause to fight for) or their name (e.g. Butch the Guard Dog who needs something to guard) also proved helpful in inspiring stories. Often, users recognize characters from a particular genre; several users created stories with secret agent characters. Common themes in these stories included stopping a mad scientist or recovering a kidnapped character or stolen item. Other users selected characters who resembled characters in animated movies; Boris the Ogre has appeared in stories as Shrek or Shrek's cousin and the fish in the undersea category often appear in stories similar to Finding Nemo, although none of the users have referred to their fish characters as Nemo.

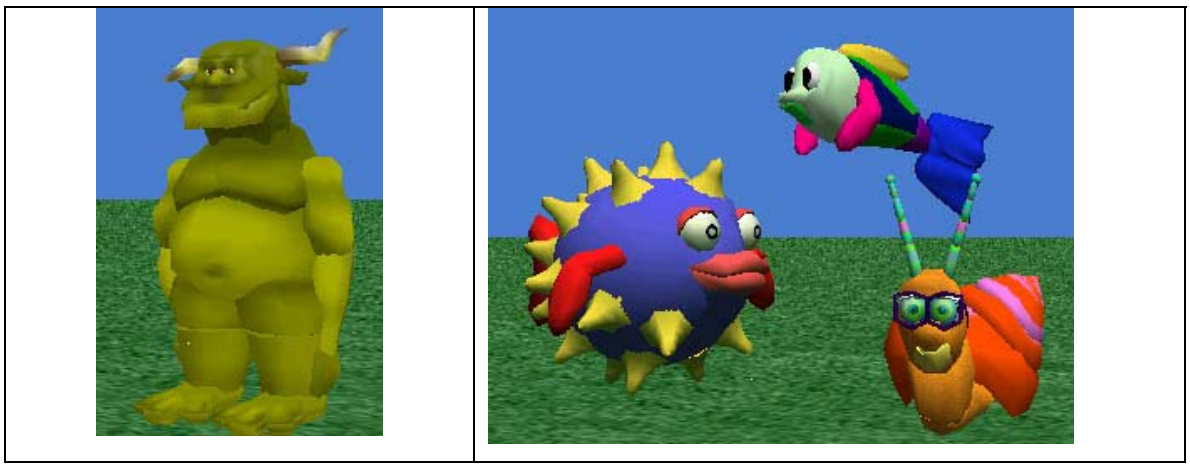


Figure 5.5: Boris the Ogre (left) sometimes appears as Shrek in stories. Fish (right) sometimes appear in stories similar to Finding Nemo.

5.5 Designing the new Story Gallery

The experience with Story Kits was valuable in exploring the space of ways that content (3D models and animations) can help children find story ideas, a key element in having a positive experience with Storytelling Alice. However, there were several ways in which Story Kits were not the right approach for use in the gallery.

5.5.1 Gallery Organization

In practice, girls rarely selected all of the characters from the same Story Kit unless they were specifically instructed to do so. Instead, they would often choose a setting from one kit and then assemble a cast of characters from several kits. Unfortunately, many of the Story Kit creators built the character-animations such that they only appeared correct when used with other characters from the same Story Kit. Consequently, using characters from multiple Story Kits rendered many of the characters' animations useless.

5.5.2 Story Beginnings

One strategy for helping users start creating a story is to provide users with the beginning of a story to complete. However, this removes the opportunity for users to select their own cast of characters, a process that many users clearly enjoy. Further, when I experimented with giving users the beginning of a story (through providing users with an Alice world with appropriate characters, a setting, and animation), it was rare for children to use the beginning of the story. Most users seemed uninterested in building on the provided story beginning, even when they were struggling to develop their own story ideas.

5.5.3 Environmental or Positional Cues

Several of the Story Kits included environmental cues (e.g. a "no faeries allowed" sign suggests a tension between the faeries and the ogres) or positional cues (e.g. a character coming into the world in a cage suggests that the character might need to escape) that suggest potential story lines. As with providing a story introduction, environmental and positional cues rely on being able to predict which characters users are going to combine with some accuracy.

5.5.4 Gallery Organization

Users clearly preferred to select characters from multiple Story Kits but were often frustrated by the process of searching for appropriate characters. When using the Story Kit based gallery, users lacked a good model for where they were likely to find appropriate characters for what they were trying to build. As a result, users often searched through nearly every Story Kit looking for a particular type of character (e.g. a girl or a dog). Often users' worlds included characters from several Story Kits.

In contrast to their tendency to select characters from multiple Story Kits, my user testers seemed to be attracted to the coherent spaces that Story Kits provided and would often use a whole setting from one of the Story Kits.

In developing the gallery for Storytelling Alice, I organized the content into characters and scenery. Users typically chose a cast and a setting as separate tasks, in either order. The characters were further broken into groups of similar characters. Most girls in my user tests tended to choose kids for the main characters in their stories. In response, I created a folder of "kids." The rest of the characters are organized into groups like "adults," "heroic," "scary," or "pets" (see Figure 5.9). These groupings more closely match the way that girls seem to select characters in their stories.

5.5.5 Character Animations

Another problem that arose from users selecting characters from multiple Story Kits is that often the characters' interaction-based animations (e.g. one character pushing another character) only perform correctly when used with other characters in the same Story Kit. This is an artifact of animating characters' limbs by rotating them a certain number of rotations. The correct rotation to reach one character's head might be very different from the correct rotation to reach another character's head. This problem can be fixed by writing animations using the new **`touch`** and **`keep touching`** animations in Storytelling Alice. At the time that the Story Kits were developed, **`touch`** and **`keep touching`** had not been added to the system.

In the Storytelling Alice gallery, each character comes with four character-specific methods. Based on the success of the explanation-requiring animations at inspiring stories, I tried to incorporate as many explanation-requiring animations as possible. Other character animations are designed to reinforce a character's likely role in a story. For example, the lunch lady has a scold method that reinforces her likely role as an authority figure.

5.5.6 Storytelling Gallery Content

There were themes in the characters and scenery that girls chose for use in their Alice programs. Many (but not all) of the stories were about human characters and had children as the story protagonists. Users also tended to choose familiar characters: often kids, parents, and teachers. This echoes Purple Moon's findings that girls wanted computer games that followed the lives of every-day characters (Laurel 2001). In addition to a collection of "kid" and "adult" characters, the Storytelling Alice gallery contains animals, fantasy characters, and characters from folklore.

In creating the gallery for Storytelling Alice, I drew from the Story Kits and the full Alice gallery more than 700 objects. While the Story Kits included some ordinary places and characters, including selected content from the Alice 2.0 gallery allowed me to expand the selection of characters and scenes.

5.6 Story Gallery

The Story Gallery is divided into two categories: scenes and characters.

5.6.1 Scenes

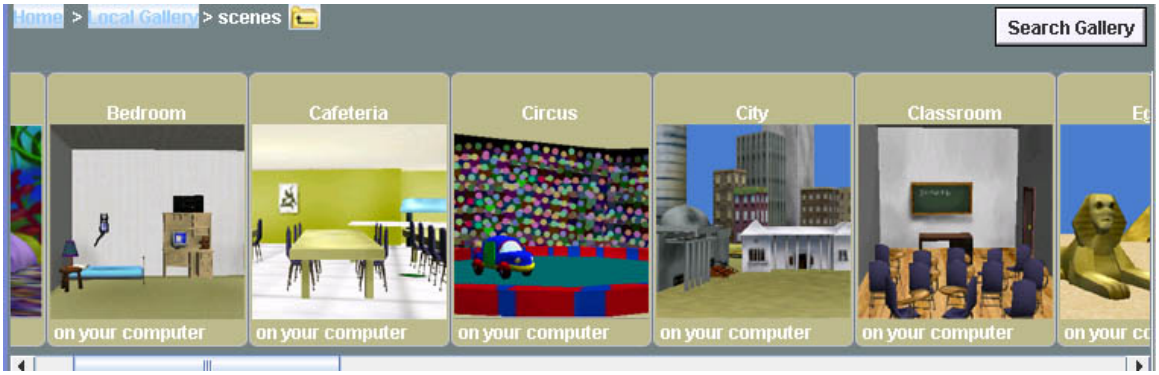


Figure 5.6: Some of the scenes available for use in Storytelling Alice.

There are a variety of scenes including: an aquarium, a bedroom, a circus, a city, a classroom, an Egypt scene, a factory, a forest, a garden, a graveyard, a gym, a hospital, a kennel, a kitchen, a lair, a living room, a neighborhood, a 3 little pigs village, a school hallway, a skate park, a stage, and a waterfall.

Figure 5.7: The 3D objects that users can compose to create a garden.

Each scene typically contains several objects that users can arrange in their Alice world to suit their own purposes. For example, the garden scene includes a "Garden" object which represents the garden grounds and walls. Users can add ferns and trees to decorate the garden. Through user testing, it became clear that the most important aspects of a scene are that it is 1) recognizable and 2) provides a sense of place. Few users wanted to create their own unique garden; instead, their sense of ownership of their program came through their stories. While it is tempting to provide small detail objects such as individual flowers to place in the garden to allow greater personalization, users often find

the process of placing small objects frustrating. The addition of detail objects seemed to provide few benefits to users. Consequently, each scene contains a small number of objects that allow users to personalize their scenes a little bit. These objects are sized to be easy to manipulate.

The vast majority of objects users can add to scenes do not have their own methods because most objects (e.g. furniture and buildings) rarely move. However, a few objects like the clown car and a laser-weapon that users may want to move have a small number of methods they can perform.

With the addition of the ability for characters to walk to a particular target, it was important to annotate some of the scenery objects with target information. Examples include doorways and details that are painted onto an object (and do not have 3D geometry associated with them) like a blackboard or a painting hanging on a wall.

5.6.2 Characters

Figure 5.8: Some of the categories of characters available in Storytelling Alice.

The characters are grouped in to several categories: adults, fantasy characters, funny characters, futuristic characters, heroic characters, kids, pets, scary characters, and undersea characters. The categories are based on the kinds of roles the characters can play in stories. Typically middle school children tend to have protagonists who are either their own age or a little bit older.

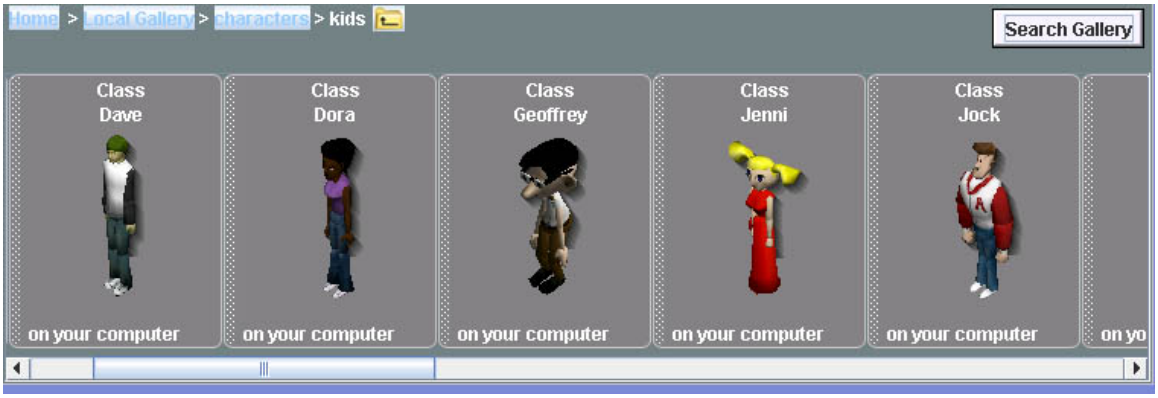


Figure 5.9: A selection of "kid" characters available in Storytelling Alice.

Each character in the Storytelling gallery comes with at least four animations that only that character can do. These animations have been designed both to support common ways that a particular character tends to be used and to provide somewhat extreme reactions that will help users find story ideas to pursue.

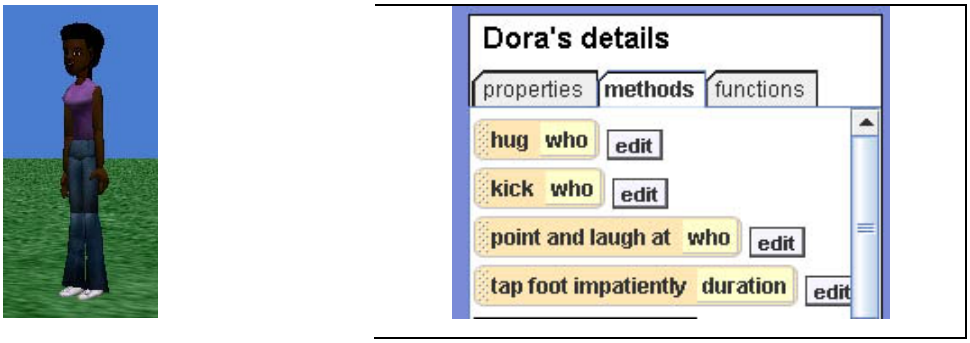


Figure 5.10: Dora, a character in the "kids" category and her character-specific methods.

Dora is an example of a character found in the "kids" category. Dora can hug, kick, or point and laugh at another character and tap her foot impatiently. All four of these methods require some explanation within the context of the story and can be used in a variety of different types of stories.

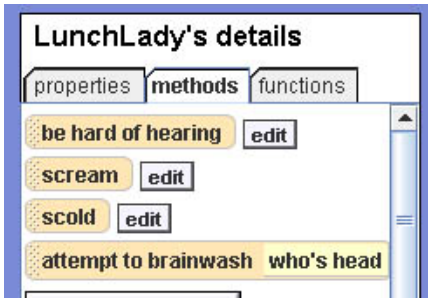


Figure 5.11: Lunchlady, a character on the "adults" category and her character-specific methods.

The Lunchlady is a character in the "adults" category who can scream, scold students, brainwash another character, or behave as though she is hard of hearing. Typically the lunch lady is used as an authority figure. To support this role, the lunch lady has two types of animations: scolding and being hard of hearing both reinforce the lunchlady's role as a slightly out-of-touch authority figure; screaming and brainwashing other characters are extreme responses to student misbehavior.

Figure 5.12: Character's default arm positions effect how they animate. Turn forward 0.25 would cause the girl's palm to face forward and the tin soldier to hold his arm out behind him.

While animating the characters in the Storytelling gallery, I also gave them a consistent default body position so that users who do animate a character's arm using the "turn" animation can do so more easily. In Generic Alice, there are two common default

positions for characters and, consequently, two behaviors for the animation model.arm.turn(forward, 0.25):

1) The character is in a t-position with their arms extended to each side. In this case, the arm rotates such that the back of the character's hand goes from facing up to facing forward (as in the picture on the left)

2) The character has their arms by their sides. In this case, the arm rotates from by the character's side to extend behind the character.

I found that users tend to have an easier time animating characters whose default position is with their arms at their sides. In this case, one can explain the behavior of the turn and roll animations using arm circles and flapping. Users were typically able to determine the correct animations for their desired motion in terms of forward and backward arm circles (i.e. turn forward or backward) and flapping (i.e. roll right or left). All the characters in the Storytelling gallery have default positions with their arms at their sides.

Chapter 6 Developing the Storytelling Tutorial

6.1 Introduction

Through user testing, I found that in addition to introducing girls to how Alice works, the tutorial must also convince girls that Alice can be used to build the kinds of stories they envision building. The stories girls envision creating tend to be more complex than the kinds of simple, mechanical examples typically chosen for tutorials. To enable girls to successfully complete story-based tutorials, I created an interaction technique called Stencils which draws a translucent blue screen which can catch mouse and keyboard events over the running Alice interface. For each step in the tutorial, Stencils cuts a hole over components with which the user needs to interact. Accompanying instructions are displayed on sticky-style notes drawn over the blue screen. Stencils is able to catch users' mistakes and ask users to redo the current step before users move to the next step in the tutorial, preventing users' mistakes from derailing their progress through the tutorial. In a study comparing the performance of users learning Alice with Stencils-based and paper-based tutorials, I found that users of the Stencils based tutorial made fewer mistakes and completed the tutorial more quickly.

6.2 Motivation

Many tutorial examples are deliberately chosen to demonstrate a particular feature or technique of a software system as simply as possible. While selecting simple examples can reduce the chance for crippling user errors during the tutorial, I found that simple examples may also leave users with the impression that the underlying software system is

boring or irrelevant. When I began user testing early versions of Alice 2.0 with girls, I found that many of the girls who successfully completed the tutorial were not interested in continuing to use Alice on their own. One girl summarized her disinterest in Alice by explaining that she thought that Alice was "a system for moving the bunny around" and wondered aloud why anyone would want to do that. While the early Alice 2.0 tutorial was successful in showing girls the mechanics of using Alice, it was unsuccessful in motivating girls to build their own programs in Alice. I found that it was necessary to create a tutorial that introduces the mechanics of using the system and shows girls examples of the kinds of stories they can create using Alice.

The typical story that a middle school girl envisions tends to have more lines of code and more objects than our original tutorial examples. Increasing the number of 3D objects and lines of code increases the number of user interface components and consequently, the potential for user error. Stencils enables users to successfully complete more complex tutorials by:

1. Helping users quickly figure out what to do in each step by drawing users' attention to components with which they need to interact.
2. Preventing users from making errors by preventing them from clicking on components unnecessary for the current step.

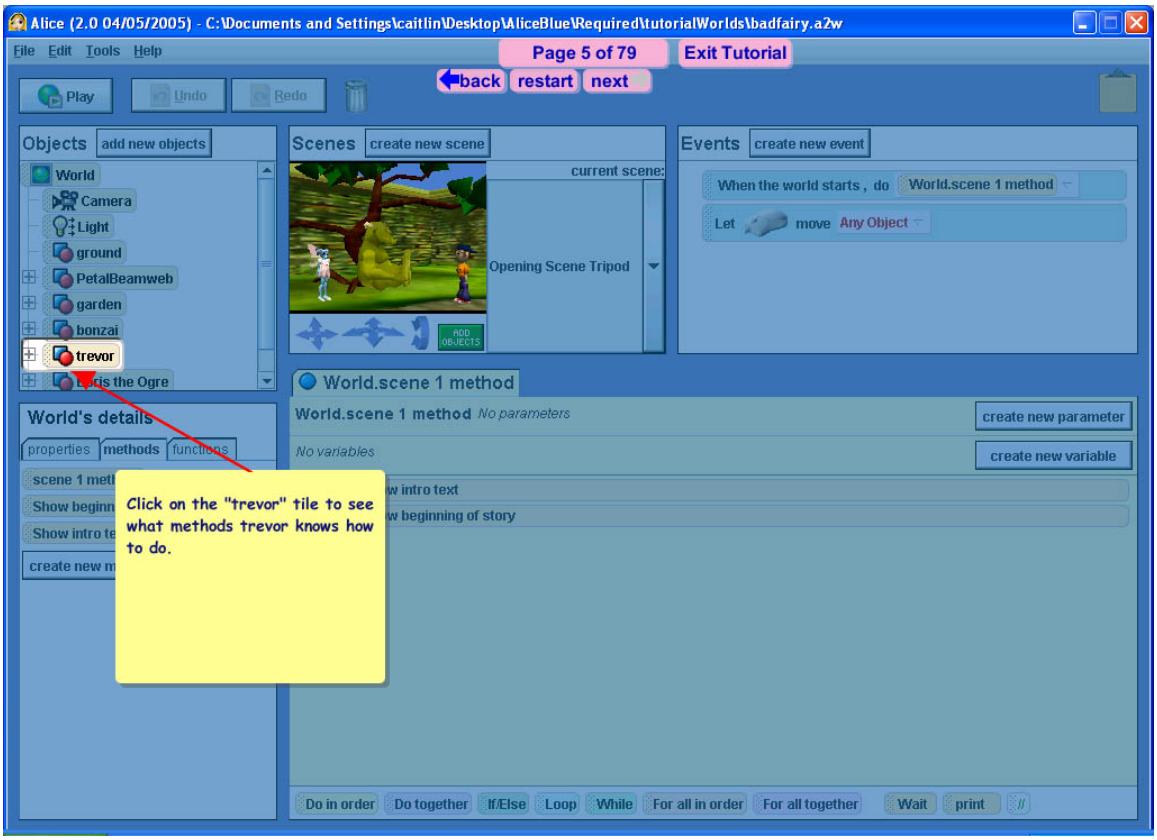


Figure 6.1: A screenshot of a Stencils-based tutorial in Alice with a hole over the interface component the user needs to interact with in the current step.

6.3 Related Work

There are three relevant areas of related work: the presentation of procedural instructions, learner-centered design, and transparent interfaces.

6.3.1 Presenting Procedural Instructions

Much of the research on how to present procedural instructions to users has been performed in the context of developing better help systems for software applications. Currently, most applications present procedural instructions for help systems in a separate window with supplementary pictures (Goodall 1991; Goodall 1992). However, researchers have found this method to be problematic for users (Knabe 1995). Users often forget steps while switching between the instruction window and the application, have difficulty locating components pictured in the instruction window, or mistakenly think that the images of interface elements presented in the instruction window are fully functioning components (Knabe 1995). Since most web-based tutorials use a similar

format, it is likely that users of web-based tutorials will encounter similar problems. While users of printed tutorials are unlikely to confuse images of interface elements with the actual interface elements, they may still have difficulty locating the interface elements or accidentally skip steps. In addition, paper-based manuals and tutorials are more costly to distribute than electronic versions.

Efforts to improve on-line presentation of procedural instructions have centered on two areas: 1) improve the quality of procedural instructions presented in a separate context, and 2) find ways to present help in context.

Early work on presenting procedural instructions demonstrated that adding pictures to textual instructions helped users complete procedural instructions more quickly, but did not improve their accuracy (Booher 1975). Because of the dynamic nature of many user interfaces, researchers have suggested (Schneiderman 1983; Baecker 2002) and evaluated (Palmiter, Elkerton et al. 1991; Palmiter and Elkerton 1991; Harrison 1995) using animated demonstrations to present procedural instructions to users. Palmiter et al. found that participants who used an animated tutorial initially completed test tasks faster than those who used a text-based tutorial, but users of the animated tutorial did not retain their learning a week later (Palmiter, Elkerton et al. 1991; Palmiter and Elkerton 1991). Harrison found that users who used animated tutorials or illustrated textual tutorials learned more quickly than users who used a non-illustrated textual tutorial (Harrison 1995). Researchers have concluded that for many types of software, animated demonstrations will not be broadly effective for presenting procedural instructions (Palmiter and Elkerton 1991; Harrison 1995).

Since many of the problems users encounter when using traditional on-line help or tutorials are caused or exacerbated by the separation between the instructions and the application, other researchers have tried to make help available in the context of the application. Coachmarks (Apple) are markings, typically a circle, cross or check in red or green, drawn over a component in the interface to attract the user's attention to the component relevant to the current step. Sukaviriya et al. (Sukaviriya, Isaacs et al. 1992) animate the cursor over the interface and replace the typical arrow cursor with

representations of the mouse and keyboard to indicate user actions. Coach/2 used an animated picture of a mouse that left a graphical trail and blinked its eyes to show mouse clicks (Selker, Barber et al. 1996). Both techniques show the user what interface components to focus on. However, users may not fully understand what actions are necessary to accomplish a given task. I have not found any studies comparing the performance of participants using in-context instructions with that of participants using more traditional instructions.

While the purpose of procedural instructions is to teach users new skills, once a user has located the relevant set of procedural instructions in a help system, the system may have enough information to perform the instructions for the user. Current versions of the Windows™ Operating System (Microsoft) include a "Show Me" feature that automatically performs the steps described in the instruction window without showing the user how the steps were performed. Although this type of feature does not help users learn new functionality, it does give users an option if they are unable to understand and perform the steps described.

Rather than trying to improve the presentation of procedural instructions, some researchers have tried to limit the number and kinds of mistakes that users can make. Carroll and Carrithers (Carroll and Carrithers 1984) found that users using a specially-created training version of a word-processing package learned to use the program more quickly and performed better on a post-test that measured comprehension than users using the unmodified version of the word-processor. In the training system, when users choose an advanced feature in the training version, the system responds with a dialog box stating that the chosen command is not available in the training system. In a later study, Catrambone and Carroll demonstrated that participants who learned to use the Training Wheels version of the word-processor with the help of a guided-exploration training card were able to transfer their knowledge to an unmodified version of the word-processor (Catrambone and Carroll 1987). Further, these participants were able to perform similar and more advanced tasks as quickly as or more quickly than users who learned to use the unmodified version of the word-processor with the same guided-exploration training card

(Catrambone and Carroll 1987). While limiting the number and kinds of mistakes users can make may help them learn new software more effectively, creating and maintaining separate training versions creates an additional development cost.

6.3.2 Learner Centered Design

Researchers in Learner-Centered software are exploring ways to create software-based scaffolding, support for learners as they are learning a new task (Shashaani 1994). While software-realized scaffolding can take many forms, some Learner-Centered systems provide scaffolding that is intended to guide learners through a process such as creating a simulation or researching a question. Emile, a system for building physics simulations, implements process control by enabling menu items that allow users to access parts of the interface relevant for later stages in simulation building only after they have completed earlier stages (Guzdial 1995). TheoryBuilder, a tool for constructing scientific models, uses reminder messages displayed in pop-up windows to remind learners to perform parts of the process they have neglected. Users can request that TheoryBuilder stop reminding them to complete a given task by clicking a "Stop reminding me" button displayed underneath the reminder message (Jackson, Krajcik et al. 1998). Other systems use the user interface to suggest the process learners should follow but do not require learners to follow it (Wallace, Soloway et al. 1998; Quintana, Eng et al. 1999).

6.3.3 Transparency in User Interfaces

Previous work has examined the use of transparency in interfaces and interaction techniques to solve a variety of user interface problems.

To make better use of screen real estate, Bartlett created stipple-based transparent controls that could exist in an application's work area without obscuring it (Bartlett 1992). Kramer proposed the use of translucent, arbitrarily shaped regions as an alternative to the overlapping windows paradigm that could more fluidly support design activities (Kramer 1994).

The Stencils technique is most closely related to the work done by Bier et al on the See-Through Interface: both use a transparent layer drawn over a user interface to change how an application responds to interface events such as mouse clicks (Bier, Stone et al. 1993; Bier, Stone et al. 1994). A See-Through Interface consists of Toolglass widgets and

Magic Lens filters that appear as though they are on a sheet of transparent glass in between the mouse cursor and the user interface (Bier, Stone et al. 1993; Bier, Stone et al. 1994). A Magic Lens changes the appearance of the user interface beneath it by applying a filter, such as *magnification* to it (Bier, Stone et al. 1993; Bier, Stone et al. 1994). By moving a Toolglass widget over a user interface object and clicking on it, a user can apply that widget's operation to the selected object (Bier, Stone et al. 1993; Bier, Stone et al. 1994). By using their non-dominant hands to position sheets containing one or more Toolglass widgets and Magic Lens filters over the user interface and their dominant hands to control the mouse cursor, users can select and operate on interface objects in fewer steps and with less cursor motion (Bier, Stone et al. 1993; Bier, Stone et al. 1994). Researchers have explored the use of Magic Lenses and Toolglass widgets in several domains including 3D virtual worlds (Viega, Conway et al. 1996), augmented reality (Looser, Billinghurst et al. 2004), generating database queries (Fishkin and Stone 1995), and debugging user interfaces (Hudson, Rodenstein et al. 1997).

6.4 My Approach

Stencils is an interaction technique that is designed to present tutorial instructions in the application context while preventing many kinds of errors. Stencils-based tutorials present users with sequences of full-screen, colored, transparent overlays (or stencils) containing holes. These stencils appear visually overlaid upon the active application interface and intercept mouse and keyboard events. Events occurring over a hole in the stencil are passed to the GUI component beneath the hole. This prevents users from interacting with components covered by the stencil. The holes in the stencil draw the user's eye to the component they should interact with during a given step. Notes on top of the stencil can supply additional information.

One potential problem with presenting tutorial instructions within the application is that users may confuse interface components belonging to the tutorial with those that are part of the application. To prevent this, interface elements associated with the tutorial have a different visual appearance than standard GUI elements, always appear on top of the stencils, and are slightly transparent so the user can see components in the underlying

interface. Based on our user testing, users do not have difficulty differentiating which interface elements belong to the help and which ones belong to the application interface. Stencils can contain four types of objects:

Navigation bars are automatically added to every stencil. They provide "next" and "previous" buttons. The navigation bar also indicates which step the user is currently performing and displays the total number of steps in the current task (see Figure 6.2 A). An "Exit Tutorial" button allows users to close the tutorial at any point.

Holes with attached notes are the most common interface elements. They provide a hole through which the user can interact with the underlying application component and an associated note that the tutorial author can use to provide necessary information. Stencils draws a red arrow to connect the note with its associated hole (see Figure 6.2 B).

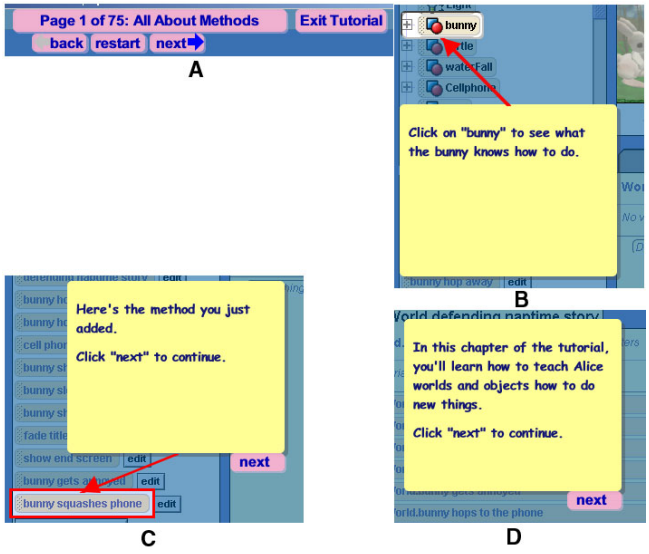


Figure 6.2: Stencil Objects – A) Navigation Bar, B) Hole with Note, C) Frame with Note, and D) Stand-alone note.

Frames with attached notes highlight a particular application component without allowing the user to interact with it. They are typically used to bring aspects of the interface to the user's attention. For example, a frame could point out the results of a completed step. An attached note provides any necessary explanation (see Figure 6.2 C).

Stand-alone notes are used to provide a motivation or describe a goal that will take more than a single step. They are associated only with the stencil, not with any particular element in the application interface (see Figure 6.2 D).

6.5 Interaction Description

In each step of the tutorial, the interface is covered by a stencil. Directions for the current step are displayed on sticky notes. To aid readability, these notes are almost opaque and are placed by the tutorial author over parts of the interface that are least relevant to the current step. However, notes are movable and the user can reposition them to get a better view of a part of the underlying interface, if desired. In user testing, users rarely repositioned a note. The stencil also contains holes over any elements of the interface that the user needs to interact with. Users can perform all necessary actions through the hole. While the rest of the interface is visible, it is not accessible: if users click on elements of the interface that are covered by the stencil, nothing will happen. By making components that are not necessary for the current step inaccessible, Stencils prevents users from accidentally activating an incorrect component and moving the application into an unknown state.

Steps in the tutorial are presented one at a time. Users move to the next step in one of two ways: for steps that require a simple action such as a mouse click or an enter key, the stencil will automatically advance to the next step when it detects the user has performed the correct action; for more complex steps, the user presses a "next" button to advance when s/he has completed the step. Pop-up menus appear on top of the stencil and interface components can be dragged from one hole to another. When users move to the next step in the tutorial, Stencils checks the current state of the application against a saved "correct state" to verify that the user has performed the step correctly. In Alice, the verification is implemented by comparing the changes that have been added to the undo stack based on the users' actions in the current step against a list of saved changes from a "correct" tutorial performance for the current step. In integrating Stencils with other applications, developers would need to implement their own state-checking algorithm. If the user has made any mistakes, stencils displays both a note stating that it believes the user has made a mistake and a "back" button that returns the user to the beginning of the previous step so that they can try again. If the user has correctly performed the step, the system advances to the next step in the stencils-based tutorial.

Occasionally, users want to return to a previous step that they have correctly completed. To allow this, there is a "previous" button as part of the navigation bar. When a user returns to a previous step, Stencils takes them to the beginning of that step by undoing all of the actions they have performed as part of the current and last steps. To move forward, users must complete the steps as directed by the tutorial. By undoing changes when the user goes back a step, Stencils ensures that the state of the program is always consistent with the tutorial instructions for that step.

6.6 Lessons from Formative Evaluation

While developing the Stencils interaction technique, I conducted formative evaluations of three versions of a Stencils-based tutorial with 15 users (7 female), ranging in age from 18 to 60. I chose to start with adult users because they are better able to analyze what aspects of a user interface are and are not working for them. Users were asked to work through short tutorial segments while talking aloud. The tutorial segments included navigating through the interface, selecting menu options, creating new interface elements, and dragging and dropping interface elements. Once the Stencils-based tutorial was usable at a basic level for my adult users, I further refined the tutorial based on testing with approximately 30 home-schooled students between the ages of 11 and 15. The primary lessons I learned were:

1. Visually reinforce the stencil as an overlay on top of the interface

I found that it was important to make holes and notes appear slightly 3-dimensional. Without a hint of 3-dimensionality, users sometimes concluded that the interface was simply tinted blue. With a shadow drawn at the holes to indicate depth and under the notes so they visually float above the stencil, users seemed to understand that the stencil was a layer on top of the existing interface.

2. Bring changes that occur underneath the stencil to users' attention.

Simple actions, such as changes in selection, sometimes cause changes in areas of the user interface that are underneath the stencil. Because the notes and stencils direct users' attention to particular regions of the interface, users are less likely to notice changes in other parts of the interface. If a particular step directs users to perform an action that will

cause a visual change in an area of the interface not exposed by a hole, the next step in the tutorial should use a frame to highlight that change.

3. When completing simple actions, such as mouse clicks or single keystrokes, users expect the tutorial to automatically advance.
I found that while users seem to prefer to control the pacing of complex actions, they expect the tutorial to automatically advance to the next step when they perform simple actions, particularly mouse clicks. Surprisingly, our evaluations indicated that users were not confused by the tutorial sometimes automatically advancing and sometimes requiring manual advancement. Consistency is often a useful strategy for minimizing surprises to the user. In this case, automatically advancing to the next step was the least surprising to users.

4. The underlying application needs to alert the tutorial to changes in the layout of the interface.
Some actions the user takes may cause elements in the interface to shift. If any of these elements have holes or frames over them, these shifts may result in holes or frames over the incorrect parts of the interface.

5. For sequences of steps that have holes over the same screen components, shifting the location of the notes provides a cue that users have moved to the next step.
For many users, the change in position of the notes from one step to the next is a cue that they have advanced to the next step. When one step asks the user to manipulate the same interface elements as the previous step, and the notes do not change location, users may conclude that the tutorial did not advance and inadvertently skip a step.

6.7 Authoring Stencils-based Tutorials

I have created a simple authoring tool for building help stencils to allow non-programmers to create Stencils-based tutorials. The authoring tool runs on top of the active application. Objects are added to the stencil by double clicking on its surface. By default, this creates a hole with an attached note. A right click menu allows authors to create a frame with a note or a stand-alone note rather than a hole with a note. The author

can reposition notes by dragging them on the surface of the stencil and add instructions or explanatory information by typing. Notes are visually attached to their associated holes or frames with a line that updates when they are moved.

After creating the necessary holes in a stencil, the author of a Stencils-based tutorial must perform the actions necessary to complete the current step. Stencils then requests and saves a list of changes the tutorial author has made during that step from Alice. These change lists are saved for every step in the tutorial and used to check that a user working through the tutorial has correctly completed each step.

6.8 Implementing Stencils

My implementation of Stencils is written using the Java Swing framework (Sun 2006). It uses the *glassPane* component in *JRootPane* to draw the stencil over the existing interface and intercept all mouse events. Each stencil maintains a list of holes and components associated with those holes. If a mouse event occurs inside a hole, the stencil passes the event to the interface element below; otherwise the stencil processes the event. Keyboard events are also controlled by explicitly managing which interface elements in the underlying application have keyboard focus. Keyboard events reach an element in the application interface only if that interface element is associated with a hole that has the stencil's focus. A focus listener for the stencil's focused object prevents the user from moving to another interface element using the keyboard.

6.8.1 Modifications to Alice

The implementation of Stencils requires two Java interfaces which allow a two-way communication between the tutorial and the underlying application (i.e. Alice). The Stencils Application Interface allows Stencils tutorials to query Alice about Alice's interface components. Stencils implements the Stencils Update Interface which allows Alice to alert the tutorial to application changes that may require Stencils to adjust. For example, the user resizing the Alice window may shift the position of underlying components.

Alice implements the Stencils Application Interface, a Java interface that provides system-specific functionality to the tutorial. This functionality includes the abilities to:

1. Request the position and size of an interface element given the name of the element.
2. Request the name of the interface element at a particular position on the screen.
3. Request a string representation of the changes.
4. Ask whether or not two strings representing changes in the world are equivalent.
5. Undo changes made to the Alice world and the interface.

Sometimes user actions will cause the layout of the underlying interface to change. For example, a user action might cause a new component to be added to the user interface. If the layout of the components in Alice changes, Alice alerts the tutorial by calling methods in the Stencils Update Interface (a second Java interface), allowing the tutorial to determine the new positions for holes or frames in the current stencil and redraw itself. My implementation of Stencils is written in Java and can be used by any Java application (implementations for other languages are possible). The Stencils implementation includes a basic authoring tool and the ability to play back Stencils-based tutorials. To use Stencils, a Java application must implement the Stencils Application Interface and make appropriate calls to the Stencils Update Interface to alert the Stencils system to changes in the layout of the user interface.

6.9 Evaluating Stencils

To evaluate the Stencils interaction technique, I conducted a study comparing the performance of users given Stencils-based and paper-based versions of the same tutorial.

6.9.1 Participants

Twenty-two Cadette Girl Scouts representing three troops from the Pittsburgh area participated in our study. The girls ranged in age from 12 to 16 years, with 18 of the 22 being between 12 and 13. When asked to rate their skill with computers, 5 chose “very good”, 14 girls chose “good”, 2 chose “fair”, and 2 chose “poor or nonexistent”. Of the 22 girls, one had prior programming experience, and 7 had experience creating webpages. The study was conducted during three one-day, four-hour workshops (one for

each troop). A $10 donation was made to the Girl Scout troop for each girl who participated.

6.9.2 Preparation of Experimental Materials

The paper and Stencils-based tutorials guide users through a sequence of changes to three Alice worlds. The textual directions to users are the same in both conditions.

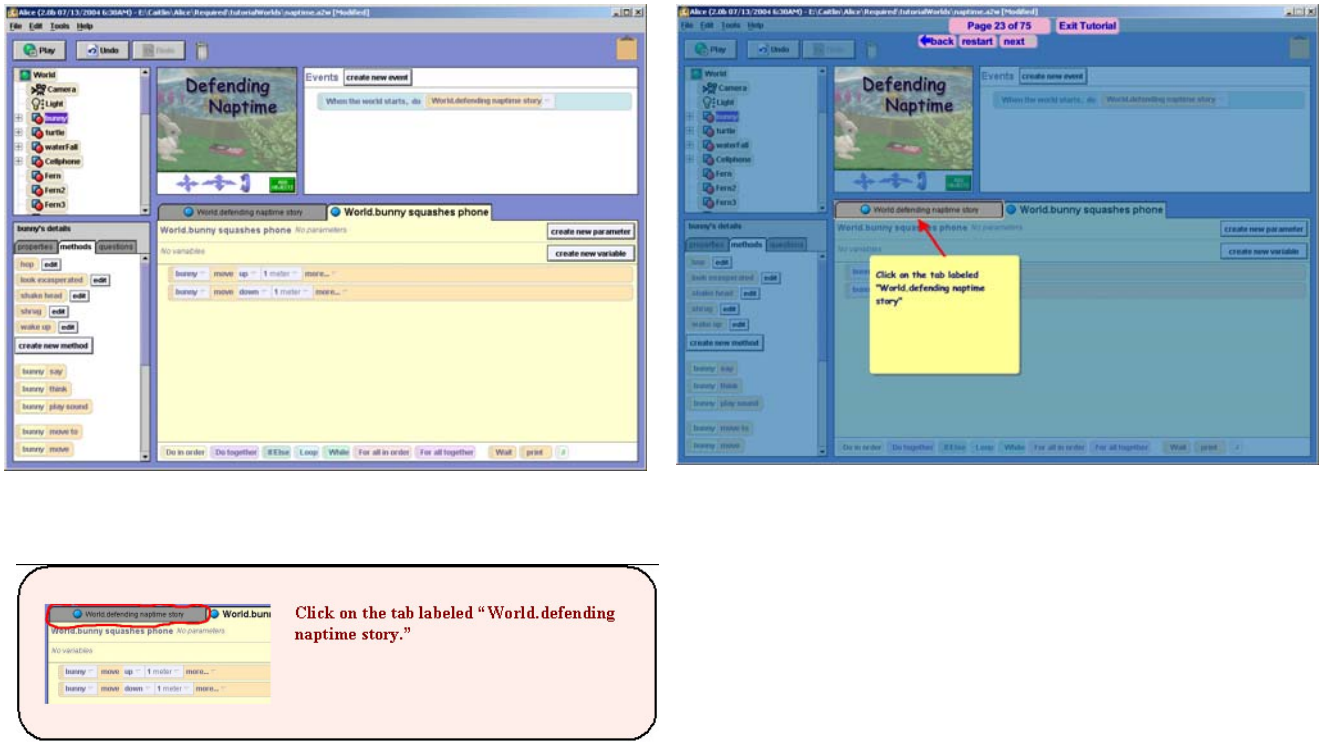


Figure 6.3: A tutorial step in the paper-based tutorial (left) and the Stencils-based tutorial (right).

6.9.3 Paper-based Tutorial

In the paper version of the tutorial, directions for each step are presented beside a picture of the GUI component the user needs to interact with for that step. Because users often have difficulty locating components on screen, the pictures of each component include enough screen context to allow users to easily identify which of the five regions of the Alice interface, their target component lies in (see Figure 6.3). To allow users to check whether or not they have correctly completed the steps in the tutorial, I have included images that show what the relevant parts of the Alice interface should look like at several points throughout the tutorial.

6.9.4 Stencils-based Tutorial

In the Stencils-based version of the tutorial, directions are presented on yellow Post-it™ style notes on the surface of the stencil. Holes in the surface of the stencils draw users' attention to components they need to interact with during the current step of the tutorial. Since our early user testing showed that users often do not notice interface changes that happen underneath the stencil, the tutorial uses frames to draw users' attention to changes that have occurred in the user interface as a result of their actions. When users press the *next* button or the stencil auto-advances to the next step, Stencils checks to make sure that the user has performed all actions necessary for the current step and has not performed extraneous actions.

6.9.5 Procedure

The study took place during three four-hour Alice workshops and used a two-group between-subjects design. Participants were randomly assigned to use either the paper or Stencils-based tutorial. To minimize the effects of differences in computer experience or academic potential among the three troops, an equal number of participants from each troop were assigned to the paper-based and Stencils-based tutorial conditions. Both the paper-based and Stencils-based conditions consisted of 3 participants from troop 1, 3 participants from troop 2, and 5 participants from troop 3, for a total of 11 participants in each condition.

During the workshop, participants completed three tasks: the tutorial, a post-tutorial survey, and a quiz designed to test users' mastery of the material presented in the tutorial. To complete the quiz, users had to complete tasks in a pre-created Alice world to answer multiple-choice questions. Participants needed to perform a variety of actions including: playing the world, finding and calling methods, navigating through the gallery of 3D objects supplied with Alice, adding 3D objects to their worlds, and editing predefined methods.

There were no time limits for completing the tutorial, post-tutorial survey and quiz. Participants were instructed not to help each other, but were told that they could ask the

experimenter for help with the tutorial, if necessary. The experimenter provided help only when requested.

6.9.6 Data Collection

To enable me to study users' performance on both the tutorial and the quiz, I recorded users' actions in two ways. I instrumented the Alice program to record any changes that users made to the current Alice world. To record actions users took that did not result in changes to the current Alice world, I used a locally developed logging program that saves screen captures and records all mouse and keyboard events. I used the screen shots and event logs to reconstruct videos of the users' computer screens as they completed the tutorial and quiz.

Using both the Alice logs and the videos of users' computer screens, I produced transcripts of all actions the users took while completing the tutorial and quiz. In addition, I recorded the amount of time spent on each tutorial and the quiz.

6.9.7 Dependent Measures

My metrics for evaluating the success of participants using the Stencils-based and paper-based versions of the tutorials included error rate, elapsed time, and number of requests for help. My metrics for evaluating learning included the number of correct answers on the quiz and the elapsed time in completing the quiz.

I counted three types of errors: skipped steps, incorrect selections that caused changes to which elements are displayed in the user interface, and incorrect actions that caused changes to the Alice world. All three types of errors have the potential to cripple users' progress through the tutorial. Any actions not described in the tutorial that caused changes to either the interface or the Alice world were counted as errors. However, if a user started an action and canceled it without making a change to the interface or the world, that action was not counted as an error. Additionally, if a user made an error but immediately corrected it (e.g. choosing the wrong item from a menu and immediately changing it to the correct one), it was also not counted as an error.

The elapsed times for the tutorial and quiz were measured beginning when the user opened the file for a given tutorial and ending when they began to load the next file (e.g. clicked on the File menu) or closed the Alice program.

6.10 Results

We used unpaired t-tests to compare the performance of participants using the stencils and paper-based tutorials.

6.10.1 Tutorial Performance

We found that users of the stencils-based tutorial made fewer errors and took 26% less time than users of the paper- based tutorial. Users of the Stencils-based tutorial skipped fewer steps ($p = 0.012$), made fewer erroneous changes to the Alice worlds presented in the tutorial ($p = 0.023$) and to the user interface ($p = 0.069$). In addition to making fewer mistakes, users of the stencils tutorial were 26% faster in completing the tutorial ($p = 0.057$): the mean time for completion of the stencils tutorial was 47 minutes, 22 seconds; the mean time for completion of the paper-based tutorial was 59 minutes, 22 seconds. Users of the stencils-based tutorial also were less likely to require human assistance to make progress on the tutorial ($p = 0.08$). The average number of errors and the distribution of error counts are shown in Table 1.

Table 6.1: Average number of errors and distribution of users' error counts for Paper and Stencils-based tutorials

		Average # Errors per User	# of Users making n Errors					
			0 errors	1-2 errors	3-4 errors	5-6 errors	6-10 errors	>10 errors
Paper	skipped steps	3.82	0 users	3 users	5 users	1 user	2 users	0 users
	interface errors	4.55	5	1	1	2	0	2
	world errors	5.09	1	5	4	0	1	0
	help requests	0.727	7	3	1	0	0	0
Stencils	skipped steps	1.27	5 users	3 users	2 users	1 users	0 users	0 users
	interface errors	1.36	4	5	2	0	0	0
	world errors	1	6	4	1	0	0	0
	help requests	0.08	10	1	0	0	0	0

6.10.2 Quiz Performance

There was no significant difference between the performance of users of the stencils-based and paper-based tutorials on a post-tutorial quiz. Users of the paper-based tutorial answered an average of 5.00 out of 6 questions correctly and users of the stencils-based tutorial answered an average of 4.82 correctly ($p = .746$).

There was also no significant difference in the amount of time necessary for the users of the Stencils-based and paper tutorial to complete the post-tutorial quiz. Users of the Stencils-based tutorial took an average of 20 minutes, 17 seconds to complete the quiz where users of the paper-based tutorial completed the quiz in an average of 18 minutes, 24 seconds ($p = .721$). These averages are based on the completion times for users who

performed all steps in Alice necessary to answer the quiz questions (stencils 8 users, paper 6 users).

6.10.3 Survey Results

In a survey about the tutorial given after users had completed the tutorial but before they had started the quiz, I found that users of the Stencils tutorial were more confident that they completed the steps in the tutorial correctly (Stencils 4.55, paper 3.64 on a 5 point scale $p = 0.029$). However, users of the paper tutorial were more confident that they could build a world in Alice after completing the tutorial than the stencils-based users were (stencils 3.55, paper 4.18 on a 5 point scale, $p = 0.051$).

6.11 Discussion

The Stencils technique is a potential alternative for presenting tutorials. Based on our data, it allows users to attain the same level of learning in a substantially shorter period of time, with fewer errors, and less reliance on human intervention to make progress.
One of our initial concerns with the Stencils approach was that users might move through the tutorial quickly and without understanding what they were learning. While the users of the Stencils tutorial did complete the tutorial more quickly, they appear to have done so without sacrificing learning. Both the paper-based and Stencils-based tutorial groups performed similarly in the number of correct answers and the amount of time it took to complete the quiz.

The increased speed of the users of the Stencils-based tutorial is probably due, at least in part, to the fact that Stencils presents the tutorial instructions in the context of the application. While paper-based tutorials require less context-switching than many online-tutorials presented in a separate window, in a given step the users of the paper-based tutorial had to find their place in the paper tutorial, read the directions, find the appropriate components on screen, and determine what the directions wanted them to do. Users of the Stencils-based tutorial needed only to determine what the directions wanted them to do.

My original goal in pursuing Stencils was to find a method for presenting tutorials that will enable users to successfully manage more complex tutorial examples. In terms of presenting more complex tutorial examples, the largest benefit of Stencils is that when a user does make a mistake, that mistake is immediately caught and the tutorial returns the user to a safe state from which he or she can try that particular step again. While Stencils also helps to reduce the number of user errors, the ability to ensure that users' mistakes cannot cripple their progress is critical for making it possible to show more complex examples within the tutorial.

6.12 Designing Tutorials for Storytelling Alice

Using Stencils, I constructed three tutorials that introduce users to Alice. One of the crucial aspects of these tutorials is that they both introduce the mechanics and show users examples of projects they might be interested in creating.

Previous Alice tutorials have focused on simple examples that users can construct from scratch. Using Stencils, I can present more complex examples that are more representative of the stories that girls envision creating. In addition to leveraging Stencils, I have used some pre-created content in both of the programming tutorials to avoid repetitive tasks or tasks that require skills that are beyond the scope of the tutorial to teach.

6.13 Tutorial 1:

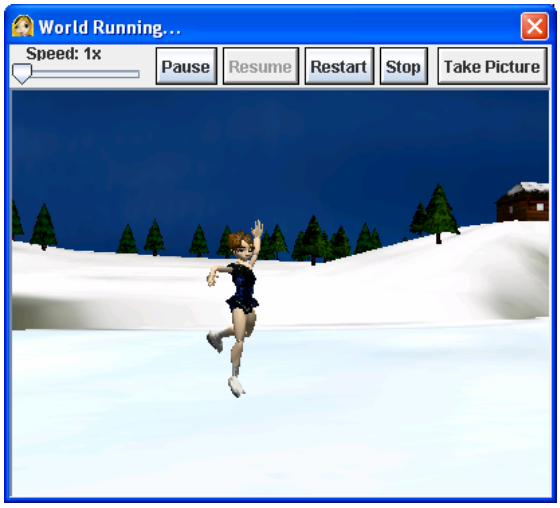


Figure 6.4: In tutorial 1, users created a routine for an ice skater.

In the first tutorial, users create a routine for an ice skater that includes jumps, spins, and skating both forwards and backwards. The tutorial is designed to provide users with a basic overview of the system and introduce them to building simple programs that control the motions of a single object.

The first tutorial:

- Provides an overview of the Alice interface
- Teaches users how to run programs
- Teaches users how to call methods for an object
- Introduces sequential execution and teaches users how to reorder the commands in their programs.

6.14 Tutorial 2:

Figure 6.5: In tutorial 2, users created a story about a boy who falls in love with an ogre.

The second tutorial guides users through building a story in which a troublemaking fairy casts a spell to make a boy fall in love with an ogre. The user adds methods and method calls to the program to make the boy walk to the ogre, kneel down, and confess his love. This tutorial is intended to introduce users to more complex programming in Alice. In the second tutorial, users control multiple objects and multiple methods.

The second tutorial:

- Teaches users how to find and call methods for multiple objects.
- Introduces users to commonly used methods.
- Teaches users how to create and use new methods for characters.
- Teaches users how to change optional parameters to have greater control over the motion of 3D objects.
- Teaches uses how to have motions happen in parallel.

In earlier versions of Storytelling Alice the 2nd tutorial told a story about a napping bunny that was woken up by a cell phone ringing. Wanting to protect his naptime, the bunny hopped over to the phone, squashed it by jumping up and down on it, and then went back to sleep. One of the goals of the second tutorial is to introduce users to the animations they are likely to use the most. After adding high-level animations the Storytelling Alice, it became clear through user testing that the most commonly used animations had shifted

from animations like **`move`** and **`turn`** to higher-level animations like **`say`** and **`walk to`**. The fairy story introduces users to the commonly used high-level animations and is more similar to the kinds of stories that girls wanted to tell based on their storyboards.

6.15 Tutorial 3

Figure 6.6: In tutorial 3, users learn how to set up scenes.

The final tutorial is designed to introduce users to the mechanics of setting up scenes. Because this tutorial does not actually create a running program, it does not include a story or activity in its own right. In the first part, the user learns to arrange objects and move the camera in a graveyard scene that contains a girl and ghost. In the second half, users create a new world and add a boy and a lot of big spiders.

The third tutorial teaches users:

- How to switch between the Alice scene view and programming view.
- How to move objects around in the scene using the mouse.
- How to move the camera in the scene.
- How to undo actions.
- How to create new worlds.
- How to navigate through the Alice gallery.
- How to add new objects to the 3D scene.
- How to make copies of objects.
- How to rotate objects.

In my user tests, it was common for girls going through the storytelling tutorials to get engaged in the story. Girls frequently laughed and made comments about how they thought characters in the story should behave.

6.16 Conclusion

Although developed for use in Alice, Stencils has broad potential for tutorials in other software systems. The Stencils technique has several advantages over previous work in the presentation of procedural instructions. Stencils greatly decreases the number and types of mistakes that a user can make. The visual representation of the stencil draws the user's eye to the component or components necessary for the current step. Each stencil provides a visual indication of what the user can do in that step, without altering the appearance of the application below. A user study comparing the performance of users given a Stencils-based tutorial with that of users given a paper-based version of the same tutorial demonstrated that users of the Stencils tutorial were faster, made fewer errors, required less help from human teachers, and learned the material covered in the tutorial as well as the users of the paper tutorial. Stencils will likely be of greatest benefit in interfaces that are highly spatial and primarily point-and-click with some typing.

Chapter 7 Evaluation Methodology

7.1 Introduction

In this section, I will describe the system to which I compared Storytelling Alice, define the metrics I used to compare the experiences of girls in the control and experimental groups, describe how the summative evaluation sessions were conducted, and provide demographic information about the 88 girls who participated in the summative evaluation of Storytelling Alice.

In discussing the evaluation of Storytelling Alice, it seems natural to begin by revisiting my hypothesis about the potential impact of a storytelling focus on girls' experience and interest in learning to program:

> *Girls who are introduced to programming as a means to a motivating end, such as storytelling will show more evidence of engagement than girls introduced to programming as an end in and of itself. I will be able to find quantifiable behavioral differences between the two groups, such as number of lines of code or time spent working on programs.*

7.2 Choosing a Comparison System

The phrase "…a traditional approach [to introducing programming] that focuses on teaching programming as an end in itself" raises two questions: 1) What programming

environment do I use in presenting programming as an end in itself and 2) How will the teaching occur?

A stereotypical introduction to computer programming often begins with building a program that prints out the message "Hello world" and may advance to simulating a bank account balance or generating the n^{th} Fibonacci number. Often, students write programs in introductory computer science using professional programming languages; today, many introductory computer science classes are taught using Java. Unlike Java, Alice provides mechanical supports to ease the process of learning to program. In Alice, users construct programs by dragging and dropping code elements, a style of program construction that prevents syntax errors. Further, Alice programs animate all state changes that occur, enabling users to watch the behavior of their program and more easily identify mistakes. Comparing girls' experiences using Storytelling Alice with those of girls who are introduced to programming using Java makes it impossible to separate the impact of storytelling support from the impact of Alice's mechanical supports for programming. To isolate the impact of storytelling support, I chose to compare Storytelling Alice to a version of Alice without storytelling support.

Because I used an early, pre-release version of Alice 2.0 as the basis for Storytelling Alice, one natural strategy is to compare girls' experiences using Storytelling Alice and Alice 2.0. However, Storytelling Alice and Alice 2.0 were developed in parallel and some of the changes made in support of storytelling were also added to Alice 2.0. Rather than comparing Storytelling Alice against Alice 2.0, I created a version of Alice that does not include any of the changes I made to support storytelling (Generic Alice).

Storytelling Alice and Generic Alice differ in three ways: the online tutorial, built-in support for storytelling, and the gallery. I will discuss each of these in turn:

7.2.1 Tutorial

Both Storytelling Alice and Generic Alice have a series of three tutorials that are presented using the Stencils interaction technique (see Chapter 7 for more details on Stencils). The tutorials in Storytelling Alice and Generic Alice both cover the same skills

and concepts in the same order. However, where Storytelling Alice introduces skills and concepts within the context of story-based projects, Generic Alice uses examples chosen for simplicity of exposition of the skill or concept.

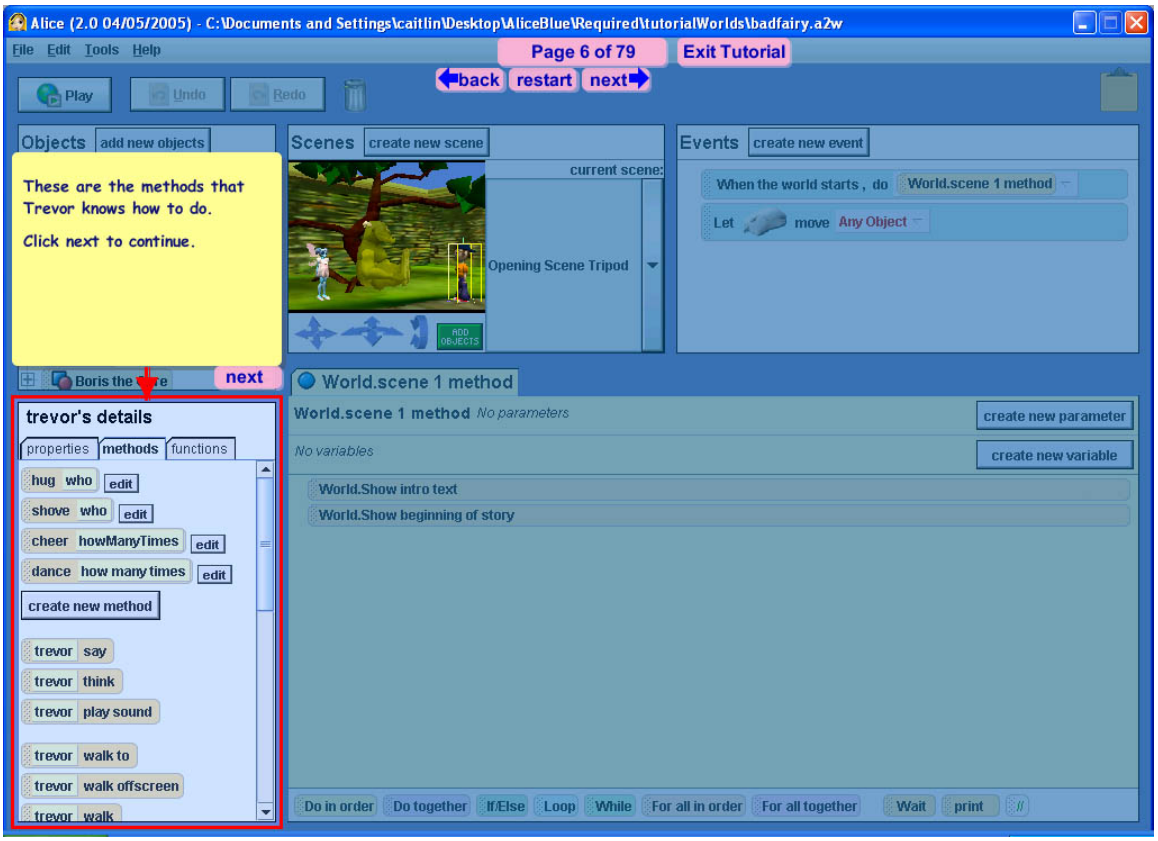


Figure 7.1: One step in a Stencils tutorial.

In addition to choosing different tutorial examples, the two tutorials motivate the examples differently. In Storytelling Alice, the tutorial presents itself as a computer program for making animated movies similar to those of Pixar or Dreamworks. The motivation for introducing concepts comes from the overarching story that users build through the tutorial. In contrast, the tutorial in Generic Alice presents itself as a way to learn how to program a computer. Where the Storytelling Alice tutorial introduces the commonly used methods by suggesting action in the story and then guiding the user through building that action, Generic Alice tutorial introduces commonly used methods both by describing the ways objects can move and by guiding the user through examples of using move, turn, roll, and resize.

A more detailed comparison of the Storytelling and Generic Alice tutorials is presented below.

7.2.1.1 Tutorial 1

Figure 7.2: Tutorial 1 in Storytelling Alice (left) and Generic Alice (right).

Tutorial 1 covers the following material:

- An overview of the Alice interface
- How to run programs
- How to call the methods for a single object by dragging and dropping method "tiles" into the method editor.
- Basic sequencing and reordering of method calls.

In Storytelling Alice, these concepts are introduced within the context of creating a routine for an Ice Skater. In Generic Alice, the same concepts are introduced through moving and turning a fishing boat.

7.2.1.2 Tutorial 2

Figure 7.3: Tutorial 2 in Storytelling Alice (left) and Generic Alice (right).

Tutorial 2 covers the following material:

- How to find and call methods for multiple objects
- How to use the most common methods
- How to create and use new methods for objects.
- How to change optional parameters in order to control the motions of objects with greater precision.
- How to make multiple methods execute in parallel

In Storytelling Alice, users learn these skills while building a story in which a fairy casts a spell on a boy. In Generic Alice, users make a mailbox move, turn, and open its door.

Because Storytelling Alice and Generic Alice provide different sets of methods, the commonly used animations in Storytelling Alice and Generic Alice differ. In Storytelling Alice, users most commonly want to characters to walk around their scene and talk to each other. The Storytelling Alice version of tutorial 2 introduces walk to, say, kneel, touch, and look at. In Generic Alice, move, turn, and roll are the most commonly used methods. Each tutorial introduces the most commonly used methods for that system.

7.2.1.3 Tutorial 3

Figure 7.4: Tutorial 3 in Storytelling Alice (left) and Generic Alice (right).

Tutorial 3 covers the following material:

- How to switch between the Alice scene view and the programming view.
- How to move objects within the scene using the mouse
- How to move the camera within the scene.
- How to undo actions.
- How to create new worlds.
- How to navigate through the Alice gallery.
- How to add new objects to the 3D scene.
- How to make copies of objects.
- How to rotate objects.

Tutorial 3 provides users with an overview of how to set up their own scenes within Alice. It is broken into two parts: in the first part, users learn to position objects and the camera; in the second part, users create a new world and then add objects to it. In Storytelling Alice, users position a girl and ghost in a graveyard scene and then create a new scene with a timid boy and a scary spider. Although there is no explicit story, both of the scenes in Storytelling Alice suggest potential conflicts through threatening characters: a ghost and a spider. In Generic Alice, users position a lighthouse in part one and then add a beach house and chairs in part two.

7.2.2 Storytelling Support

Storytelling Alice includes two kinds of built-in support for storytelling: a set of high-level animations that more closely match the kinds of actions girls needed for their stories and support for creating multiple scenes. Many girls' storyboards included multiple scenes.

Storytelling Alice and Generic Alice provide different basic animations. Table 7.1 shows a comparison of the available animations for a humanoid character in Storytelling Alice and Generic Alice.

The animations in Generic Alice were inspired by basic transformations in 3D graphics (translate, rotate, and scale) and modified based on user testing to enable non-technical users to use them. In Generic Alice, users must individually rotate each joint (i.e. hip, knee and ankle) of a character's legs to make the character walk.

Through studying the kinds of actions that middle school girls want to incorporate into their stories, I developed an alternative set of higher-level animations that more closely matches the actions girls envision characters in their stories performing. In Storytelling Alice, basic character actions like walking, sitting, and touching other objects are provided.

In Generic Alice, all objects including people, soda cans, and furniture can perform the methods listed in the "Generic Alice" column of Table 7.1. In Storytelling Alice, there are different types of objects: humanoid characters, non-humanoid characters, and scenery objects. Humanoid characters perform the animations listed in the table above. Non-humanoid characters perform the subset of humanoid animations that do not require arms and legs. This excludes the following humanoid animations: walk to, walk, walk offscreen, sit on, lie on, kneel, and fall down. In place of walk animations, non-humanoid characters can move in a direction (by sliding) or move to an object or another character. Scenery objects can only perform simple motions like move and turn. A more detailed

listing and description of the animations in Storytelling Alice can be found in Chapter 5. A description of the animations in Generic Alice can be found in Chapter 3.

Table 7.1: A list of the animations a person can perform in Storytelling Alice and Generic Alice in the order they appear in the user interface. A small number of animations including move and turn appear in both systems.

Storytelling Alice	Generic Alice
Say, think	Move
Play sound	Turn
Walk to, walk offscreen, walk	Roll
Move	Resize
Sit on, lie on	Play sound
Kneel	Move to
Fall down	Move toward
Stand up	Move away from
Straighten	Orient to
Look at, Look	Point at
Turn to face, turn away from	Set point of view to
Turn	Set pose
Touch, Keep Touching	Move at speed, turn at speed, roll at speed

7.2.3 Scene Support

In addition to the high-level animations provided in Storytelling Alice, I found through user testing that many of the stories that girls wanted to create required multiple scenes. Storytelling Alice helps users to create and manage multiple scenes (additional details about multiple scene support can be found in Chapter 5). Generic Alice does not provide explicit support for creating multiple scenes but users can create the effect of multiple scenes by combining features available in Generic Alice. However, the process of creating multiple scenes in Generic Alice is too complex for typical novice users.

7.2.4 Gallery

Both Storytelling Alice and Generic Alice come with a gallery of 3D objects that users can add to their Alice worlds. The Generic Alice gallery contains a broad selection of more than 350 objects ranging from animals to buildings to buttons and switches.

Through user testing, I found that the gallery of objects can be a source of story inspiration. The gallery of 3D objects in Storytelling Alice includes both characters with clear roles and animations that require explanation within the story, two techniques that helped girls to find and develop story ideas. Additional information about the design of the storytelling gallery can be found in chapter 6.

Figure 7.5 below shows the some of the characters and scenes available in the Storytelling Alice Gallery. Figure 7.6 shows some of the objects available in the Generic Alice Gallery.

Figure 7.5: Character (above) and scenes (below) from the gallery in Storytelling Alice.

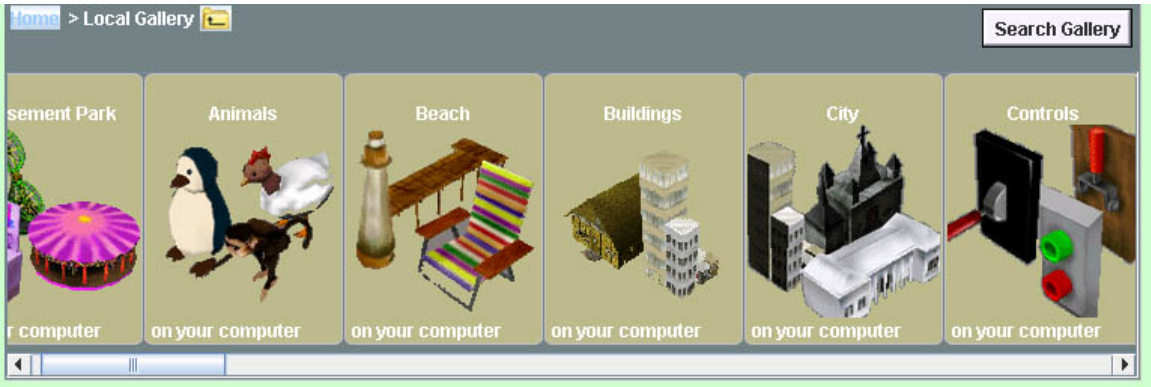


Figure 7.6: Objects from the gallery in Generic Alice.

7.2.5 Teaching

Teachers' presentation of a classroom lesson can have a large impact on students' interest in and success at mastering the lesson material. To control for that (as much as possible), I used the online tutorials to present all Alice and programming related material and gave one set of verbal directions to all participants (e.g. "find the shortcut to Alice icon on your desktop and double-click it"), both those using Storytelling Alice and those using Generic Alice. On a practical level, because Storytelling Alice and Generic Alice are different, it would be nearly impossible to present a classroom-style lesson to both groups without calling attention to the differences between the systems and unintentionally biasing students. Instead, I presented the "teaching" through the tutorials and accompanying materials. To minimize the risk of embedding a bias in the tutorial and accompanying materials, I tried to reuse as much of the structure and language as possible for both of the conditions. Copies of all the materials given to participants are available in the appendix.

7.2.6 Supplementary Materials

Particularly with middle school students, there are limits on the amount of material one can put into a tutorial and expect students to master at one time. In both the Storytelling Alice and Generic Alice tutorials, students typically master the basics of creating simple, sequential programs. While the tutorials in both systems introduce more advanced skills like creating methods and having multiple methods execute in parallel, I have seen few participants who grasp these concepts after completing the tutorial and without further

practice (outside the context of the tutorial). However, I wanted to provide a way for participants to learn about looping and writing parameterized methods.

To provide participants with reinforcement for methods and parallel execution and introduce other programming concepts, I created booklets that explain and show an example of using methods, methods with parameters, loops, and parallel execution. As with the tutorials, the Storytelling Alice booklet presents concepts in a storytelling context and Generic Alice presents concepts within a programming context. The booklets were provided as an additional resource; participants were not required to read the booklets. Copies of the booklets for Storytelling Alice and Generic Alice can be found in the appendix.

7.3 Methods

The summative evaluation of Alice took place as a series of one-time, four hour workshops. Participants were randomly assigned to either the control group (using Generic Alice) or the experimental group (using Storytelling Alice). To avoid biasing participants based on the names Storytelling Alice and Generic Alice, I referred to the systems by their screen background colors: Storytelling Alice was called Alice Blue (because the background color was blue) and Generic Alice was called Alice Green.

At the beginning of the session, participants completed a short survey that asked questions about their academic and computer background as well as their interest in computer science. When all participants had completed the survey, I explained that they were going to try out a computer program called Alice and that I was testing two different versions of Alice. To avoid exposing participants in one condition to the version of Alice they were not using, I set the computers for Storytelling Alice and Generic Alice up in different parts of the room so that participants could not see what the screens of participants in the other condition. I also instructed all participants that they could talk freely to other participants in their condition, but they could not talk with participants in the other condition.

I gave all instructions to the group as a whole, rather than addressing the participants using Alice Green and Alice Blue separately. I asked each participant to complete the three tutorials and then build "something to show everyone" at the end of the four hour session. Each participant was given a booklet showing more advanced programming concepts within their version of Alice as a reference, but participants were not required to use the references. I do not have a record of which participants used the references, but in user testing session it was rare for users to read through them.

At the beginning of the session, I told participants that they were free to ask questions. I did not initiate contact with any of the participants, but I answered questions that participants asked as concisely as possible. However, for the most part, participants in both conditions eventually worked through the problems they encountered.

Participants had two hours and fifteen minutes to complete the tutorial and create a program using the version of Alice to which they were assigned. After two hours and fifteen minutes, participants completed a survey and a programming quiz. When I handed out the survey and quiz, I instructed participants to complete it on their own without talking to others.

For middle school aged children, four hours is a long time. After all of the surveys and quizzes were returned, I gave groups ten to fifteen minutes to use the bathroom and get a drink of water or eat a snack (some groups brought along snacks). Then, I gave participants thirty minutes to try the version to which they were not assigned (i.e. the participants who used Storytelling Alice tried Generic Alice and vice versa) and decide which version of Alice they wanted to take home on CD. Because participants did not need to complete the tutorial for the second system (the mechanics of both systems are the same) and were not required to create a finished world, participants only needed enough time to get a sense of the differences between the systems. Participants were not given a copy of their Alice program to take home but were told that they could email me to get a copy of it. While some participants seemed initially disappointed that they could

not take their programs home, none of the participants in either condition contacted me to get their programs.

Finally, I asked participants to pick a program that they created in either of the two versions of Alice to show everyone. Participants opened the relevant version of Alice and loaded their program. When they were ready, participants walked around the classroom and watched the programs that other participants had created.

A full schedule of the 4-hour evaluations session is shown in Figure 7.7.

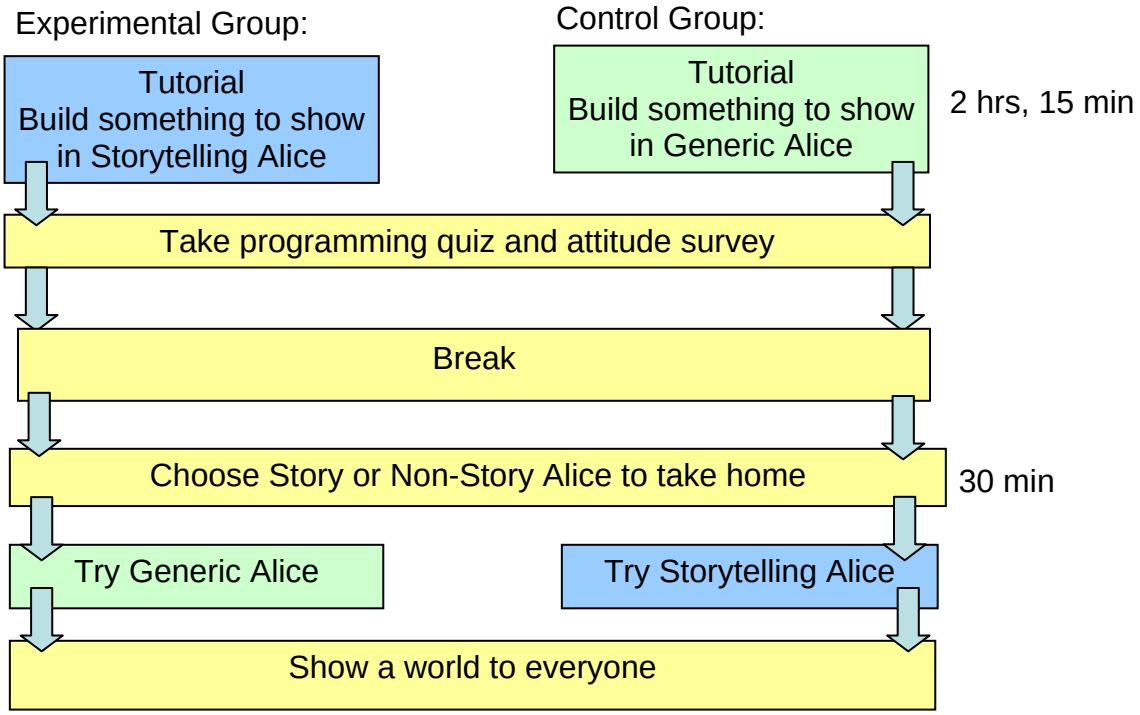


Figure 7.7: Schedule for evaluation workshops.

7.3.1 Data sources:

I collected five types of data: pre- and post-surveys, programming quiz performance, logs of participants' actions within Alice, the programs that participants created, and behaviors such as which participants snuck extra time to work on their Alice programs and which version of Alice participants chose to take home.

7.3.2 Surveys:

The pre-survey asked questions about participants' academic background including the type of school they attend (i.e. public, private, or home-school) and grades and participants' computer usage including number of hours of usage in the last week, skill level, prior programming experience, etc.

The post-survey included an attitude survey focusing on participants' experience with Alice, questions about their future interests in using Alice and/or taking computer science courses, how they would describe their experience using Alice to a friend, etc. Both the pre- and post-surveys can be found in the appendix.

7.3.3 Programming Quiz:

After completing the post-survey, participants took a short programming quiz. The quiz asked participants to predict the behavior of short Alice programs. Each question had four descriptions of the programs' behavior for participants from which participants selected an answer. Quiz questions covered sequential programming, events, parallel execution, loops, method calls, and parameters. A copy of the programming quiz can be found in the appendix.

7.3.4 Log Files:

I instrumented both Storytelling Alice and Generic Alice to record all of the actions that users take within the program. These logs include both programming activities such as adding, deleting, moving, or modifying a line of code, creating a method, adding a loop, etc and non-programming activities such as adding, deleting, or positioning characters or objects within the 3D scene. The Alice logs allow me to ask a variety of questions including:

- How much time participants spend on the programming and non-programming aspects of Alice.
- Which programming constructs participants use in their programs.

- Which of participants' programs (the program created in either Storytelling Alice or Generic Alice) they choose to share with their peers.
- Whether or not participants sneak extra time to work on their programs before sharing them.

7.3.5 Alice Programs:

In addition to examining the Alice logs, I also collected the programs that users created. While the log files are an excellent source of countable data, it is difficult to extract more qualitative information. Participants running programs are a potential source of more qualitative information such as what kinds of programs (e.g. stories, artistic animations, random motion, etc) participants in both condition tended to write as well as information about what kinds of goals and actions prompt exploration of different programming concepts.

I also recorded which version of Alice (Storytelling or Generic) participants chose to take home.

7.4 Participant Demographics:

A total of 88 girls participated in the evaluation of Storytelling Alice; 45 were in the control group using primarily Generic Alice and 43 were in the experimental group using primarily Storytelling Alice. Participants in both the control and experimental groups were given thirty minutes to try the version of Alice to which they were not assigned.

The average age for the participants was 12.6 years (12.8 years in the control group and 12.5 in the experimental group) and nearly all participants were in grades 5-9, with the majority in the 7^{th} and 8^{th} grades. Overall, 76 participants reported attending public school and 12 (7 in the control group and 5 in the experimental group) attend private school. No home-schooled students participated in the evaluation of Storytelling Alice. Most participants reported getting "mostly A's" or "A's and B's". This raises a question about whether or not the test pool incorporated a representative group of students. I was

unable to find information about average grades for Pittsburgh area middle school students, although in observing participants working with Alice, it was clear that participants had a broad range of academic abilities. Determining whether or not different ethnic groups are represented in the same proportions as the general population can also provide insight into whether or not the participants were a representative group of students. Based on the 2000 census, Allegheny County was 84% Caucasian, 12% African-American, and 3% from other ethnic groups. The participants for the evaluation of Storytelling Alice were 89% Caucasian, 9% African-American, and 2% from other ethnic groups (in this case, Asian and Hispanic). While the proportions from each ethnic group are not an exact match, they are not wildly divergent which supports my sense that the evaluation participants represented a fairly typical mix (for Western Pennsylvania) of middle school aged girls.

Table 7.2: Academic Demographics for Girl Scouts who participated in the summative evaluation.

		Storytelling Alice	Generic Alice	All Participants
Number of Participants		43	45	88
Ages	High:	16	17	17
	Low:	10	11	10
	Mean:	12.5	12.8	12.6
	Standard Deviation:	1.3	1.2	1.3
Grade in School	Grade 5:	4	0	4
	Grade 6:	8	6	14
	Grade 7:	13	19	32
	Grade 8:	12	15	27
	Grade 9:	4	3	7
	Other grades:	2	2	4
Grade in School	High:	11	12	5
	Low:	5	6	1
	Mean:	7.3	7.5	3.2
	Standard Deviation:	1.3	1.1	1.0
School Type	Public:	38	38	76
	Private:	5	7	12
	Home-school:	0	0	0
Academic Performance	Mostly A's:	22	16	70
	A's and B's	13	20	65
	Mostly B's:	3	5	10
	B's and C's:	4	4	8
	Mostly C's:	1	0	1
	C's and D's:	0	0	0
	Mostly D's and below:	0	0	0
	No Answer:	0	0	0

The 88 participants reported using computers for an average of 9.3 hours per week. 87.5% of participants reported using computers for a mix of entertainment and schoolwork. Nearly 41% reported that they use computers mostly for entertainment. 10.2 % of participants had previously written a computer program and 36.4% had created a web page. Most participants (82.9%) described their skill with computers as either "good" or "very good." However, 38.6% of participants reported asking for help with installing new software either "very frequently" or "somewhat frequently" and 56.8% reported asking for help with troubleshooting either "very frequently" or "somewhat frequently."

Table 7.3: Computer-related Demographics for Girl Scouts who participated in the summative evaluation.

		Storytelling Alice	Generic Alice	Total Number	%
Number of Participants		43	45	88	
During the last week, how often did you use a computer for any purpose?	High:	70	45	70	
	Low:	0	0	0	
	Mean:	9.4	9.3	9.3	
	Standard Deviation:	12.9	9.9	11.4	
What do you use computers for?	Only schoolwork:	2	2	4	4.5%
	Mostly schoolwork, some fun:	10	9	19	21.6%
	Equally for schoolwork and fun:	11	11	22	25.0%
	Mostly fun, some schoolwork	16	20	36	40.9%
	Only fun:	4	2	6	6.8%
	No answer	0	1	1	1.1%
Have you ever written a computer program?	No:	30	30	60	68.2%
	Yes:	4	5	9	10.2%
	Don't know:	9	10	19	21.6%
	No answer:	0	0	0	0.0%
Have you ever made your own web page?	No:	32	20	52	59.1%
	Yes:	9	23	32	36.4%
	Don't know:	2	2	4	4.5%
	No answer:	0	0	0	0.0%
What is your skill level at using computers?	Poor:	1	0	1	1.1%
	Fair:	5	2	7	8.0%
	Good:	14	17	31	35.2%
	Very good:	20	22	42	47.7%
	Excellent:	3	4	7	8.0%
	No answer:	0	0	0	0.0%
When something goes wrong with your computer, how frequently do you ask friends or family members for help fixing it?	Very frequently:	12	11	23	26.1%
	Somewhat frequently:	14	13	27	30.7%
	Neither frequently nor infrequently:	7	7	14	15.9%
	Somewhat infrequently:	7	9	16	18.2%
	Very infrequently:	3	5	8	9.1%
	No answer:	0	0	0	0.0%
When you want to install a new computer program, how frequently do you ask friends or family members to help you install it?	Very frequently:	10	9	19	21.6%
	Somewhat frequently:	6	9	15	17.0%
	Neither frequently nor infrequently:	7	5	12	13.6%
	Somewhat infrequently:	7	4	11	12.5%
	Very infrequently:	12	18	30	34.1%
	No answer:	1	0	1	1.1%

Chapter 8 Summative Evaluation Results

8.1 Introduction

This chapter describes and compares the learning and motivation of Storytelling Alice and Generic Alice. Participants in both conditions showed statistically similar mastery of programming concepts. However, participants who used Storytelling Alice showed more evidence of motivation than those who used Generic Alice. Participants who used the Storytelling version of Alice spent a greater percentage of their time programming, performed as well as users of Generic Alice on a post-Alice programming quiz, had a stronger interest in taking a future Alice class, and were more likely to sneak extra time to work on their Alice programs than users of Generic Alice.

8.2 Participants' Behavior within Alice

In both versions of Alice, the process of creating a program involves three activities: 1) selecting 3D objects and positioning them within the 3D scene (which I will call scene layout) 2) constructing and editing programs and 3) running programs. The process of creating a program in both Storytelling and Generic Alice is iterative and users typically return to each of the three activities multiple times. One important metric in determining the success of Storytelling Alice is how much time girls spend on programming related activities (either editing or running their programs). Increasing girls' interest in using

Storytelling Alice is of limited benefit if girls elect to focus on scene layout rather than programming activities.

Both versions of Alice log all of the actions that users take in constructing their programs. Based on the log files, I can track the amount of time users spent on scene layout, program construction, and running their program. Participants were given 2 hours and 15 minutes to complete the tutorial and create a program in their assigned version of Alice. Participants worked with Alice for the entire 2 hours and 15 minutes. However, because participants completed the tutorial at different rates, not all participants had the same amount of time to work on their programs. Users typically completed the tutorial in 30-45 minutes. Because of the varying amount of time users had for self-directed Alice use, I compare the percentage of their total Alice time that users devoted to each activity.

I analyzed the differences in the percentage of time participants using Storytelling Alice and Generic Alice with an unpaired t-test. Overall, participants who used Storytelling Alice spent 54% ($p<0.001$) more time editing their programs and 42% ($p < 0.001$) less time laying out their scenes. Users of Storytelling Alice also spend slightly more time running their programs. In languages like Java or C++, most beginning programs run almost instantly so users do not devote a significant amount of their time running their programs. In Alice, programs are animated so users watch their programs execute. Consequently, running programs in Alice includes the time that users spend identifying problems or debugging.

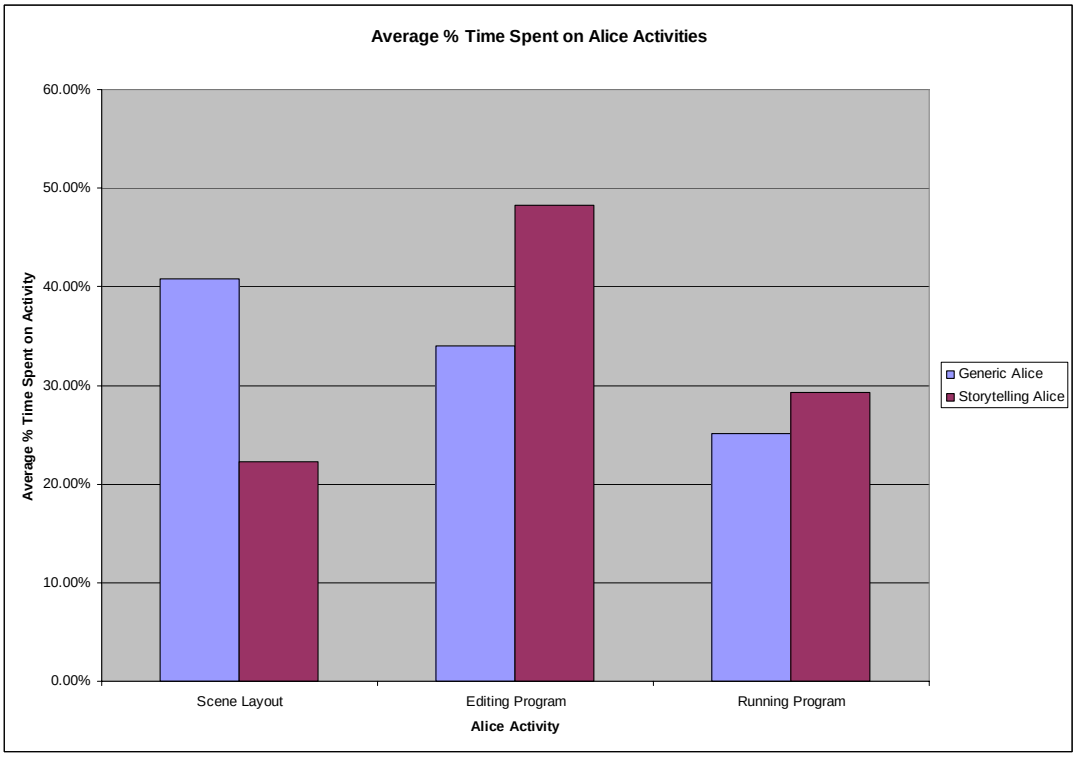


Figure 8.1: Average Percentage of Time users of Generic Alice and Storytelling Alice spent on scene layout, program editing, and running their programs.

Table 8.1: Percentage of time participants using Generic Alice and Storytelling Alice spent on scene layout, program editing, and running their programs.

		Alice Version	Number of Subjects	Mean % of Time	High	Low	Standard Deviation	p-value
Alice Activity	**Scene Layout**	Generic Alice	45	40.80%	96.80%	12%	21.9	
		Storytelling Alice	43	22.30%	47.20%	2.70%	10.6	p < 0.001
	Editing Program	Generic Alice	45	34%	61.30%	3.20%	14.1	
		Storytelling Alice	43	48.30%	67.10%	29.20%	8.4	p < 0.001
	Running Program	Generic Alice	45	25.10%	44.90%	0%	10.3	
		Storytelling Alice	43	29.30%	47.50%	8.30%	8.58	

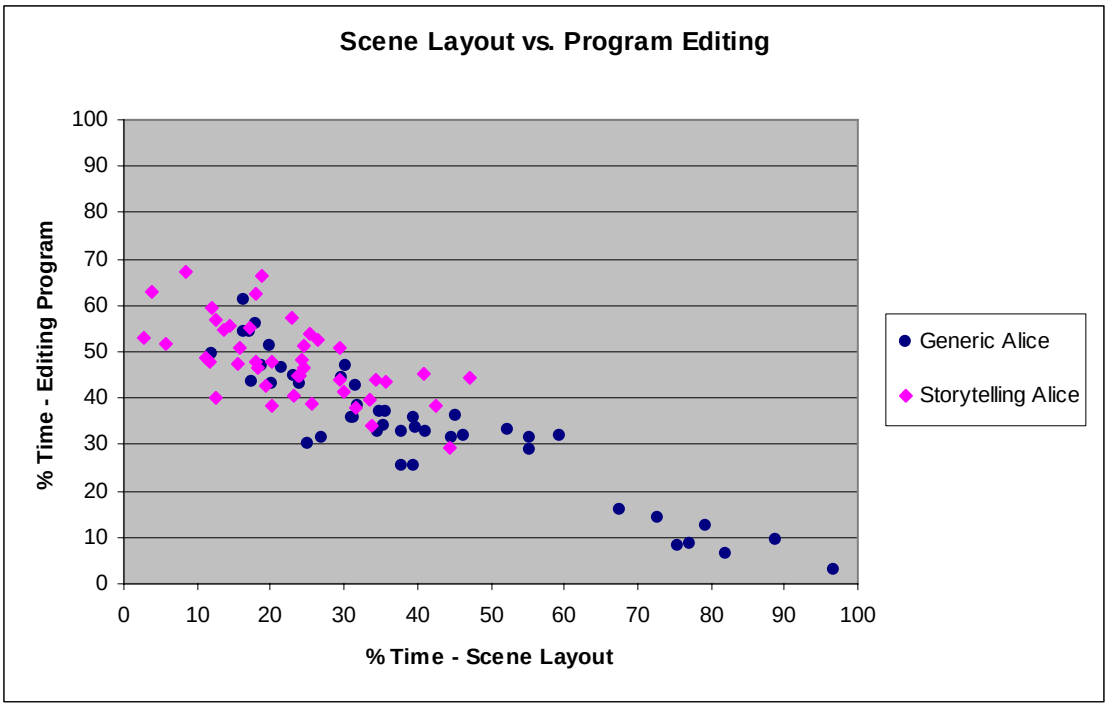


Figure 8.2: Percentage of time spent on scene layout vs. program editing for participants who used Generic Alice and Storytelling Alice.

There was a broad spectrum of ways that participants who used Generic Alice partitioned their time. At one end of the spectrum, users spent 10-20% of their time on scene layout and 50-60% on editing their program. At the opposite end of the spectrum, users spent 70-100% of their time on scene layout and 0-20% of time editing their program.

In contrast to the usage patterns for Generic Alice, participants using Storytelling Alice were more tightly grouped together. At one end of the spectrum, users spent 0-10% on scene layout and 50-70% of their time editing their program. At the opposite end of the spectrum, users spent 40-50% of their time on scene layout and 30-50% of their time on programming. While there were several Generic Alice users who spent nearly all of their time on scene layout, there is no similar group among Storytelling Alice users.

Based on my observations of participants and on participants' Alice logs, there are two factors that may account for the differences in the usage patterns of Storytelling Alice and Generic Alice users: 1) users of Storytelling Alice were somewhat more likely to find a goal they were committed to pursuing than users of Generic Alice and 2) users of Generic

Alice were somewhat more likely to get frustrated while programming and return to scene layout and users of Storytelling Alice.

In observing participants using Generic Alice, I found that the process of selecting and arranging objects in the 3D world is a rewarding activity for many users. As a computer scientist it is easy to view the process of selecting and laying out objects as a necessary but fairly uninteresting part of using Alice. To middle school girls, selecting and arranging objects may provide a chance for self-expression, similar in some respects to the process of choosing clothing for an avatar or furniture to go in a virtual house.

8.2.1 Programming Constructs

In addition to examining how participants chose to spend their time, I examined participants' usage of programming constructs such as loops, methods, and parallel execution.

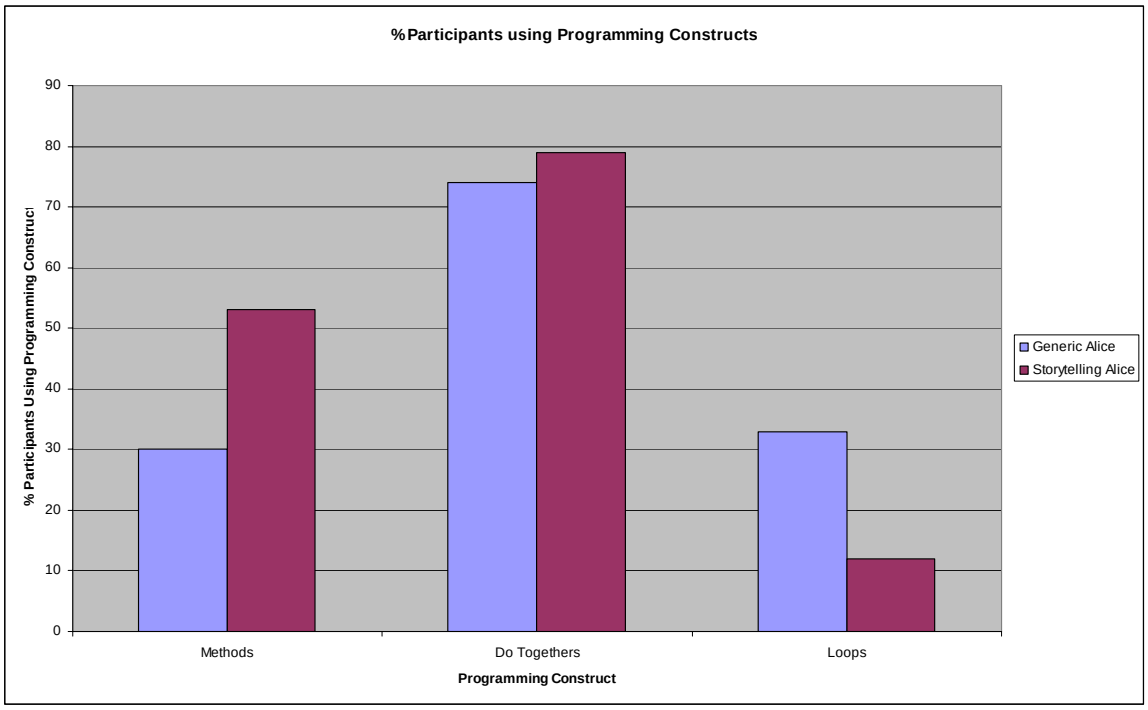


Figure 8.3: Percentage of Participants who used methods, do togethers, and loops in their programs.

	% Participants who created methods	% Participants who used Do Togethers	% Participants who used Loops
Generic Alice	30	74	33
Storytelling Alice	53	79	12
p-value	$p < 0.05$		$p < 0.05$

Figure 8.4: Percentage of participants who used methods, do togethers, and loops in their programs.

Users of both Storytelling Alice and Generic Alice experimented with programming constructs beyond simple sequences. A majority of the participants in both groups used Do Togethers to have multiple animations occur simultaneously. 53% of the users of Storytelling Alice created a new method and used it in their program as opposed to 30% of the users of Generic Alice. 33% of the users of Generic Alice used loops as compared to 12% of the users of Storytelling Alice. Given the relatively short period of time that users spent programming, it is unreasonable to expect them to master all the programming constructs typically taught in an introductory computer science course. The users of Generic Alice spent approximately 62 minutes editing and testing their programs. The users of Storytelling Alice spent approximately 81 minutes editing and testing their programs.

8.3 Participants' Programming Sessions

To give a flavor for what participants in each condition did, I will contrast three sessions of individual users from each condition: 1) a case in which the user spent a lot of time on scene layout and less time on programming relative to other subjects in their condition (low programming), 2) an average case (average programming), and 3) a case in which the user spent a lot of time on programming and less on scene layout relative to other subjects in their condition (high programming). I will refer to the specific users by their subject identification strings which consist of the name of the computer they used during the evaluation (e.g. fish, castle, instruments, etc) and the date of the workshop. To enable easy comparison between the groups of low, average, and high programmers, I will describe both low programming examples, then both average programming examples, and finally both high programming examples.

8.3.1 Low Programming: Subject Fish_02_11_2006 Using Generic Alice

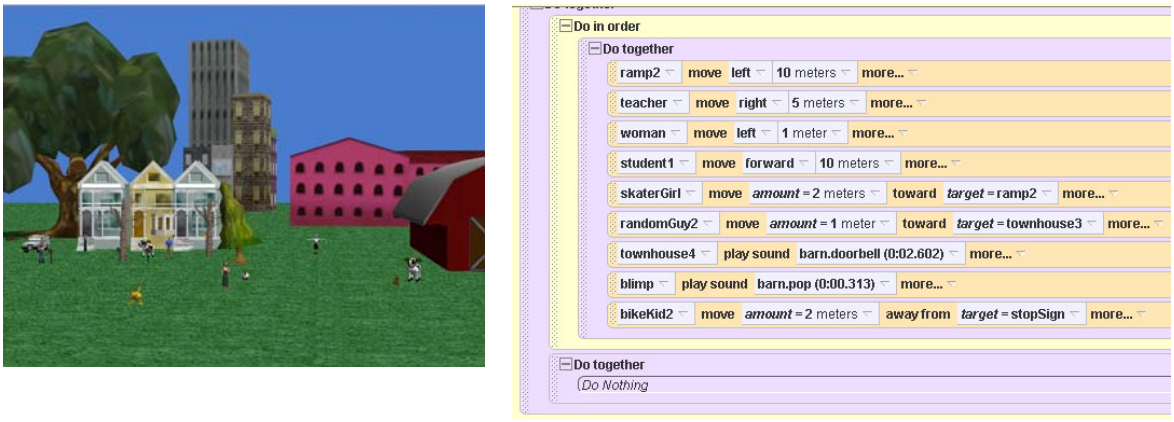


Figure 8.5: A screenshot of the world created by Fish_02_11_2006 and the program that animates it.

Fish_02_11_2006 spent a total of 79.4% of her time on scene layout and 12.5% editing her program. The world she created has 61 objects, not all of which are visible to the camera. Objects in the 3D scene include a series of street signs, several mailboxes, sports equipment, a jet, a helicopter, some beach houses, and several human characters. Of the 61 objects in the scene, only 9 have any motion associated with them. There is no apparent goal behind the animation in this world; several characters slide to new positions in parallel and two sounds play.

Fish_02_11_2006 begins her session by making several small scenes without associated programs in Alice: a beach scene, a world with objects from the amusement park, and another beach scene with a variety of objects including several beach houses, a lighthouse, a pier, and a fish. She spent approximately 20 minutes creating the scene and moving both the objects and the camera. She did not attempt to add any animations. After 20 minutes, she started a new world and added a huge variety of seemingly unrelated objects from houses to tennis rackets and a blimp. Her first programming action is to create a new method which she calls “blimp move in the air”. When users create a new method, Alice opens a new editor in which they can place the instructions for that method. Rather than defining what it means for the blimp to move, Fish_02_11_2006 dragged in the tile for “blimp move in the air” and created a recursive method call. When she played the world, nothing happened and she returned to scene layout. After adding

another set of objects, she creates another new method on the barn which she calls "open doors" and calls it recursively. When that did not work, she asked for help and I explained that she needed to define "open doors" with other commands because she was teaching Alice what it meant to for the barn to open its doors. She then deleted the recursive call and added **`move`** and **`move towards`** animations on for different 3D objects. She played her world again and nothing happened. So she added a do together and moved the lines of code into it and played it again. Again, nothing happened. So, she returned to scene layout and set up a beach scene in a new place in her 3D scene. Having been unsuccessful in getting her program to animate, she requested help again and I showed her how to call her "barn door open" method from the main method that Alice was running when the world starts. As a final step, she added a **`Do Together`** and moved her other animations into it. At the end of the session, Fish_02_11_2006 had 9 commands executing in parallel.

Fish_02_11_2006 demonstrates several of the patterns that emerge among the low programmers using Generic Alice. The low-programming participants using Generic Alice often spend a long time on scene layout before they even attempt to add any programming statements. One participant did not actually begin to program until the last five minutes of the 2 hour, 15 minute period. Based on my observations, these participants are trying to find an idea for something that they want to animate by adding and arranging 3D objects in their world. Often users add a large collection of objects and sometimes start several new worlds while searching for an idea. The shift from scene layout to programming frequently happens in one of two ways: 1) users find an idea to pursue or 2) users tire of scene layout and switch to programming because it is a new activity. It is reasonable to question why users devote large amounts of time to scene layout. Based on my observations, there are three main reasons: 1) through adding objects to their Alice worlds, users develop a confidence in their scene layout skills and are hesitant to move into an activity (programming) in which they feel less confident and 2) users forget how to animate objects in Alice and do not want to ask for help 3) users do not find the process of programming in Generic Alice appealing.

When they begin programming, not all participants are immediately successful at creating a program that causes something on screen to animate. Often programs that do not animate are caused by the user creating a new method that is not called in the program or creating a recursive method. Participants who are not immediately successful in creating a program that produces animation often return quickly to scene layout. These participants often spend a long time adding and moving objects (some create entirely new scenes) before attempting to program again. Participants who are successful in creating a program that animates sometimes begin to experiment with their programs by adding new methods and changing values. Others watch the program animate once or twice and then return to scene layout because it is more immediately satisfying. Scene layout is both easy and a potential source for ideas, so when users either do not have ideas or get frustrated by encountering programming problems, they often spend time on scene layout.

8.3.2 Low Programming: Subject Instruments_12_17_2005 Using Storytelling Alice

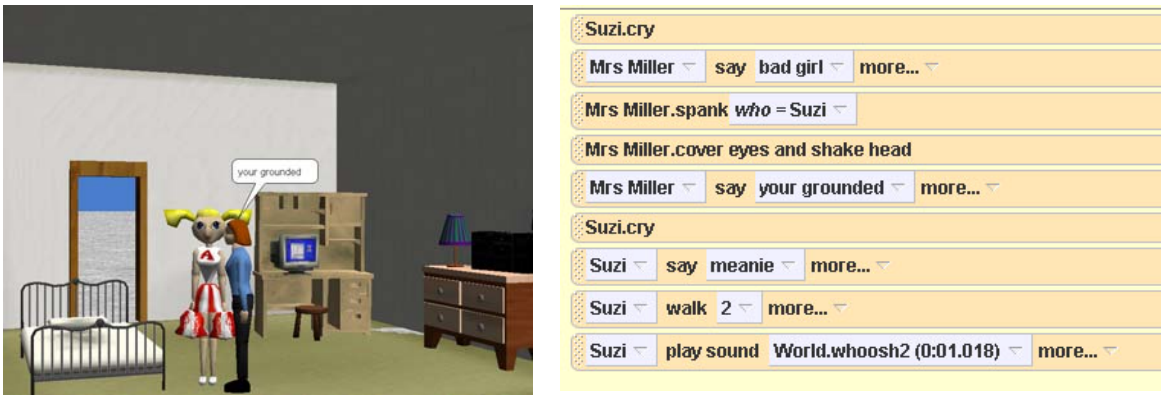


Figure 8.6: A screenshot of one of the worlds created by Instruments_12_17_2005 and the program that animates it.

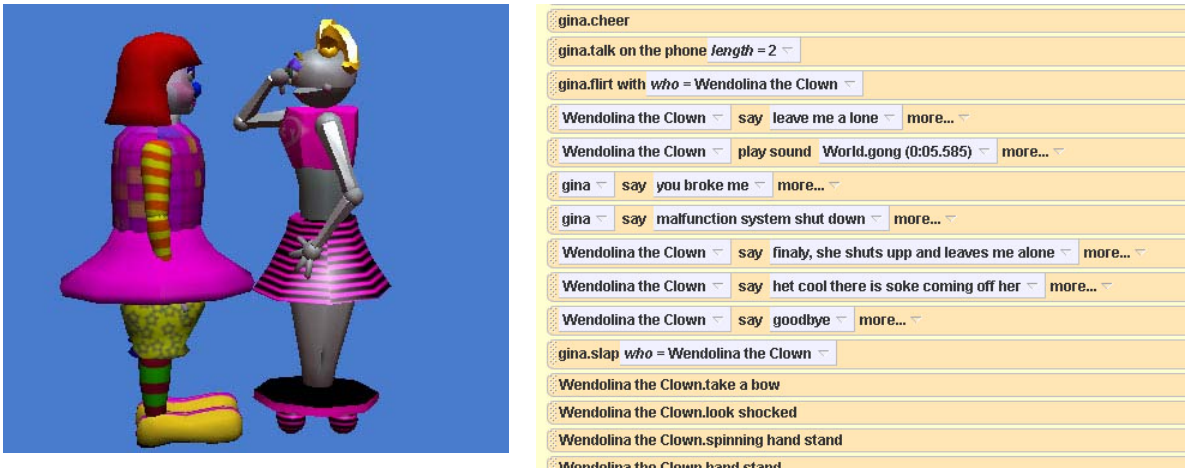


Figure 8.7: A screenshot of another of the worlds created by Instruments_12_17_2005 and a segment of the program that animates it.

Instruments_12_17_2005 spent a total of 42% of her time on scene layout and 38% on editing her programs. Rather than creating a single program, she created several short scenarios including: 1) a short scene in which a girl gets grounded (see Figure 8.6 above) 2) a short scene in which a fairy casts a spell on two kids who fall in love 3) an aquarium scene in which fish and an octopus perform a variety of animations without a clear story and 4) a scene involving a robot and clown talking (see Figure 8.7 above). None of these programs are fully realized stories. We never learn what the girl in the first program has done to deserve being grounded or see how she reacts to it, for example. Unlike the participants using Generic Alice, participants like Instruments_12_17_2005 (who did comparatively less programming than other participants using Storytelling Alice) still spent a significant amount of their time programming.

Instruments_12_17_2005 created a series of short programs, beginning with programs that were largely sequential. In one, she added girl and boy characters and, inspired by the girls' **`slap`** animation, developed a scene in which the two characters are fighting and trading insults. There is no justification given for the fight. The program she developed is purely sequential but is developed over several iterations in which she added a few lines and then played her program. The fairy and aquarium stories are also simple sequences of instructions.

In her next program, she began to explore slightly more advanced concepts. She set up a bedroom scene and added a woman and a girl (mother and daughter). In the scene, the mother is scolding the daughter, spanking her, and telling her that she is grounded. The girl character (a cheerleader) did not come with any methods that would be appropriate reactions to being grounded or spanked, so Instruments_12_17_2005 developed a cry animation over several iterations. She began by having the girl touch her face and then added a Do Together in which the girl touches her face with both hands and looks at the ground in shame. The method is called twice in her main program: once at the beginning and again after the mother has announced that she is grounded.

The following two Alice worlds are less interesting. In the first, she set up a graveyard scene but did not do any programming. In the second, she added a few characters and adds some of the methods that come with them, plays it twice and then moves on to create another world without saving.

Finally, she creates a program in which a clown is trying to get a robot character to leave by verbally attacking her. Initially, her program consists of a list of method calls. However, she spends some of the time editing the robot character's flirt method to make it play slower.

Instruments_12_17_2005, like the other low-programmers using Storytelling Alice, developed several different fairly short scenarios that used largely sequential code. Towards the end of the session, perhaps as she began to feel some mastery of sequential code, she began to branch out into slightly more advanced programming by creating her own method and editing a method that came with another character. Like the low-programming participants using Generic Alice, low-programmers using Storytelling Alice still seemed to search through the gallery for ideas. While not all of their ideas materialized into stories or even parts of stories, most added at least some animations to each of the scenes that they created. Based on my observations, the character-specific animations in Storytelling Alice entice users to program because they want to see particular animations such as one character slapping another.

One of the problems that occurred among the low programmers using Generic Alice was that it was common for users to begin programming by making a new method, and not understanding that they had to define the behavior for the that method, calling it recursively. This problem seems to occur much less frequently among the low-programmers using Storytelling Alice. I believe that this can also be attributed to the attractiveness of the animations in Storytelling Alice. Users of Storytelling Alice were more likely to experiment with animations their characters already knew rather than immediately creating new ones. Participants' early success with programming (often through calling character-specific method) makes them more likely to continue programming rather than immediately returning to scene layout.

8.3.3 Average Programming: Subject Fish_11_20_2005 Using Generic Alice

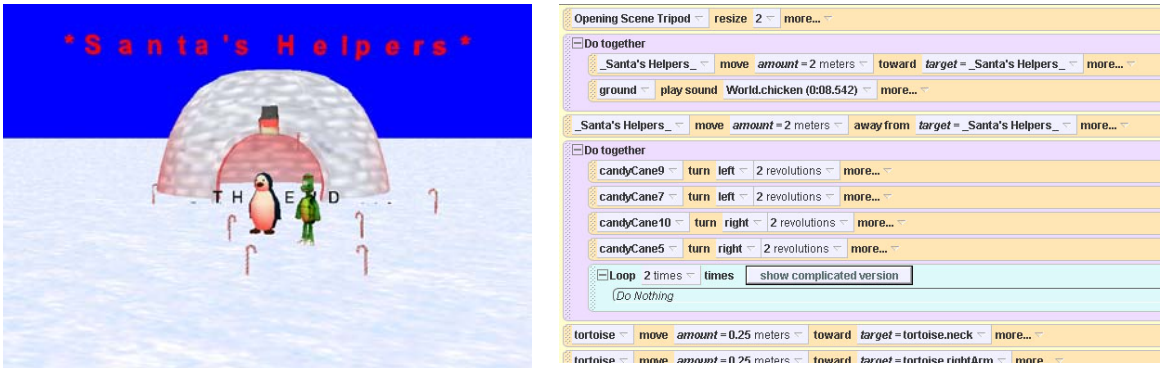


Figure 8.8: A screenshot of one of the worlds created by Fish_11_20_2005 and a segment of the program that animates it.

Fish_11_20_2005 began her session by creating and deleting several scenes as she searched for an idea. These scenes included a beach scene with beach houses and town houses, an airport scene with a runway, control tower and burning building, and an amusement park with a carousel, bumper cars, a haunted house, and swings. In the amusement park, Fish_11_20_2005 adds her first characters: an Alice Liddel (the character in Alice in Wonderland) and a Cinderella character. But, she does not move into programming. As with the previous scenes, she simply deletes all of the objects in

the world. In this last scene, she also deletes the Alice camera and the scene's light, which had the effect of causing the view of the 3D world to go black.

Fish_11_20_2005 began a new world and added a seemingly unrelated group of objects including a penguin, tortoise, road signs, skateboard, skateboarding ramp, and a candy cane. She then made several copies of the candy cane and positioned them throughout the scene. At this point she decided to turn her project into an animated Christmas e-card and added a 3D text object to title her card "Santa's Helpers" and made the text appear red (see Figure 8.8). Having declared a purpose for her project, she began editing her program. Much of her programming has a strong exploratory flavor. She began by adding animations that moved the 3D text towards the camera and rotated it. Then, she returned to scene layout and experimented with changing the sky color and adding fog. When she returned to programming, she made several of the candy canes rotate. She dragged several candy cane turns into a do together and added a loop and dragged a line of code (candy cane turn) into the loop. She changed the loop count (from once to infinity) and played the world several times. She then added additional lines of code into the loop: a move towards command and some of the character-specific methods that come with the penguin. Eventually, she removed all of the method calls from inside the loop. She ended her world by adding code that makes the tortoise and the penguin fly into the air and out of view. As a final touch, she added 3D text that says "the end".

Many of the average-programmers tend towards an exploratory programming style. There are two common programming patterns: 1) adding several of one of method (e.g. move towards or turn) and using different parameters in each method call or 2) setting up a simple program and repeatedly making a simple change and playing the world. Sometimes this exploration results in a motion that sparks an idea and helps to transition the student from an exploratory programming pattern into an intentional programming pattern in which it is clear that they have an idea that they are trying to realize. One of the average programmers in the Generic Alice case eventually developed a "dance" routine involving tortoises and hares that move and turn in a synchronized way. But, most of the programs created by average-programmers using Generic Alice never moved out of

exploratory programming. The vast majority of the animations that subjects used were on the entire object, not its body parts. Users of Storytelling Alice also used primarily animations that applied to an entire object rather than its body parts, but using the available animations within Storytelling Alice, users are more readily able to create the kinds of animations they envision.

As in the low-programming condition, more than half of the average-programmers created (but did not define) new methods for higher-level animations like having a penguin flap its wings, a character walk, or the doors on a barn open. Despite their interest in having higher-level animations for characters, the vast majority of users animated only entire objects, rather than body parts. One average programmer experimented with rotating the arms of an Eskimo character.

8.3.4 Average Programming: Subject Castle_11_5_2005 Using Storytelling Alice

Figure 8.9: A screenshot of one of the worlds created by Castle_11_5_2005 and a segment of the program that animates it.

Unlike the low-programmers using Storytelling Alice, most of the average programmers using Storytelling Alice seemed to focus their attention on creating one, larger program. Castle_11_5_2005 spent 20% of her time on scene layout and 48% on editing her program. The resulting program tells a story about a father trying to take his kids on vacation and getting lost. In the end, the son calls his mother on a cell phone to rescue them. The program contains three scenes separated into their own methods and a main method that controls the camera and lighting, and calls the methods for each individual scene. In addition, Castle_11_5_2005 created a "put hands on hips" method for Joey to

accentuate a point in the story where he gets frustrated with his father. Castle_11_5_2005 also used Do Togethers to sequence the motion of multiple characters.

Unlike previous cases, Castle_11_5_2005 appears to have arrived at a story idea almost immediately. She begins by adding Joey, Jenni and the father character and then begins programming. She begins by having the father say "Let's get the vacation started kids." The two children walk up to their father and she experiments with different durations. Using a Do Together, she ends the first scene by having the two children exit the scene together.

In the second scene, Castle_11_5_2005 adds her three characters, a pyramid, and a sphinx. When she begins to animate the second scene, she experiences some problems: 1) she mistakenly uses the characters from scene 1 in her code for scene 2 and 2) she does not change the method called to play scene 2 when the world starts. Rather than giving up and returning to scene layout, Castle_11_5_2005 asked for help to resolve these problems. Next, she turned to creating a method for Joey called "put hands on hips" She adds the two touch animations and experiments with the parameters to make it look right. It takes her sixteen iterations of changing the parameters of touch and playing the world to arrive at an animation that she is happy with. Later in the scene, she constructs a "look scared" method for Joey. She tries several different touch animations, but does not settle on one that she likes and eventually deletes the implementation of "look scared."

The third and final scene combines pre-defined character methods with dialog added by the user. She constructs it from start to finish with very little modification. Finally, she asks for help to put her scenes together. With help, she creates a method entitled "whole thing" and calls each scene. Between scenes, she adds code to move the camera and control the lights.

Having completed her original story, Castle_11_5_2005 begins a new world. In contrast to her previous world in which she seemed to start with a story, her second world seems to develop into a story over time. She begins by adding a girl, a man, and a horse. She

calls several methods of the girl's methods: kissing the horse, sitting on it, kneeling, and then straightening. When she plays her world, the girl kneels and then straightens up. The "straighten up" animation causes characters to unbend their limbs but does not move the character. Consequently, calling straighten (as opposed to stand up) after the kneel animation caused the girl to be standing in the ground. The girl standing in the ground provides the inspiration for a new story: the girl says "ouch" and Castle_11_5_2005 begins a new scene with the girl in the hospital being treated for her injury. At this point, there were only a few minutes left in the session and she returned to put a few final touches on her original world.

In both versions of Alice, users seem to expect that the method currently selected in the method editor will be the one that is executed. I believe that supporting a model of executing the selected method would allow users to explore more freely within Alice.

Overall, the average programmers using Storytelling Alice typically worked on one or two worlds over the course of the session. The average programmers using Storytelling Alice were more likely to begin with a basic concept for a story. Consequently, they spent less time adding, deleting, and positioning objects before moving into programming. However, there are still examples of "found" stories (e.g. Castle_11_5_2005's story in which Kristin gets injured) which motivate further programming. The average programmers using Storytelling Alice made frequent use of Do Togethers and most also created, defined, and used their own methods. The new methods were a mixture of scene implementations and actions for characters (e.g. put hands on hips).

8.3.5 High Programming: Subject Flamingo_12_17_2005 Using Generic Alice

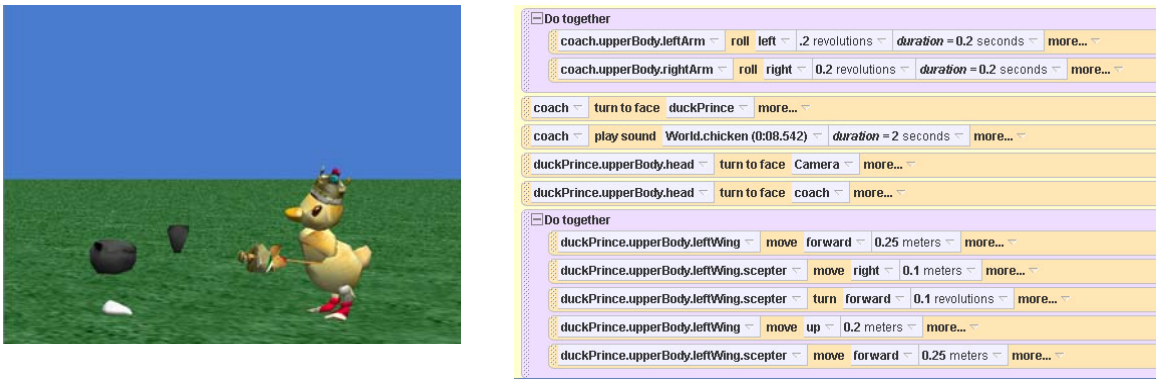


Figure 8.10: A screenshot of the world created by Flamingo_12_17_2005 and a segment of the program that animates it.

Flamingo_12_17_2005 spent a total of 16% of her time on scene layout and 61% editing her program. The world that she created shows a duck prince and a coach character. The duck prince points his scepter at the coach who explodes into pieces. Finally, the duck moves his scepter in victory. Figure 8.10 above shows Flamingo_12_17_2005's Alice program after the coach has exploded. The program includes several do Togethers and two loops, although one only executes once. Flamingo_12_17_2005 did not create any new methods.

Like many of the average-programmers using Generic Alice, Flamingo_12_17_2005 begins with a seemingly unrelated collection of characters including a tortoise, a chicken, a fence, a couch, and a ninja. The beginning of her session is typical of the average-programmers: she adds a few lines of code, plays them, and often deletes them immediately. In this initial exploration stage, Flamingo_12_17_2005 explores the move to, move towards, and turn animations using the evil ninja and the tortoise. Next, she begins experimenting with animating characters' body parts, initially with the move animation but later with turn. In contrast to many of the average-programmers using Generic Alice, Flamingo_12_17_2005 transitions from experimenting with animations to attempting to use them for a specific purpose: in this case to animate the head of a snowman falling off. There is a long segment of moving the snow man's head backwards

and then down by different amounts. Having gotten the head to move from atop the snowman's midsection to the ground, she moves all of the lines inside a Do Together.

Having completed her initial goal, Flamingo_12_17_2005 creates a series of new worlds to which she does not add more than a couple of animations. Her worlds include a taj mahal scene, an island scene with several characters, and a scene involving the duck prince and a coach character. In this final one, she creates a new method to lower the coach's arm and calls it from her main method (scene 1 method). Initially she uses **`arm.move down 1`** but, after playing it, experiments with the roll animation instead.

She begins again and re-adds the duck prince and the coach. She begins by trying to animate the coach's arms so that they hang by his side. After a few attempts, she gets a reasonable result and moves the relevant roll animations into a Do Together. She then turns her attention to animating the duck moving his scepter. The scepter does not automatically move with the duck's wing and she experiments with a combination of move and turn methods to create the appearance of the two moving together. Then she uses a series of move animations (with different directions and distances) on the coach's body parts to make the coach explode. She changes the durations of the explosion moves to .25 seconds and puts them inside a Do Together. Finally, she adds a sort of victory dance for the duck in which he moves his left wing and scepter together and then turns away from the camera and wiggles his tail. Flamingo_12_17_2005 uses a loop to have the duck wiggle his tail multiple times.

Where generally, the average programmers using Generic Alice did not typically progress from exploratory programming to intentional programming, the high-programmers all built at least one thing that exhibited some actual control and intentionality. Two of the four in the top cluster used exploding animations. Most still created and experimented with several scenes before they began to work towards a particular goal.

8.3.6 High Programming: Subject Horse_12_10_2005 Using Storytelling Alice

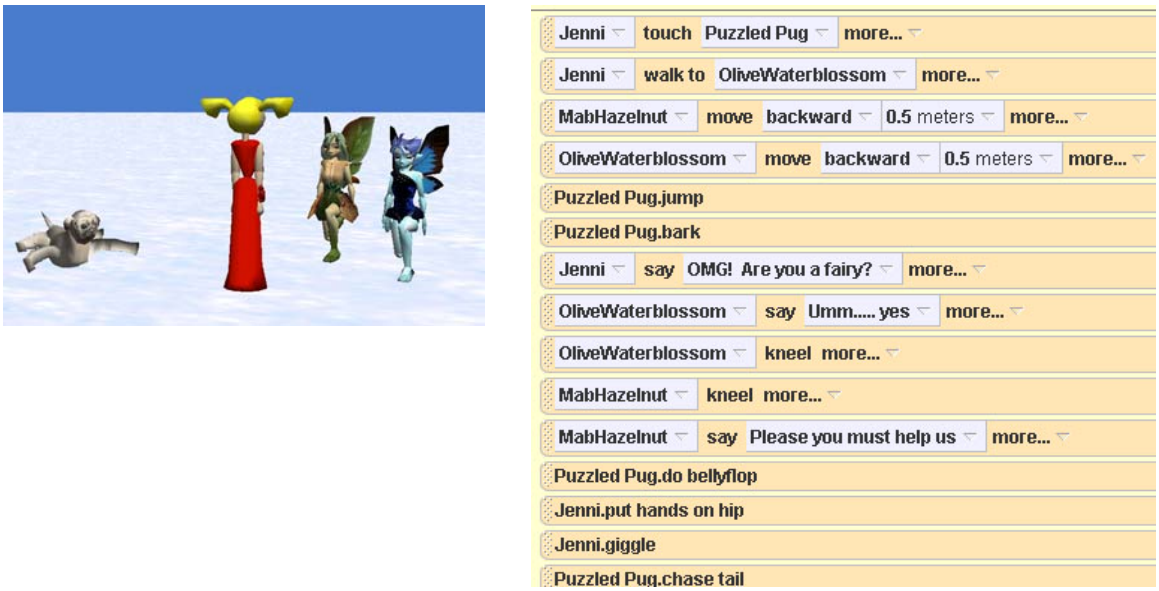


Figure 8.11: A screenshot of the world created by Horse_12_10_2005 and a segment of the program that animates it.

Horse_12_10_2005 created a story about two fairies who have lost their dog (a poodle) and request help in rescuing it from a girl, Jenni. They find the dog being held by Douglas the tree. Douglas says that if they correctly answer a riddle, he will release the fairies' poodle. Jenni eventually determines the correct answer and saves the poodle. Horse_12_10_2005's world includes two scenes (implemented in their own methods) as well as a main "my story" method which calls the two scene methods. In addition, she created a "put hands on hip" method for Jenni in which Jenni places both of her hands on her hips at the same time. The animation that she produces does not clearly show Jenni putting her hands on her hips, but Horse_12_10_2005 does not spend a lot of time trying to improve the appearance of the animation.

Horse_12_10_2005 begins by creating and implementing a "gasp" method for Jenni but deletes it without playing her program. From this point, she creates the scene from start to finish with very little revision. Horse_12_10_2005 has Jenny walk over to the faeries, the faeries kneel down, and there is a short dialog between them establishing that the faeries'

dog is missing. Horse_12_10_2005 ‘s basic pattern is to add a few lines to the end of the program, play it, and then add a few more lines.

In her second scene, Horse_12_10_2005 adds a talking tree character, a poodle, the faeries, and Jenni. She changes the method that is called when the world starts to “scene 2 method”. Like the first scene, she builds the second scene with very few revisions. However, she does change the durations of many of her say animations to make the dialog easier to read.

Like Horse_12_10_2005, all of the high-programmers using Storytelling Alice focused the majority of their time on a single story that they began to develop very early in the session. Many of the programs that they produced are dialog heavy and, while there are several with multiple scenes, they seem to do less exploration of programming concepts and constructs than the average programmers using Storytelling Alice. The stories themselves often have a drawn out feel to them. Many create the impression that the girls have continued to add to them past the point at which they had ideas they wanted to pursue.

Although purely qualitative, it seems one of the key elements associated with doing more programming is progressing from the “tinkering” stage to a stage in which users have a goal that they are trying to move towards. While some users will “tinker”, others are hesitant to begin programming until they have a story idea. Non-tinkerers without story ideas tend to add objects to their worlds until they have an idea and if that idea proves challenging they often return to scene layout.

8.4 Attitude Measures

After working with their assigned version of Alice for 2 hours and 15 minutes, participants completed a post-Alice survey which included an attitude survey and several questions about their future interest in Alice and computer science. A copy of the post-Alice survey can be founding the appendix.

8.4.1 Attitude Survey

Based on exploratory factor analysis, I created two scales from the attitude survey: 1) the entertaining scale measures how much users enjoyed working with their version of Alice and 2) the ease scale measures how easy users felt it was to use their version of Alice. Scores for all of the attitude questions ranged from 1 to 5 with 1 corresponding to "Strongly Disagree" and 5 corresponding to "Strongly Agree."

8.4.1.1 Entertaining

The entertaining scale included the following statements on the attitude survey:

1. Using the computer during the workshop today was fun.
2. Using the computer during the workshop today was interesting.
3. Using the computer during the workshop today was boring (scores for this question were reversed).
4. The computer animation program I used today is cool.
5. The computer animation program I used today is entertaining.

Cronbach's α for the entertaining scale is 0.86.

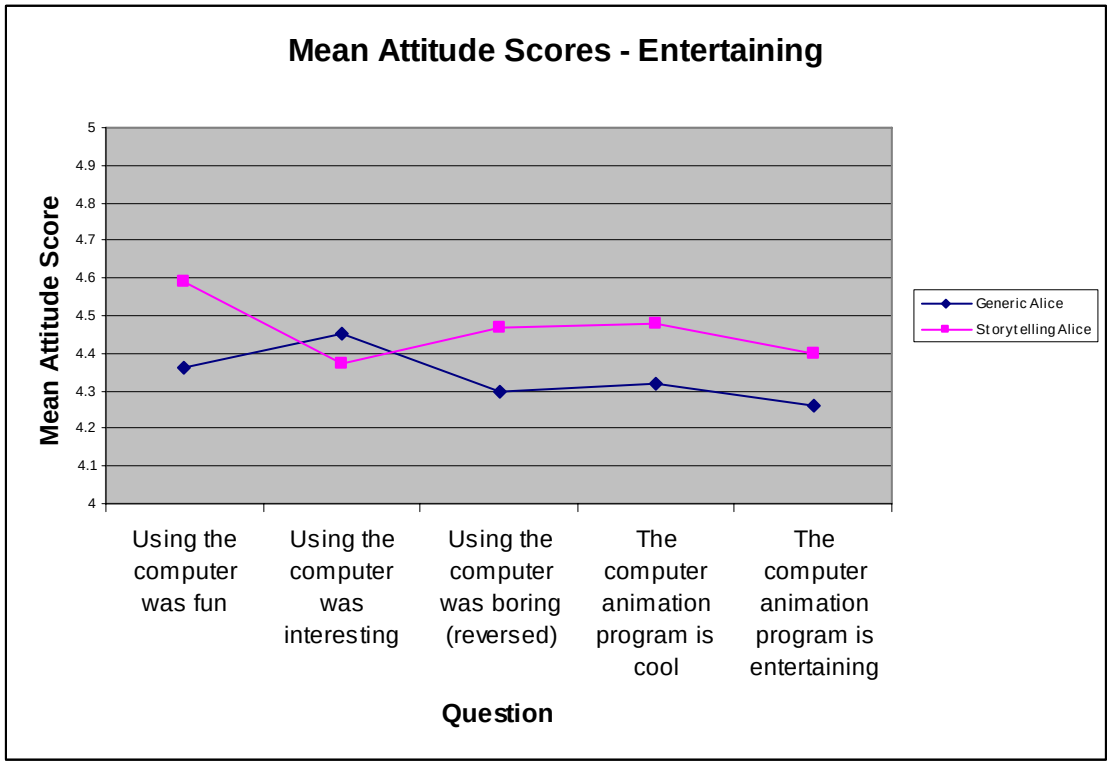


Figure 8.12: Mean scores for attitude questions in the entertaining scale.

Users of Generic Alice and Storytelling Alice did not differ significantly in how much they enjoyed using their version of Alice ($p = 0.25$). Participants in both groups enjoyed working with Alice. However, the scores for all but one of the questions in the entertaining scale were slightly higher for users of Storytelling Alice than Generic Alice. Participants with higher grades enjoyed working with either version of Alice than participants with lower grades ($p = 0.09$). Participants' enjoyment of Alice did not differ significantly based on their age ($p = .26$), computer confidence ($p=.20$), and computer usage ($p=.80$).

8.4.1.2 Ease

The ease scale included the following questions:

1. Using the computer during the workshop today was frustrating (scores for this question were reversed).

2. The computer animation program I used today is confusing (scores for this question were reversed).
3. The computer animation program I used today is annoying (scores for this question were reversed).
4. The computer animation program I used today is easy to learn.

Cronbach's α for the ease scale is 0.63.

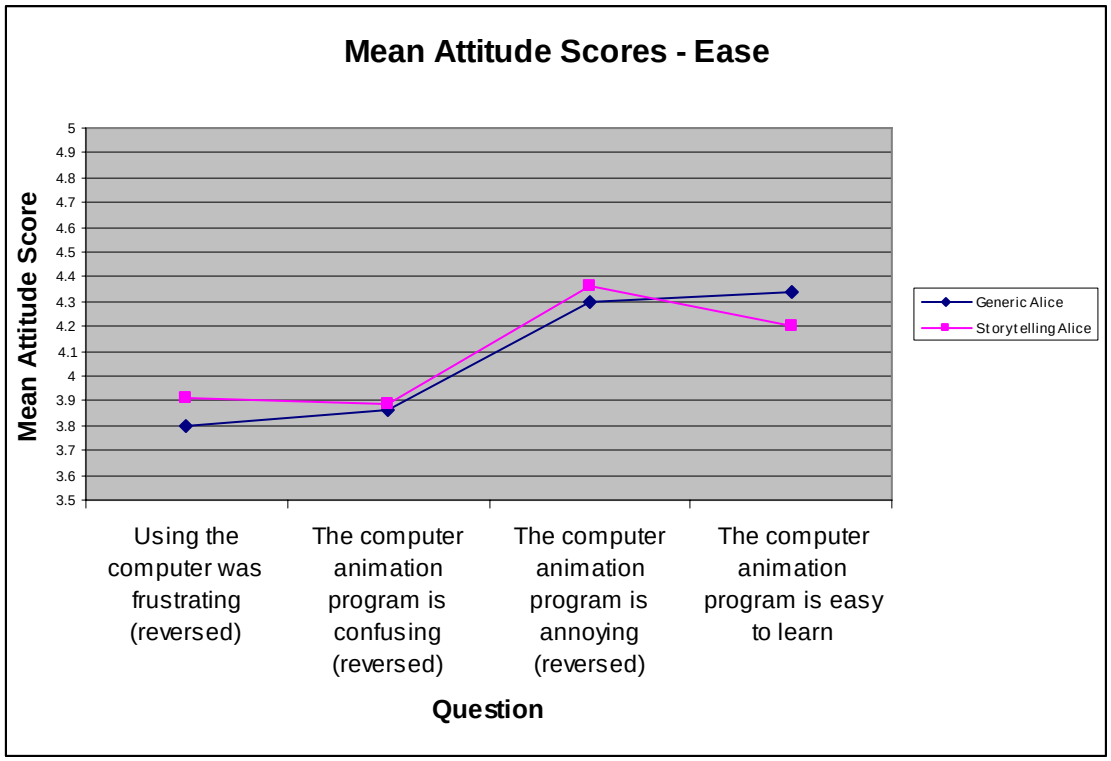


Figure 8.13: Mean scores for attitude questions in the ease scale.

Users of Generic Alice and Storytelling Alice did not differ significantly in how easy they felt it was to use their version of Alice ($p = .90$). This is not surprising since the process of creating a program in Generic Alice and Storytelling Alice is almost identical. Participants with higher grades found Alice easier to use than participants with lower grades, regardless of version ($p = .01$). Girls with higher computer confidence also found Alice easier to use, regardless of version ($p = .02$). Younger students also tended to find Alice easier to use than older students, regardless of version ($p = .08$). It is surprising that younger students find Alice easier to use than older students. One potential explanation

for this finding is that girls' academic confidence tends to drop during middle school (AAUW 1996). Younger girls may tend to have higher overall confidence in themselves than older girls and therefore find the process of learning Alice easier. Girls' perception of Alice's ease did not differ significantly based on their computer usage ($p = .21$).

However, despite the fact that participants who used Storytelling Alice and Generic Alice had statistically similar attitudes, participants who used Storytelling Alice were more likely to be interested in taking a future Alice class. One potential explanation for users of Generic Alice being less interested in future Alice classes is that the primary draw in using Generic Alice is the self-expression that comes from creating virtual worlds by selecting and arranging objects. After working with Alice for a relatively short period of time, users typically master the process of finding and arranging objects and may feel that they can continue without further instruction. In contrast, most of the potential for self-expression in Storytelling Alice comes through programming, which is considerably more complex. Consequently, users of Storytelling Alice may feel that additional instruction would help them to better express themselves through Alice programs.

Storytelling seems to have very little impact on participants' confidence in their ability to learn either more advanced concepts in Alice or a general-purpose programming language like Java or C++. This is not surprising given that process of programming in both Storytelling Alice and Generic Alice is the same. The storytelling focus may help to place computer programming in a motivating context but it does not make programming easier.

In conducting the Alice workshops, I noticed that girls often did not see a strong connection between creating stories or animations in Alice and the discipline of computer science as a whole. In the short term, the fact that girls do not immediately connect the process of creating stories in Alice with computer science may make them more receptive to learning how to program computers using Alice. However, in the long term, it will be necessary to find ways to help girls understand the connection between writing Alice

programs and the discipline of computer science if we hope to inspire more girls to study computer science.

8.4.2 Additional Survey Questions

Based on exploratory factor analysis, I created two scales based on the additional survey questions: 1) the future alice use scale is a measure of participants' interest in continuing to use their version of Alice in the future 2) the computer science interest scale is a measure of participants' interest in pursuing computer science.

8.4.2.1 Future Alice Use

The future alice use scale included the following questions:

1. If you used Alice (the computer animation program you used today) again, how long do you think you could use it at one time without getting bored? (Scores ranged from 1 or "Less than 1 hour" to 5 or "More than 4 hours").
2. If you had the computer animation program you used today ("Alice") on a computer at home, how often during the next month do you think you would use it? (Scores ranged from 1 or "Never" to 5 or "More than once a week during the next month").
3. Would you be interested in taking another Alice class? (Scores ranged from 1 or "Definitely Not" to 5 or "Definitely Yes").
4. Do you think you could create a world in Alice that you would be proud to show your friends? (Scores ranged from 1 or "Definitely Not" to 5 or "Definitely Yes").

Cronbach's α for the future Alice use scale is 0.83.

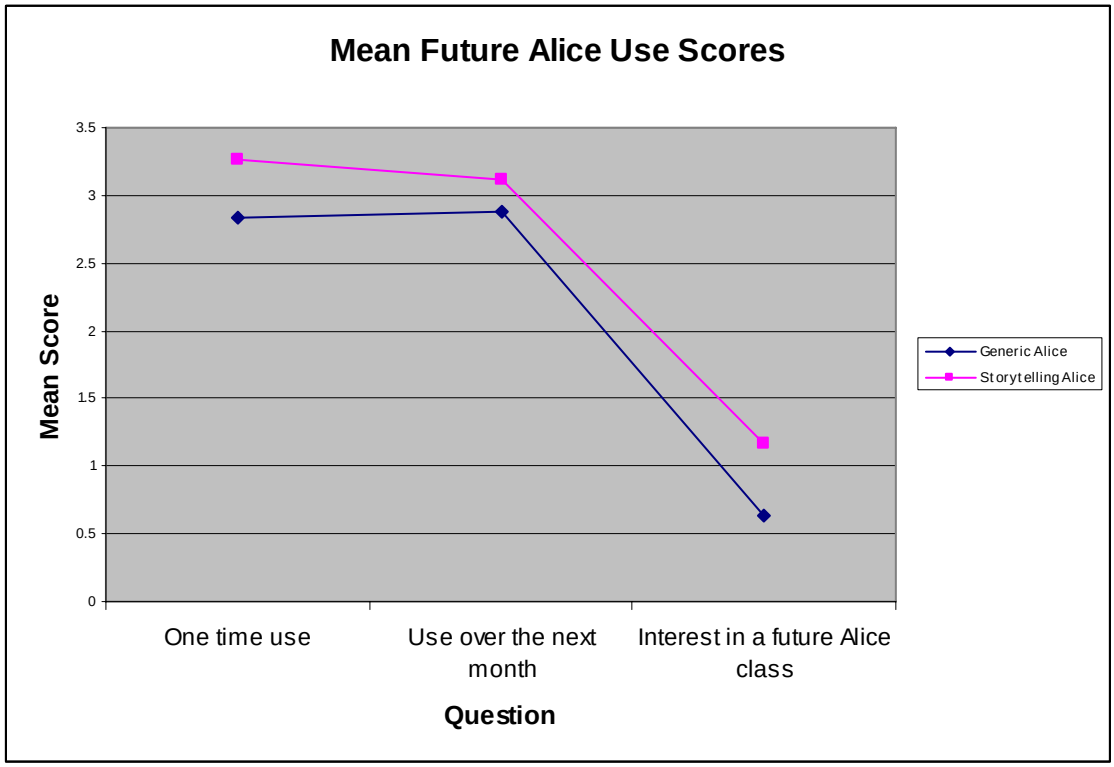


Figure 8.14: Mean scores for questions in the future Alice use scale.

Participants who used Storytelling had a stronger interest in continuing to use Alice in the future than those who used Generic Alice ($p = 0.05$). One potential explanation for this is that girls using Storytelling Alice may have felt that it had greater "replayability." Where users of Generic Alice often made worlds that seemed to consist of arbitrary motion (see Chapter 10), users of Storytelling Alice most often created stories. More of the users of Generic Alice may have felt that they had exhausted the interesting aspects of interacting with Generic Alice. Grades ($p = .22$) and past computer use ($p = .69$) were not significant predictors of future interest in Alice.

8.4.3 Computer Science Interest

The computer science interest scale included the following questions:

1. Do you think you could create a world in Alice that you would be proud to show your friends? (Scores ranged from 1 or "Definitely Not" to 5 or "Definitely Yes").
2. Do you think you could learn to use advanced features in the Alice program? (Scores ranged from 1 or "Definitely Not" to 5 or "Definitely Yes").

3. Do you think you could learn a computer language like Java or C++? (Scores ranged from 1 or "Definitely Not" to 5 or "Definitely Yes").
4. Would you be interested in taking a computer science class in high school? (Scores ranged from 1 or "Definitely Not" to 5 or "Definitely Yes").
5. Can you imagine growing up to be a computer scientist? (Scores ranged from 1 or "Definitely Not" to 5 or "Definitely Yes").

Cronbach's α for the computer science interest scale is 0.80.

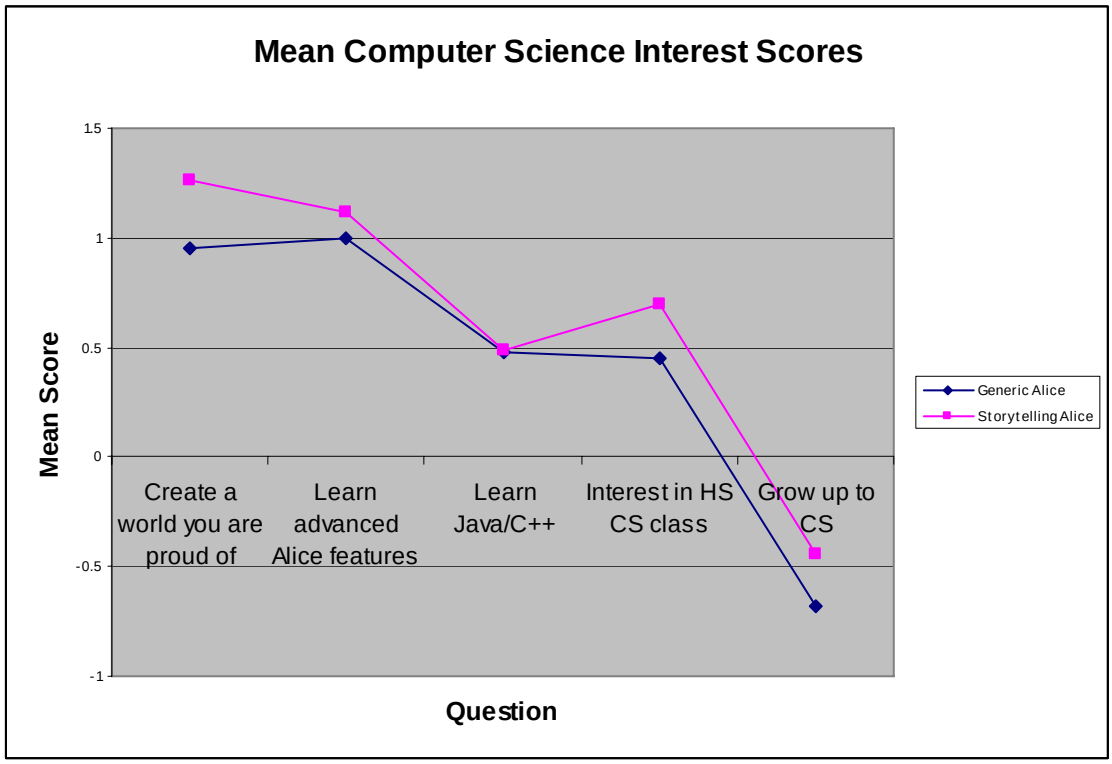


Figure 8.15: Mean scores for questions in the computer science interest scale.

A single four-hour workshop is a fairly short period of time in which to change students' interest in pursuing computer science. Not surprisingly, there was no significant difference in interest in pursuing computer science between users of Generic Alice and Storytelling Alice ($p = .33$), although users of Storytelling Alice expressed slightly higher interest on most questions. However there is a strong relationship between participants' interest in using Alice in the future and their interest in pursuing Computer Science ($r = .54$, $p < .0001$).The strongest predictors of future interest in computer science were strong

academic performance ($p = .006$) and high confidence with computers ($p < 0.001$). Age ($p = .19$), and computer usage ($p = 0.2$) were not significant predictors of future interest in computer science.

8.5 Programming Quiz Performance

After completing the survey, participants were asked to complete a 7 question programming quiz which presented short segments of code in Alice and asked participants to select the most appropriate description the behavior of the code from a list of four choices. Questions covered sequential code, parallel code (i.e. do togethers), loops, methods calls, and method calls with parameters.

Based on exploratory factor analysis, the quiz has two factors: 1) programming structures and 2) events.

The programming structures scale included a question each about:

1. sequential execution
2. parallel execution
3. simple loops
4. more complex loops
5. method calls
6. method calls with parameters

Cronbach's α for the programming scale is 0.74.

There was a single question that asked users to predict which method Alice would play given an image of the user interface.

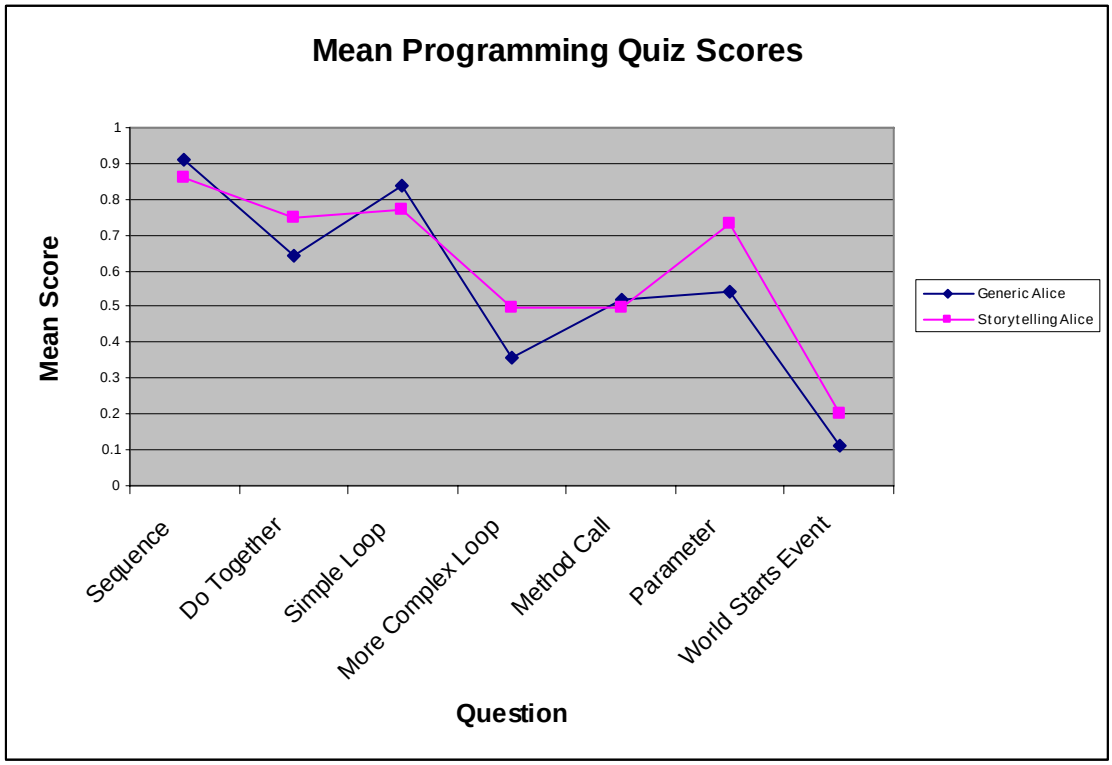


Figure 8.16: Mean Scores on the Programming Quiz for users of Generic Alice and Storytelling Alice.

Since users of Storytelling Alice spent more time on programming, we might expect to see better performance on their post-Alice programming quizzes on either the programming structures scale (p=0.44) or the events questions (p=.25) Yet, users of Storytelling Alice did not perform statistically better than those who used Generic Alice. One possible explanation is that creating stories is inherently less rich than creating animations in Generic Alice and the additional time on programming was offset by the lesser value of programming stories as opposed to animations. There were participants using both Storytelling Alice and Generic Alice who exhibited patterns that may not be as conducive to learning programming as others. In Generic Alice, it was fairly common for users to create "totally random" worlds that exhibit absolutely no control. One might argue that this is a tinkering-based learning style. Some participants did use random experimentation as a learning tool. However, others appeared to simply try random methods until something visually interesting happened without trying to understand or control the behavior of the methods they called.

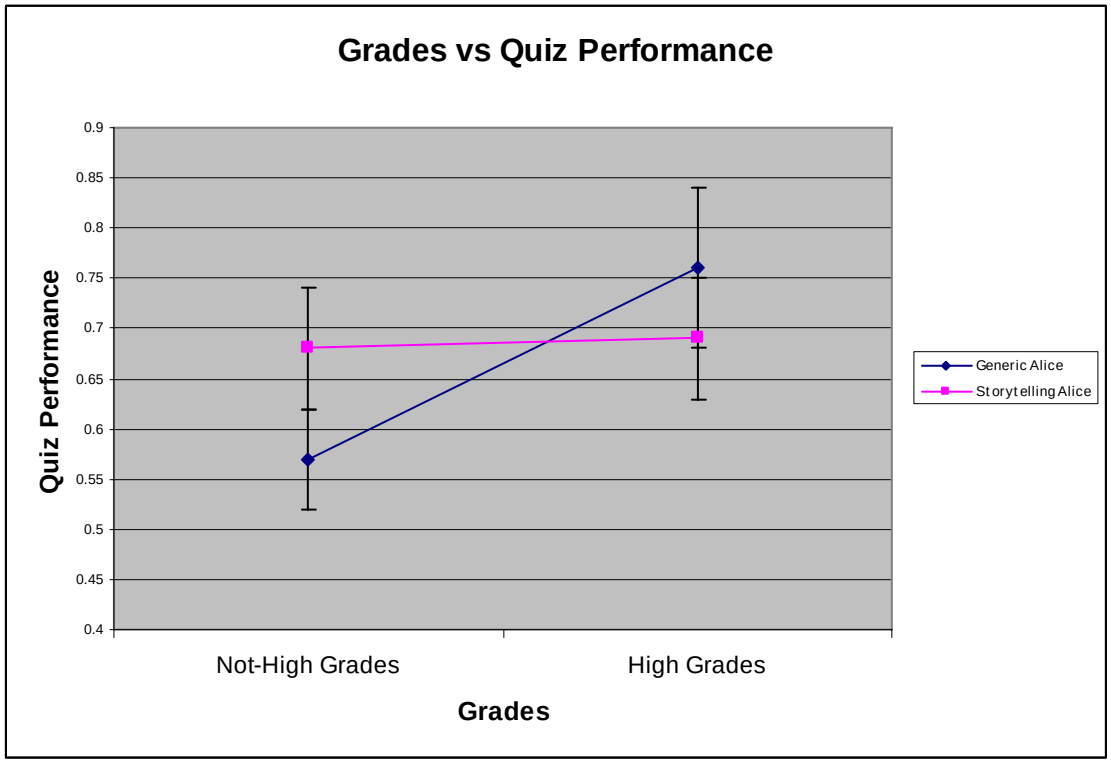


Figure 8.17: Grades vs. Quiz performance for users of Generic Alice and Storytelling Alice.

Storytelling Alice had a greater benefit for students without high grades (defined to be students who receive A's and B's or below). During formative testing, I noticed that less bright students seemed more likely to get frustrated creating programs in Alice 2 and return to scene layout. Storytelling Alice may make the process of programming more motivating for these students, motivating them to spend more time programming. Among those with high grades, users may have devoted their intellectual energy to improving the story which, depending on their story goals, may not always coincide with learning new programming concepts.

The strongest predictor of programming quiz performance on the programming structures questions was academic performance. Students with higher grades tended to perform better on the programming quiz than students with lower grades (p=.06). Computer confidence (p=.58), computer usage (p=.22) and age (p=.74) were not significant predictors of programming quiz performance.

The strongest predictors of events performance were strong academic performance (p=0.03) and computer confidence (p=.01). Computer usage (p=.78) and age (p=.24) were not significant predictors of programming quiz performance.

8.6 End of Workshop

At the end of the workshop, I gave girls 30 minutes to try the version of Alice to which they were not assigned (i.e. Storytelling Alice participants tried Generic Alice and vice versa). After trying the other version, they were asked to choose one version of Alice to take home. To close the session, I asked participants to choose a single world to show everyone. Participants were given time to watch each others' Alice worlds.

8.6.1 Choosing Storytelling Alice or Generic Alice

Table 8.2: Participants' choices of which Alice version to take home.

Alice Version	Chose Generic Alice?	Chose Storytelling Alice?	p-value
Generic Alice	26.70%	73.30%	
Storytelling Alice	11.60%	88.40%	p < 0.001

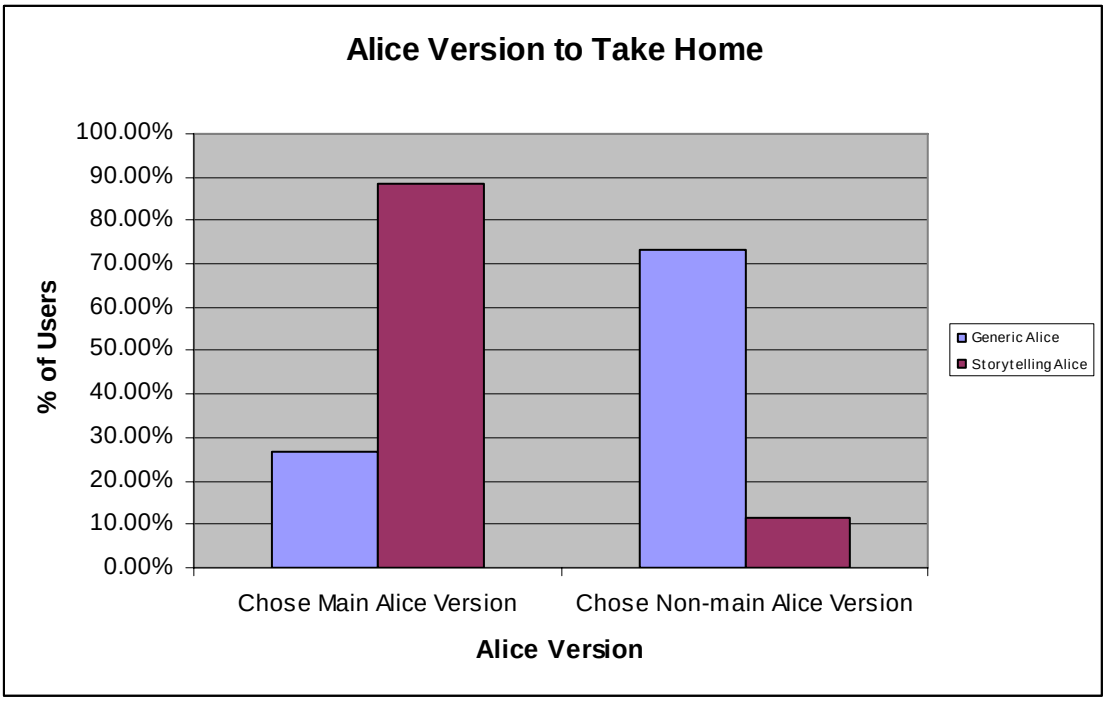


Figure 8.18: Participants choices of which version of Alice to take home.

Because participants had 2 hours and 15 minutes with their main version of Alice and only 30 minutes with the other version, it is reasonable to expect that participants would tend to choose the version with which they had the most experience. In fact, both groups showed a significant preference for Storytelling Alice ($p < 0.001$) based on a chi-squared test. Nearly three quarters of the Generic Alice participants elected to take Storytelling Alice home and more than 88% of the Storytelling Alice participants also elected to take Storytelling Alice home. Where 73% of the Generic Alice participants chose to take the version of Alice with which they had only 30 minutes, only 11.6% of the Storytelling Alice participants chose their 30-minute version.

In three cases, there were siblings (totaling six subjects) in the testing groups who colluded to ensure that they had both versions of the system at home. If I remove these pairs from the data, the preference towards Storytelling Alice becomes slightly stronger: 76.2% of the Generic Alice users choose to take home Storytelling Alice and 10% of the Storytelling Alice users choose to take home Generic Alice. The fact that participants who used both Generic Alice and Storytelling Alice overwhelmingly chose Storytelling Alice as the system they wanted to take home demonstrates that Storytelling Alice has a stronger appeal than Generic Alice for most girls.

8.6.2 Showing a World

Table 8.3: Participants choices about what to show.

Alice Version	Show World from Non-Main Version	Show Main World without Changes	Show Main World with Changes	p-value
Generic Alice	32%	52%	16%	
Storytelling Alice	2%	47%	51%	p<0.001

As with the choice of which system to take home, it is reasonable to expect that girls would primarily choose to show the world that they had the longest period of time to create. In this case, we do see a tendency in that direction: 68% of the participants using Generic Alice and 98% of the participants using Storytelling Alice showed a world from their assigned version of Alice. However, a surprising 32% of the Generic Alice participants chose to show a world that they created in Storytelling Alice in 30 minutes rather than the world they had approximately 90 minutes to create in Generic Alice.

Participants using Storytelling Alice were more than three times as likely to sneak a few extra minutes to make final changes to their Alice programs. This tendency to sneak extra time is another indication of girls' engagement with programming using Storytelling Alice. While participants were preparing to share their Alice programs with other members of their Girl Scout troop, there was a period of several minutes during which participants could make final changes to their Alice programs, but there was no expectation that they should do so; their instructions were to load the Alice program they wanted to share. During this period, some participants took extra time to make final changes to their Alice programs. Among the users of Generic Alice, 16% of participants made changes to their Alice program before sharing it. Among the users of Storytelling, 51% of users made final changes to their Alice program before sharing it.

8.7 Summary

The results of the summative evaluation of Storytelling Alice demonstrate that storytelling is promising approach for introducing girls to computer programming. However, there is still room for improvement in introducing middle school girls to computer programming.

Participants who used Storytelling Alice spent more of their time on programming and less time on scene layout than participants who used Generic Alice ($p < 0.001$).

Participants who used Storytelling Alice expressed a stronger interest in using Alice in the future ($p < 0.03$ when participants' computer confidence is considered).

Nearly three times as many participants who used Storytelling Alice show motivation to work on their programs (as measured by the numbers who sneak extra time to continue working) as participants who used Generic Alice ($p < 0.001$).

88.4% of Storytelling Alice users and 73.3% of Generic Alice users chose to take Storytelling Alice as the version of Alice they wanted to take home ($p < 0.001$).

Users of Storytelling Alice and Generic Alice expressed statistically similar on scales measuring the entertainment and ease of their version of Alice.

Users of Storytelling Alice and Generic Alice also performed similarly on a post-Alice programming quiz.

Users of Storytelling Alice and Generic Alice expressed statistically similar interest in pursuing computer science in the future.

Chapter 9 What Girls Create

9.1 Introduction

In this chapter, I describe and provide examples of the kinds of projects that girls created in both Generic and Storytelling Alice. As they were used within the Alice workshops, both Generic Alice and Storytelling Alice are a medium for self-expression. By examining the projects girls create in Generic Alice and Storytelling Alice, we gain some insight into the benefits that creating Alice programs provide to girls (beyond gaining experience and confidence with computer programming).

9.2 Generic Alice Worlds

Users begin creating programs in Generic Alice by adding 3D objects to their virtual world. Often, users are drawn to character-like 3D objects like people and animals. Having selected people and animals, users naturally want to animate their 3D objects in the ways that people and animals move: a user may want a ballerina to dance, humans to walk, and bunnies to hop. But the animations in Generic Alice do not easily map to the kinds of actions that users want their characters to perform, so users naturally begin to experiment with the animations available in Generic Alice. Users employ different strategies in exploring the Generic Alice animations. Some of these strategies are more likely than others to enable the user to develop an understanding of how to construct the motions they envision out of the animations Generic Alice provides. Among the worlds

that girls who used Generic Alice created, there are two basic groups: worlds based on arbitrary motion and worlds that exhibit some intentionality.

Fundamentally, intentionality is a subjective measure since I cannot know exactly what was going through participants' minds as they worked on their programs. In deciding whether or not a program exhibited intentionality I asked myself how one would describe a given program. If I could come up with a description for all or part of the program's animation that was more detailed than "objects are moving in space", I declared that it had at least some intentionality. I present these as qualitative results intended to provide insight into what girls choose to create. To strengthen the results, one could have several external viewers rate the intentionality of the programs that girls created.

9.2.1 Arbitrary Motion

When users cannot immediately see how to create the animations they envision their characters performing, many of them begin to add calls to arbitrary animations and play their programs to see what happens. While some users progress from exploratory programming to creating animations that demonstrate some control over the animations they are creating, 62% or 28 of the 45 worlds that users created with Generic Alice consist of seemingly arbitrary animation; characters and/or their body parts move around the screen without any coherence or clear purpose. While it is possible that a few of the users envisioned animations of pseudo-random motion, it was clear from observing girls interacting with Alice that many of them were at a loss to create any specific animation. Instead, these users added an assortment of method calls and then played their programs hoping to stumble on visually interesting animations. Figure 9.1 shows an example screen shot from one of the arbitrary motion worlds. In this world, characters and their body parts rotate around different axes and fly to different positions in space.

Figure 9.1: Flamingo_01_28_2006 created a world with a random collection of characters that move in arbitrary ways. Sometimes the whole character moves, sometimes only a part (like a leg or arm) is animated.

9.2.2 Intentional Motion (17 worlds)

38% or 17 of the 45 worlds created by Generic Alice users demonstrate that they had developed and used intentional control in specifying the motions of their characters.

9.2.2.1 Intentional Motions

15% or 7 of the 45 users created programs which contained one or two simple character motions but were otherwise largely arbitrary motion. Examples of character motions include having a cat swish its tail, a penguin open and close its mouth, and a bunny jump up and down. These worlds seem to the result of users transitioning from experimenting with the Generic Alice animations to exerting control over them in the service of creating small character motions.

Figure 9.2: Fish_10_01_2005 made an animation involving characters standing in front of houses. The characters put their arms by their sides and the girl on the left waves hello.

9.2.2.1.1 Intentional Sequences

There were 10 worlds which longer sequences of animations (as opposed to individual motions) that had a clear purpose. 7 were short story-like sequences and 3 contained characters performing choreographed dance routines. None of the users attempted to create games or interactive pieces. However, the tutorial focused on teaching the skills necessary to create non-interactive worlds rather than interactive worlds.

9.2.2.1.2 Story-like Sequences

There were 7 short story-like sequences. Most of the action in these story-like sequences is motivated by the existence of a clear aggressor (e.g a mummy, dragon, etc) and the other characters' reactions to that aggressor. In two animations, people run away from mummies. Another user created a program in which a magical duck casts a spell on a man whose body explodes by flying into pieces. In most of the story-like sequences, users rely on the physical appearance of characters to identify them as heroes, villains, or victims. Most also incorporate simple gestures such as having character raise their arms in fear before sliding quickly away or a dragon rotating sideways after being stabbed by a knight that help to communicate the action in the story.

Figure 9.3: Sailboat_12_10_2005 created a story-like animation in which a penguin moves to the lever, the lever turns, and the Christmas tree lights come on.

Figure 9.4: Lighthouse_12_3_2005 created a story-like animation in which a knight slays a dragon and the princess declares the knight her hero.

9.2.2.1.3 Choreographed Dance Routines

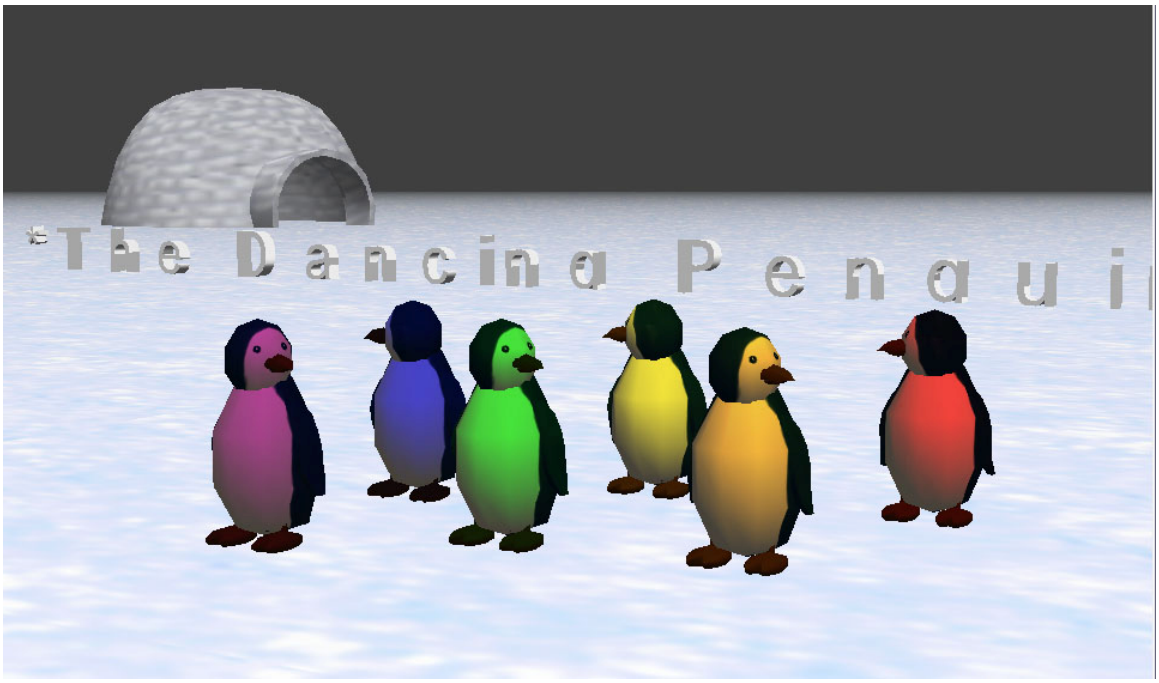


Figure 9.5: Lighthouse_01_14_2006 created a dancing penguins animation in which the penguins turn, jump, and look different directions in sequence and in parallel.

Three worlds took the form of choreographed dance routines for a group of characters. The dance routines made heavy use of move and turn with characters performing the same motions together and in sequence. Dance routine-type programs are a good match for Generic Alice. Users do not need to communicate a story, several characters doing simple motions in sequence or in parallel often produce a pleasing result, and the dancers themselves often only move within a constrained space which helps users avoid the trial and error that often accompanies trying to get characters to move to a particular location (e.g. into the igloo) using animations like moving a distance and turning a number of revolutions. It is interesting to note that two of the three dance-based animations incorporated penguin characters that came with their own higher-level methods. Using the methods that came with the penguin objects made it possible for girls to make somewhat more interesting looking dances. The two girls who used the penguins combined the penguins' character-methods like jumping, walking forward, and looking left and right with simple move and turn animations. The author of one of the two penguin dances also included a fan dancer who waved her fans and swayed at the waist.

In looking at the programs created with Generic Alice, it is striking that more than 60% of users never moved beyond experimentation to intentional control. Girls often struggle with using the low-level animations in Generic Alice to build up interesting animations. Further, in Generic Alice users can only communicate a story through the body motions of their characters because Generic Alice lacks a **`say`** method; essentially they can create a silent film. It is difficult for directors to create compelling silent films even with the help of trained human actors. Creating silent films by individually controlling the motions of each joint for each character is still more difficult.

A small number of the girls using Generic Alice added 3D-text objects to their worlds to help communicate information about their animations. Several programs incorporated 3D-text as titles (like the "The Dancing Penguins" in Figure 9.5) or to turn animations including winter-themed objects into Christmas e-cards. A few programs also incorporated 3D-text to enable characters to speak. For example, in the world shown in Figure 9.4, 3D-text saying "My hero!!!!!" flies forward and over the princess's head after she is rescued. However, since each 3D-text phrase has to be explicitly placed within the world and managed, the process of creating an elaborate dialog is too complex for most novice Alice users.

9.3 Storytelling Alice Worlds

All of the users of Storytelling Alice created programs that exhibit some intentionality. This is likely attributable to two factors: 1) the characters perform higher level actions like walking and sitting that more readily match the kinds of things kids envision their characters doing 2) the existence of the say and think animations makes it much easier for kids to communicate what is going on. For example, girls can have their characters say "I am really sad" even if they cannot communicate the character's emotional state through animation.

The programs created with Storytelling Alice are almost exclusively stories which fall into 3 broad groups: relationship stories (51% or 22 of 43 programs), good vs. evil stories (21% or 9 of 43 programs), and other (28% or 12 of 43 programs).

9.3.1 Relationship Stories

The single largest group of stories that users of Storytelling Alice wrote involved relationships. Girls seemed to use the stories that they wrote to explore issues in their own lives. However, many of the stories were "larger than life" in the sense that they portrayed both scenarios and character's choices that seem beyond the scope of what most of the story-authors would be likely to encounter in real life (based on my admittedly limited knowledge of them as individuals). For example, one story about divorcing parents depicted the children kicking the parents out of the house. There were three main groups of these: romantic relationships, peer relationships, and familial relationships.

9.3.1.1 Romantic Relationship Stories: 12 programs

Figure 9.6: Dress_01_14_2006 wrote a story involving a guy named Dave who has been having relationships with three different girls. They find out and kick his legs off in retaliation. The story ends with the statement "And thats why you dont cheat on girls!!! It Makes Your Legs Fall Off!!!"

Relationship stories ranged dealt with issues such as jealousy between two girls who liked the same boy, a girl rejecting a suitor but returning to him after she is rejected by someone else, and the embarrassment that comes from friends' public commentaries

about your relationships. The stories depict a variety of scenarios ranging from relationships among middle or high-school aged characters to stories that seem to almost be commentaries on relationships. One program shows a series of boys asking girls out and the girls rejecting them in different ways. Another program shows a girl and boy swimming together and kissing at the base of a waterfall. A third program shows two ogres getting married and pledging eternal love only to declare thief dislike for each other in the next scene.

9.3.1.2 Peer Relationships: 6 programs

Figure 9.7: Horse_01_28_2006 wrote a story which begins by showing the title "There was a boy, named Leon, that was a inflexible, unchanging, bully! And this is what happened to him…" During the course of the story, the nerd character sees the tree wave and Leon responds by taunting him. But, the tree begins talking to Leon and he sees the error of his ways and promises not to further pick on the nerd character.

Some of the worlds depicted peer relationships. One shows the brittle nature of friendships among adolescents: Two girls are hanging out together and one offers to make a cool weird noise. Her friend's response is to say "You're weird" and leave. The noise-making girl is distraught by her friend's rejection. Two stories included unlikable

characters (a spider and a large, scary dog) who lament their lack of friends when people run from them in fear. Others programs show peers taunting and arguing with each other.

9.3.1.3 Familial Relationship Stories: 4 programs

Familial stories range from innocent depictions of imperfect parents to stories involving painful subjects like divorce. On the innocent end of the spectrum, one girl created a story about a father who takes his reluctant children on vacation, gets thoroughly lost and somehow they all end up in Egypt. The son calls his mother using his cell phone to request that she come to rescue them. In one of the more serious stories, two siblings are fighting and the older sister maliciously tells the younger sister that she (the younger sister) was adopted. In another, the parents are divorcing and the kids decide they do not want to live with either parent so they kick both parents out of the house.

Figure 9.8: Castle_11_5_2005 wrote a story in which a father and his two children get lost while on vacation and mom has to come and rescue them.

In general, girls' relationship stories tend to focus on difficult situations. In part, this is probably due to the fact that girls are using the activity of creating stories in Alice as a way to think through issues and situations they are facing. However, the focus on difficult situations may also be a consequence of focusing on stories: stories need a conflict to be

interesting. It is more difficult to create a satisfying story that centers on a happy scenario than a sad one.

9.3.2 Good vs. Evil: 9 programs

The second-largest category of were conflicts between good and evil. The conflict between good and evil provides an easy source of tension, and one that is frequently used in mainstream movies and books. In the good vs. evil stories created by girls using Storytelling Alice, violence or the threat of violence were often (but not always) employed as a way to resolve conflicts. This provides additional support for Laurel's claim that girls do not object to the violence in video games as much as they object to the lack of strong stories (Laurel 2001). Other research has found that when girls design their own video games they are less likely than boys to provide violent feedback when a player does not successfully complete a level (Kafai 1995).

Figure 9.9: Castle_12_07_2005 created a story in which the wolf comes and attempts to befriend the three pigs in hopes of eating them later. The pigs get scared of the wolf and a ninja appears to frighten the wolf away.

In some stories evil characters are disciplined by more powerful good characters. For example, in one story, an evil samurai attacks an innocent pig. A good magical tree comes to the pig's defense and resurrects the pig, enabling the pig to attack the samurai in retaliation. In other stories, the evil characters seem to triumph. In one such story an evil sheriff wants to take over the world. When his minion expresses doubts, the sheriff disciplines him by picking him up and tossing him across the room.

9.3.3 Other Alice Programs: 12 programs

The remaining worlds created with Storytelling Alice do not fall neatly into a single category. These miscellaneous worlds include two stories about finding lost dogs, two stories depicting running and swimming races, and three choreographed routines (circus and cheerleading). These last three are similar in nature to the choreographed dance routines created by users of Generic Alice.

Nearly all of the users of Storytelling Alice made stories (with the exception of the 3 choreographed routines). Further, all of the users of Storytelling Alice (as compared to 38% of the users of Generic Alice) moved from experimental programming into intentional programming. In a sense, Storytelling Alice helps to minimize the time users take to figure out what they should do with the system, how to use the tools the system provides, and begin actually working towards a goal. Although it is possible to use assignments to provide students with a goal, this approach may be significantly less motivating for students.

There are benefits to constructing animated stories that go beyond girls gaining experience with and confidence in their abilities to program a computer. Storytelling provides girls with an opportunity to think about and role play through or simply vent about issues they are facing in their own lives. Additionally, storytelling is a form of communication. When girls show their stories to a peer, the peer often comments about aspects of the story that they do not understand which provides a natural motivation for girls to revise their stories.

Chapter 10 Conclusions and Future Work

10.1 Conclusions

The results of my study suggest that participants who used Generic Alice and Storytelling Alice were equally successful in learning programming concepts. However, I found that participants who used Storytelling Alice showed more evidence of engagement with programming; they spent a 42% more time programming instead of devoting time to non-programming activities within Alice and expressed greater interest in future use of Alice than participants who used Generic Alice. With less than 0.5% (Vegso 2005) of women entering college intending to major in computer science, it is encouraging that Storytelling Alice motivates more than half of middle school girls to sneak extra time to program. These results suggest that the storytelling approach is highly promising for attracting more middle school girls into computer science.

Although participants who used Storytelling Alice showed more signs of engagement with programming than participants who used Generic Alice, users found Generic Alice and Storytelling Alice equally entertaining. I believe that users of Generic Alice may have enjoyed their overall experience with Generic Alice in part because they found the process of selecting and laying out 3D objects in the virtual world to be entertaining. The fact that users of Generic Alice spent less time on programming than users of Storytelling Alice may provide some support for this explanation.

10.2 Future Work

My future work lies in three broad areas: 1) designing programming environments that will help to interest girls (and boys) in studying computer science, 2) addressing the computer science pipeline problems, and 3) designing environments that leverage girls' motivation to program animated stories to teach critical thinking and communication skills.

10.3 Designing Motivating Programming Environments

One way to think about the future work in presenting computer programming to middle school girls is to consider the properties of an ideal approach to introducing middle school girls to computer programming. It would have two properties:

1. Girls would find the process of learning to program enjoyable for its own sake.
2. Girls' programming experiences would provide them with a framework for understanding the discipline of computer science.

Often introductory computer science courses concentrate on getting students to master a list of programming constructs. However, a middle school introduction to computer programming and computer science should not be evaluated based on the number of AP Computer Science test questions students can correctly answer by the end of the course. While it is important that girls get some exposure to programming constructs and concepts, I would argue that the most important aspects of a middle school introduction to Computer Science are that girls end the experience believing that computer science can be fun and that they can be successful computer scientists. A successful computer science introduction at the middle school level will motivate more students to enroll in high school and college level computer science courses. The details of particular programming languages and AP material can be deferred until high school or college.

Storytelling Alice represents a step towards a programming system that can provide students with a positive first experience with computer programming. However, there is still considerable potential for improvement. Through user testing, I was able to make the first two to three hours of learning to program a motivating and rewarding experience for many middle school girls. While the ability to provide middle school girls with a relatively short-duration positive experience with programming represents progress, it is unlikely that a single two to three hour experience will be sufficient to convince girls to pursue computer science. To begin to sway girls' perceptions of computer science, we will likely need to extend the length of time that girls are motivated to devote to computer programming in order to broaden their exposure and deepen their confidence in their ability to succeed in computer science. Determining the length of time girls should devote to learning to program and which programming constructs and concepts girls should master to have the greatest positive impact on girls' interest and confidence is an open question.

In working with the middle school girls who participated in the formative and summative testing of Storytelling Alice, I found that while girls are often confident in their ability to master Storytelling Alice, they have difficulty relating the process of creating stories in Alice to computer science. Girls often seemed skeptical that their ability to create animated movies implied an ability to learn to create computer software. Future work in improving Storytelling Alice falls into two categories: 1) extending the length of time over which girls remain engaged by programming in Storytelling Alice and 2) providing support to enable girls to move towards advanced programming concepts that will help to build their confidence and provide them with a context for understanding (and making informed decisions about) computer science.

10.3.1 Extending Engagement with Computer Programming

Computer games, while not always educational, do keep users engaged in a computer activity for lengthy stretches of time. One interesting area for future research is to examine the the reward strategies used in games to determine if they can help to keep users engaged in the activity of computer programming. There is one important

difference between gaming and Alice: games need to keep users interacting with any aspect of the game whereas Storytelling Alice should keep users engaged in the activity of programming, a subset of the possible activities within the program. In examining potential reward strategies for Storytelling Alice, it is critical to consider not just whether or not kids are motivated to continue interacting with the program but also the intellectual merit of the activities they are choosing. Some potential additions to Storytelling Alice may encourage users to spend more time with the program but discourage them from spending their time on the programming aspects of the system.

One of the most commonly used rewards in computer games is granting users access to new content: additional characters, clothing, tools, or new levels. Unfortunately, the cost of generating large quantities of high-quality art assets for use in a research project is prohibitive. However, based in part on the success of Storytelling Alice at getting middle school girls interested in learning to program, Electronic Arts has donated the 3D characters and animations from The Sims 2 for use in the next version of Alice. The Sims is the all-time best-selling PC game and is one of the few games that has a larger female than male player population, although it is widely played by children of both genders (Smith 2006). Consequently, users are likely to recognize The Sims 2 assets. The opportunity to use "real" characters from a familiar game to create more expressive stories may help maintain users' motivation for longer periods of time. The assets from The Sims 2 afford an opportunity to explore using content-based rewards to keep users engaged in programming within a storytelling version of Alice 3.

10.3.1.1 Custom Characters

Figure 10.1: A screenshot of the character-builder in The Sims 2. The user is currently choosing a hairstyle and color for their Sim character.

Using the Sims characters, it will be possible to create a character-builder that will allow girls to create their own characters. For example, users may elect to create Sims characters that resemble themselves and their friends. While the ability to create specific characters will probably be motivating for girls, it creates a risk that girls will elect to spend the majority of their time perfecting the physical appearance of their characters rather than using their characters in programs. Where girls who used Generic Alice often chose to express themselves through "decorating" their Alice worlds rather than programming, girls using Alice 3 may choose to express themselves through the physical appearance of their characters. I would like to design an experiment to determine whether girls will do more programming when they are given unrestricted use of the character-builder or when they have to "earn" new characters by programming.

10.3.1.2 Very High-level Methods

In creating Storytelling Alice, I added a set of high-level methods that allowed users to more easily create the kinds of stories that they envisioned. Constructing stories by individually controlling each joint on a character's body was overwhelming for some users and frustratingly tedious for others. We are faced with the opposite problem in incorporating the Sims characters and animations into Alice: the Sims characters come

with a large set of very high-level animations that characters can perform (~2000). The animations from The Sims 2 are at a higher level than those provided in Storytelling Alice. For example, in addition to walking and sitting, Sims characters can perform a wide variety of very specific animations like jumping on a bed or making a sandwich. However, unlike the character-specific methods in Storytelling Alice which are written within Alice and can be edited by users, the animations that come with the Sims 2 models are created in such a way that they will not be user editable or viewable as methods in a programming language. Unlike in Alice 2 where characters animations are procedurally generated, the Sims 2 animations are created through keyframing. While users will have access to a large number of well-crafted character animations, users will not be able to use character animations as programming examples or starting points for creating their own animations.

The existence of a large library of high-quality animations allows us to ask a number of important questions:

.

- *If users have a large set of very specific actions on which to base their stories, will they still learn a variety of programming constructs?* In analyzing girls' storyboards and the programs created by Storytelling Alice users, I found that character actions like hugging another character or dribbling a basketball provide the main motivations for the use of programming constructs like loops and parameters arose. There is some risk that by providing users with a large set of high-level animations we may remove the motivation to learn programming constructs and concepts. Although it is possible to envision higher-level actions such as washing a pile of dishes that could motivate concepts like loops, however the actions that motivated the usage of more complex programming constructs in girls' storyboards (all created before interacting with Storytelling Alice), will largely be provided with the Sims characters. To encourage users to explore more complex programming constructs may require choosing a different domain like controlling crowds or creating smart characters. These other domains may not be as motivating to girls as storytelling.

- *Can we give users a small "starter" set of animations and gradually unlock additional high-level methods as users demonstrate mastery of different programming constructs?* A model of gradually unlocking animation content has the nice property that girls who devote more time to programming will gain the ability to create more visually appealing animations than beginning users, creating a motivation to progress.

10.3.1.3 Increasing Expressiveness

Rather than providing an extensive library of pre-created actions (such as the library of Sims animations), it may be desirable to create a smaller "starter" set of animations and allow users to develop actions they need for their stories. To successfully create such a system we must provide girls with a reasonable set of animations for animating characters' body parts. While the existence of the *touch* and *keep touching* methods makes it easier to create animations involving physical contact with other characters or objects, girls often want to create gesture based actions such as conversational gestures or waving to another character. Studying how girls describe the gestures that they want their characters to perform and creating animations that girls can easily compose to achieve those gestures is another area for future work.

Users are often frustrated by the lack of facial expressions; characters' faces are typically images that cannot be animated. For example, a user who is telling a story in which a character is sad or angry may want the character's face to reflect their emotional state and cry or look frustrated but the character's face remains stuck in a permanent smile. Facial animation is a difficult problem that has been explored in computer animation. Techniques include the Facial Action Coding System (FACS) which combines the basic motions of facial muscles (or action units) to create facial expressions (Ekman, Friesen et al. 2002) and blending between pre-defined 3D facial shapes (Joshi, Tien et al. 2003). Neither of these techniques is appropriate for use by middle school students without some simplification. Performance animation, the use of a video camera to capture a performance of target facial animations (Chai, Xiao et al. 2003) might also provide an appropriate user interface.

In developing Storytelling Alice, I began with an early version of Alice 2. In Alice 2, all characters can perform the same set of low-level animations inspired by 3D graphics. Using the Alice 2 animations, it was possible for users to build any story for which there were appropriate models. In early user testing of Alice 2, I found that many users struggled to find a story idea that they wanted to pursue. In many ways, this seemed similar to writer's block. Through the StoryKits seminar and further user testing, I found that the characters and animations in the gallery could provide a starting point for users' stories. In Storytelling Alice, each character can perform a small set of unique animations. Users often generate story ideas by constructing a sequence that motivates the use of a particular character specific animation. The addition of a large number of high-level animations for characters has the potential to make it more difficult for users to find story ideas.

Further, there is a potential design challenge in providing users with access to a large number of high-level animations while maintaining the property that the available character-specific animations can serve as story inspiration. While preserving the story-inspiration potential of animations is of particular importance in creating a programming system for middle school girls, I suspect that helping users find story-inspiration will be beneficial for a broad spectrum of users.

10.3.2 Simulating Movie Extras

Girls' storyboards and the questions they asked during user testing revealed a need for the ability to create film extras for their movies. Extras are the characters that fill in a scene: the nameless students in a classroom, the people walking in the park, or the audience members at a show. Girls often want to assign some basic behaviors to these characters to add detail and believability to their scenes, but do not want to take the time to individually script the behavior of each extra within the scene. Modeling the behavior of extras would provide a context for discussing the use of computing as simulation. Computational simulations also afford the opportunity to experiment with small changes to starting conditions and observe the impact of those changes on the behavior of the system. The realization that small changes in starting conditions can sometimes have a large impact on the how events unfold is an important lesson. For middle school students,

it seems most appropriate to focus on enabling users to develop interesting behaviors for groups of characters. Some systems for novice programmers have focused on simulation (Repenning 1993; Smith, Cypher et al. 1994; Resnick 1996; Kloper and Begel 2006), but none within the context of allowing users to create extras for animated movies. There is also extensive work within the computer graphics community on optimizations that will enable real-time rendering of small and large numbers (Aubel, Boulic et al. 2000; Dobbyn, Hamill et al. 2005) of virtual humans at an acceptable frame-rate. Because large-scale human simulation can be computationally expensive, providing simple mechanisms for switching between high and low-cost representations of human beings may enable users to explore performance issues without requiring them to understand all of the low-level graphics details.

As with the development of the high-level animation set for storytelling, the development of tools for controlling the motions of extras should be designed based upon study of the types of extras girls want to add to their stories. It is possible that providing basic structures such as flocking(Reynolds 1987) and selecting actions at random from a list may provide sufficient support.

10.3.3 Achieving Goals without Complete Control

In The Sims video game, users do not have full control over their characters' actions. If, for example, a Sims character is feeling tired, they may not be willing to clean up the kitchen without first taking a nap. Further, a user cannot instruct two Sims characters to fall in love; the user will have to manipulate the moods of both characters such that they fall in love. Rather than dictating the story to their characters, users essentially have to coax the characters into performing a story. For some Sims players, this indirect control is part of the appeal of the game. The fact that characters can only be indirectly coerced rather than directed to perform certain actions may help to make those characters believable as real people and reinforce users' engagement with their task.

Based on the appeal of the Sims' incomplete control, I would like to investigate scenarios in which users have to achieve goals involving multiple characters but only have programmatic control of a single character. For example, one goal might be to "make

character A become friends with character B." Users can program the actions of character A; character B's actions and emotional state are computer-controlled. At the beginning of the project, girls might create a strategy for developing friendship based on a fixed set of initial conditions. As they become more experienced, they could adapt their friendship strategy to handle a broader range of initial conditions. The program controlling character A would have to select appropriate actions based upon character B's responses. To create a "friendship" program for character A would require use of logic and conditional statements, concepts that do not arise frequently in girls' stories. In some respects, this is an alternative programming interface for shorter Sims-like scenarios.

10.3.4 Computer Games

Recently, there has been a strong interest in exploring using computer game related examples to interest a larger audience of students in studying computer science. While I strongly believe that storytelling provides a gentler introduction that is motivating for a broader group of students, computer games can also be used to illustrate a variety of complex concepts within computer science. As students become comfortable with the basics of computer programming and are ready to tackle more complex materials, games may provide a good source of motivating examples. However, because games still differentially appeal to girls and boys, it will be important to carefully select which kinds of games we ask students to create.

Currently, it is possible to create simulations and intelligent characters within Storytelling Alice, but these types of projects are both complicated and awkward. To adequately support storytelling, it was necessary to study the kinds of stories that girls wanted to create and modify Storytelling Alice to more readily support girls in creating stories. To make games and simulation style projects approachable for middle school girls will require studying how girls think about and describe games and simulations and developing and testing appropriate modifications to Alice.

10.4 Addressing Computer Science Pipeline Issues

Two of the challenges in attracting a larger and broader community of students to computer science are 1) finding ways to reach students and 2) developing methods for measuring our progress in attracting new students to computer science.

10.4.1 Broadening the Focus to All Students

While I concentrated primarily on creating a programming system that makes the process of learning to program appealing for girls, my long term goal is to create a programming system that provides a better introduction to computer science for all middle school students, not just girls. I included some boys and underrepresented minority students in the development of Storytelling Alice to help ensure that I did not make changes to the system that made it appeal only to girls. However, formal verification that storytelling is a good approach for everyone, not just girls, is an important next step. I would like to repeat the evaluation of Storytelling Alice and Generic Alice with groups of middle school students including boys, African-American, Hispanic, and other minority students.

I believe that the majority of changes I made to support storytelling will make the process of learning to program more attractive for both boys and girls. However, in creating Storytelling Alice, I added a small number of animations (i.e. kneel and fall down) to support girls in commonly occurring storylines. In examining the kinds of stories that boys and minority students create, I may find other animations that are necessary to support the kinds of stories that they would like to create.

10.4.2 Integrating Alice into Schools

To have the maximum impact, we should consider how we are going to get Storytelling Alice (or the systems that come after it) into the hands of the greatest number of girls. Finding ways to integrate usage of Storytelling Alice into middle school curricula seems to have the greatest potential for reaching a large number of girls. While I plan to continue improving Storytelling Alice, the practicalities of integrating usage of Alice into schools may inform the design of future versions of Storytelling Alice. There are several possibilities for how Storytelling Alice might be used both in computer science courses and in core classes like English or History.

10.4.2.1 Computer Science Courses

Arguably, the easiest way to integrate usage of Storytelling Alice into middle schools is through computer science or computer skills courses. Unfortunately, computer science is not a part of most middle school curricula. Consequently, if it is offered at all, it will tend to be offered only as an elective and may be more likely to be offered at schools in economically advantaged areas. Despite the relative rarity of computer science courses in middle schools, taking curricular time in a computer science course to address issues like relating Alice programming to general-purpose programming and providing students with a more realistic view of the kinds of careers one can pursue with a computer science degree is entirely appropriate. In non-computer science courses, these topics might be a distraction both to teachers and students. In a computer science course, the important issues moving forward will be extending the range of concepts that students are motivated to tackle and helping them to relate their experiences with creating movies in Alice to the broader discipline of computer science.

10.4.2.2 Non-Computer Science Courses

Integrating Alice into non-computer science courses may have the potential to reach a much broader group of students because courses like English, history, and math are required subjects of study for all students. So, finding ways to integrate usage of Alice into these classes can greatly expand the numbers of students who are exposed to Alice. Expanding the audience of students who have a positive first experience with computer programming may help to significantly broaden the community of students interested in computer science. Creating stories has the potential to get students to think about a variety of issues that could be relevant across the middle school curriculum. In English courses, Alice could be used to get students to consider how a character's choices in a novel they are studying affected how the story played out and think about how different choices may have resulted in different circumstances. In history courses, students could be asked to role-play as historical figures in order to encourage them to think more deeply about the complexity of historical situations. Creating animated stories may have a role in a variety of courses beyond English and history as well. However, in non-computer science classes computer science concepts are unlikely to be at the forefront

and may become a distraction. While lesson plans for use in particular classes can help teachers get started, teachers, many of whom feel overburdened already, are unlikely to use Alice if they need to become competent programmers to successfully deploy Alice in their classrooms. In this setting, what we want is to minimize what teachers need to master but provide pathways in the Alice environment for students to learn about, experiment with, and (hopefully) begin to master, a broader range of topics than those necessary for completing their assignments.

10.4.3 Encouraging Exploratory Learning

Although Alice includes a lot of support to make the process of learning to program easier by preventing syntax errors and helping users to visualize their programs as they execute, there are still aspects of creating programs in Alice that are troublesome for beginning programmers. For example, users may accidentally choose to open a character's method and begin editing it. Then, when they press the play button, nothing happens because the method is not called anywhere. Users often have no understanding of why their program does nothing when they press play and therefore are at a loss when trying to figure out how to move forward. One could argue that the ability to make these kinds of mistakes is acceptable within the context of a computer science course because students will have access to a teacher who can help to explain concepts like methods and recursion and it is important to maintain similarity with more standard programming languages like Java and C++ to ease the transition to using these languages later. However, in middle schools, students are unlikely to need to immediately transition to a professional language like Java or C++. Further, help from teachers may not be as readily available both because teachers are not as experienced with programming themselves and because they do not want to devote class time specifically to computer programming-related issues. To facilitate usage of Alice in non-computer science courses, it will be important to make it safe for users to explore programming by minimizing the potential for users to make mistakes that "break" their programs.

10.4.3.1 Supporting Informal Use

Because integrating any programming system into middle schools is likely to be a long and complicated road, many researchers and educators focus on providing extra-

curricular opportunities to learn computer programming or the resources that will enable children to teach themselves programming. While there are certainly fewer organizational issues to work through, in many ways this is a more difficult path. Rather than competing with typical school lessons and assignments, informal usage software has to compete with students' leisure activities. Explicitly incorporating a reward structure that helps to maintain users' engagement with Storytelling Alice will likely help increase the appeal of using Alice as a leisure activity. In addition, developing an online community or local animation festivals which enable users to share their work and see what others have done may play a role in keeping users engaged.

Parents support and encouragement plays an important role in keeping children engaged in activities over time. However, at present it seems far from certain that parents would actively support their children's usage of Alice. Teachers at the middle and high school levels report that parents are currently hesitant to encourage their children to pursue computer science. Media reports of outsourcing jobs overseas have convinced many parents that computer science does not provide the kind of job security and opportunities that they want for their children (Patterson 2005). In fact, the Bureau of Labor Statistics predicts strong growth in computing related jobs; computer and mathematical science is one of three occupational groups that are expected to account for nearly 75% of new professional jobs. (Hecker 2005) Before parents are likely to encourage their children's interests in learning to program, it will likely be necessary to educate parents about the projected needs for computer scientists.

10.4.4 Evaluating at Longer-Term Engagement

Although the long-term goal of my work is to draw a larger and more diverse group of students into computer science, it was not feasible to evaluate my thesis based on the number of girls who enroll in high school or college computer science courses in several years. Particularly in the early stages of designing Storytelling Alice, it was important I be able evaluate and iterate quickly. Metrics like the number of girls who sneak extra time to work on their projects can provide insight into whether we are making progress in making computer science appealing to girls in hours rather than years. As I extend the length of time that girls spend programming in Storytelling Alice, it will be important to

develop metrics for evaluating girls' motivation over days, weeks, and months rather than hours. By developing a suite of metrics we can use to evaluate how successful a piece of software is at engaging users over gradually lengthening periods of time, I believe that we will be able to develop a deeper understanding of what factors contribute to girls' engagement with computer science and how we can design programming systems and curricula that help to interest more girls in studying computer science.

10.5 Moving Beyond Computer Science

Helping students develop critical thinking and problem solving skills is an important goal for K-12 education (Skills 2006). Activities like Storytelling Alice may hold more promise in imparting high level learning and thinking skills to students than teaching computer science skills.

10.5.1 Teaching Communication

I have observed that the desire to share animated movies with friends provides a nice entrée into developing communication skills. Unlike book reports or even short stories, students are very motivated to share the movies they create. The process of sharing their stories with other students helps students to realize which aspects of their stories they are not effectively communicating to their audience. The potential for sharing movies with a larger audience through Google Video or You Tube may provide additional motivation for creating effective stories. Because the ability to communicate ideas is a fundamental skill for all adults, one important avenue for future work might be to investigate the use of Alice as a tool for teaching communication skills. This process might include investigating different ways to capture audience feedback and present it in such a way that users understand which aspects of their stories are and are not working and are motivated to address the weaker parts.

10.5.2 Complex Reasoning

The world around us is filled with complex systems whose behavior depends on the behaviors and interactions of smaller parts within the system: cars, weather, and manufacturing plants, to name just a few. Yet our schools do little to prepare students to work with and attempt to understand the behaviors of complex systems. On a day to day basis, complex reasoning may provide students with the ability to work through problems

they encounter with household electronics or cars, even without specific training as electricians or mechanics. On a more global scale, when we, as a country, make environmental policies that will impact neighboring states, countries, and the rest of our planet over many years, we would like our citizens to be able to recognize that seemingly simple actions such as how much energy we consume in heating our homes can have profound environmental effects. Particularly when students move into the domain of creating simple simulations that have unanticipated behaviors, programming provides children with some hands on experience dealing with complex systems that they create themselves. When their programs do not behave as expected, children have to learn to isolate the problems and solve them. They learn to ask questions that will allow them to narrow the scope of a problem and they learn that a single malfunctioning piece within a program may cause problems in seemingly unrelated parts of the program.

10.6 Conclusion

The results from my evaluation of Storytelling Alice suggest that storytelling is a promising approach for attracting more middle school girls into computer science. However, the results are based on short, four-hour workshops. To build on the success of Storytelling Alice, I would like to concentrate both on extending the time that girls devote to programming within Storytelling Alice and increasing the range of programming concepts that girls master. Providing users with the ability to create more detailed animations and control the behavior of film-style extras may help to motivate users to explore a wider range of programming constructs. Developing social simulations in which users have goals to achieve without complete control may also provide a motivating environment that has the potential to introduce interesting programming concepts. For example a simulation might provide users with a single character that they can control programmatically and a goal such as "make your character become friends with Jenny (another character)."

Storytelling Alice has the greatest potential for impact if it helps to draw all students to computer science, not just girls. Two important avenues for future work are 1) to formally evaluate the impact of the storytelling focus on boys and minority students and 2) to find ways to enable Storytelling Alice to reach the greatest number of students.

Schools provide a pathway that will enable us to reach the greatest number of students. Because computer science is not a part of most middle school curricula, we should find ways to integrate the use of Storytelling Alice into core classes such as English and history. Supporting user exploration and providing ways for users to teach themselves new programming concepts may help to maximize Storytelling Alice's potential to draw students into computer science while not requiring teachers to focus on teaching computer science skills in their core classes.

Chapter 11 Programming Languages and Environments for Novice Programmers

Note: This chapter is drawn from my ACM Computing Surveys paper entitled "Lowering the barriers to programming: a taxonomy of programming environments and languages for novice programmers" (June, 2005). I have added the Scratch, StarLogo TNG, JPie, and RAPUNSEL systems to update the taxonomy.

11.1 Introduction

Learning to program can be very difficult for beginners of all ages. In addition to the challenges of learning to form structured solutions to problems and understanding how programs are executed, beginning programmers also have to learn a rigid syntax and rigid commands that may have seemingly arbitrary or perhaps confusing names. Tackling all of these challenges simultaneously can be overwhelming and often discouraging for beginning programmers. Since the early 1960's, researchers have built a number of programming languages and environments with the intention of making programming accessible to a larger number of people. This chapter presents a taxonomy of these languages and environments and discusses the challenges they address.

For the purposes of this chapter, we define programming as the act of assembling a set of symbols representing computational actions. Using these symbols, users can express their intentions to the computer and, given a set of symbols, a user who understands the

symbols can predict the behavior of the computer. This definition excludes many of the "Programming by Demonstration" systems (Cypher 1993), where the computer observes the user's actions and uses internal heuristics to generate a program for the user. In these systems, the user cannot accurately predict what program will be produced.

In this paper, we describe the high level organization of our taxonomy, present the taxonomy and briefly describe all of the categories and systems within those categories. We then present two additional tables: a table of the most influential systems and a system comparison table. The system comparison table compares all systems in our taxonomy, based on 1) what programming constructs they support and 2) their approaches to making programming more accessible to novice programmers. Finally, we summarize the approaches and discuss some possible avenues for future work in this area.

11.2 Taxonomy

In creating a programming environment for novices, one of the first questions that must be answered is why novices need to program. There are a variety of possible motivations for learning to program: to pursue programming as a career path, to learn how to solve problems in a structured and logical way, to build software customized for personal use, to explore ideas in other subject areas, etc. The systems in this taxonomy (see Figure 1) fall into two large groups: systems that attempt to teach programming for its own sake and those that attempt to support the use of programming in pursuit of another goal, such as teaching cognitive modeling to psychology students. Because these two goals place very different constraints on systems, the taxonomy is organized first by the system goals, either teaching or using programming, and, second, by the primary aspect of programming that the system attempts to simplify. Each system appears in the taxonomy only once. However, many of the systems in the taxonomy have built on the ideas of earlier systems. Consequently, a system that was influenced by natural language programming may not be classified with other natural language systems if supporting natural language programming was not the systems' primary contribution.

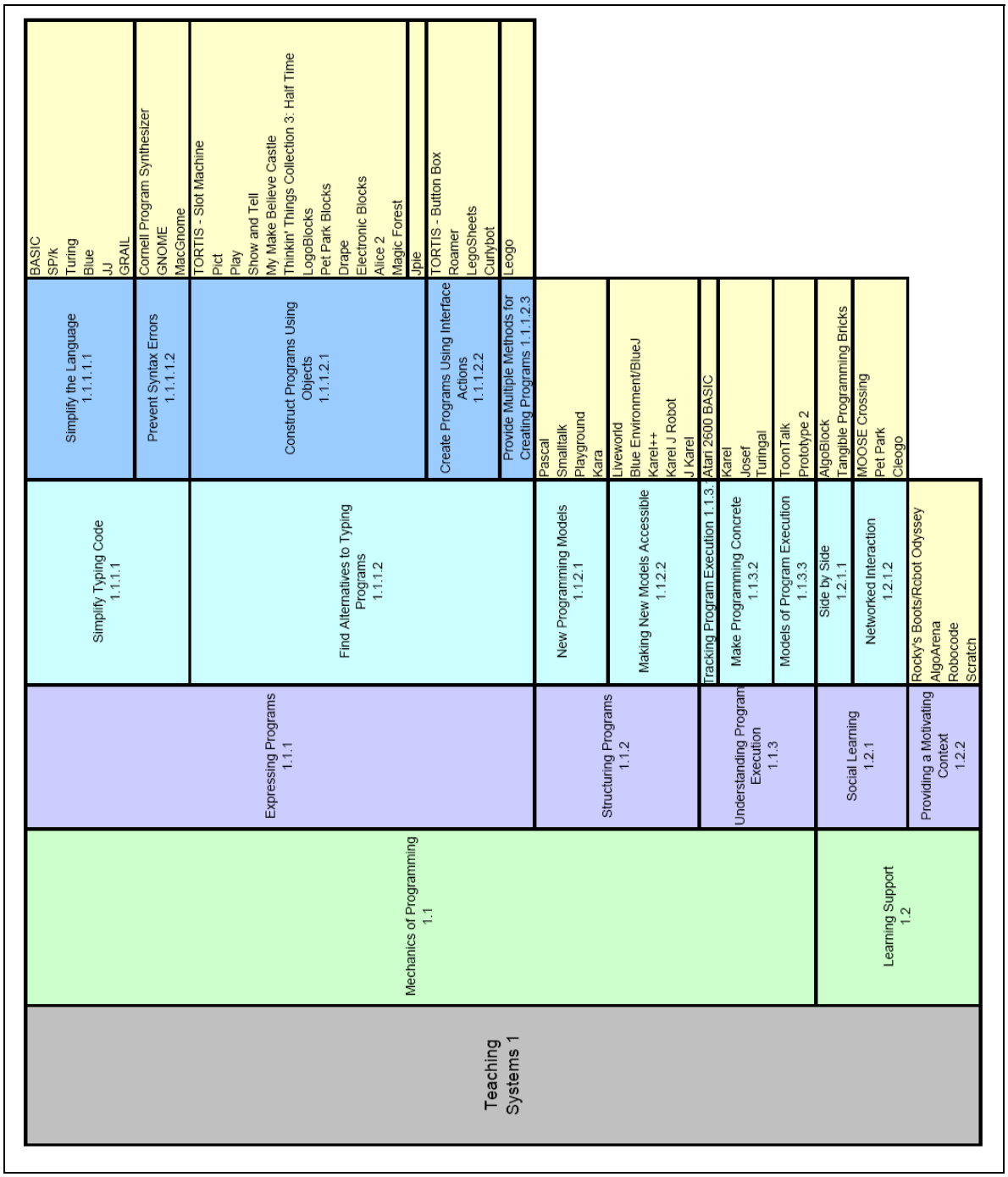


Figure 11.1: Taxonomy – Teaching Systems

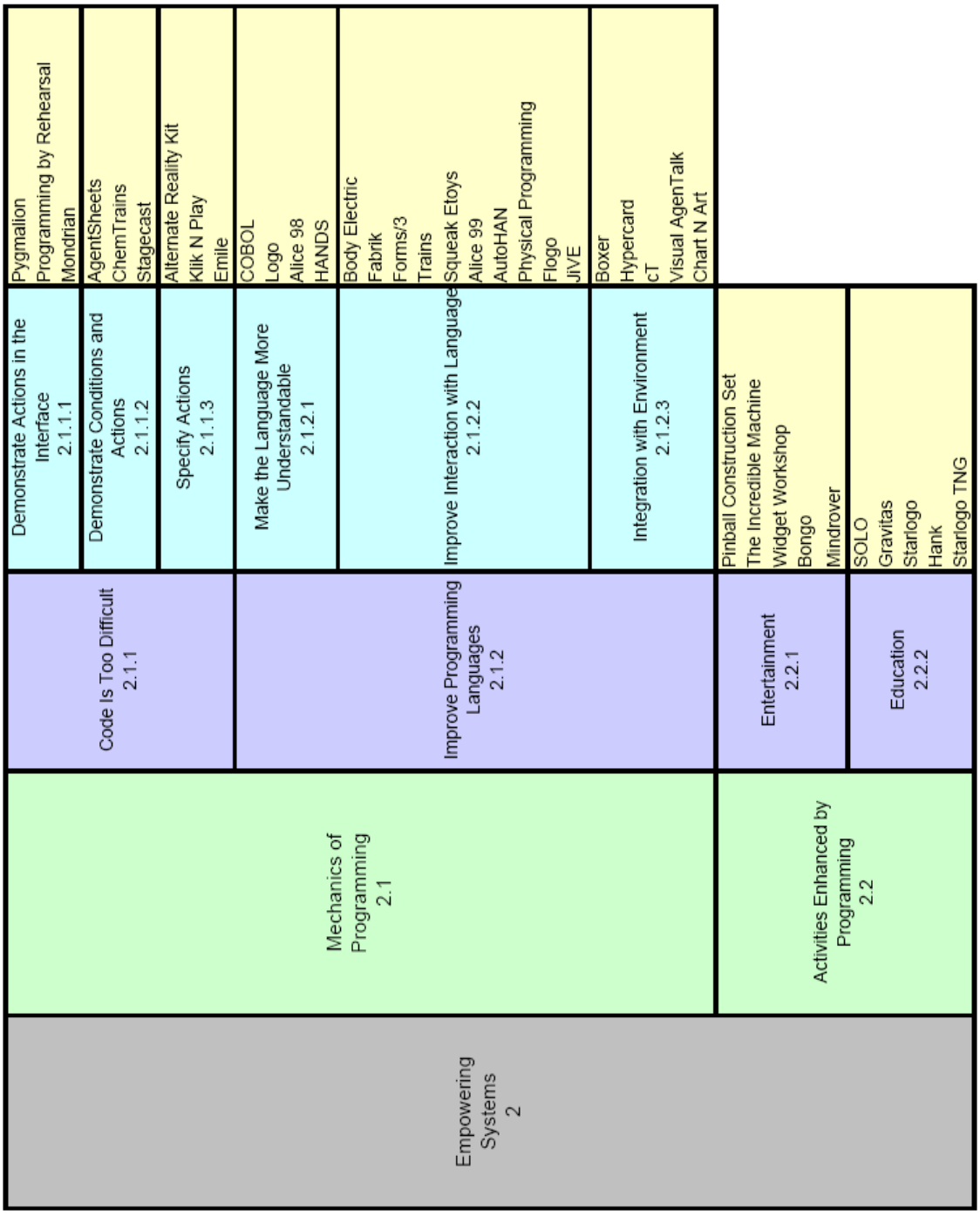


Figure 11.2: Taxonomy- Empowering Systems

11.3 Teaching Systems

These systems were designed with the goal of helping people learn to program. Most of the systems in this category are (or include) simple programming tools that provide novice programmers exposure to some of the fundamental aspects of the programming process. After gaining experience with a teaching system, students are expected to move to more general-purpose, commercially available languages. A few systems attempt to provide support in learning a more general language from the start. Because students interacting with teaching systems are expected to transition to general purpose languages, many teaching systems are intentionally similar to general-purpose languages. For example, knowing that a student will eventually have to do "for loops" in a Java-style, the designers of teaching languages are less likely to introduce a different style of looping. Because general-purpose languages are not always designed with beginners in mind, the systems in this category are juggling two possibly conflicting goals: making it easier for beginners to get started programming, and giving students a background that makes it easy for them to transition from the teaching system to a general-purpose language.

The teaching systems focus on several areas that can be difficult for novice programmers. The majority of the systems in this category address the mechanics of programming: both expressing intentions to the computer and understanding the actions of the computer (Norman 1986). Other systems attempt to place programming in a context that is accessible and motivating to a wider audience of people, either by providing concrete reasons for programming or by supporting novice programmers working together and learning from one another.

11.3.1 Mechanics of Programming

The systems in this category are designed around the hypothesis that the primary barrier in learning to program lies in the mechanics of writing programs. To successfully write a program, users must understand several topics: how to express instructions to the computer (e.g. syntax), how to organize these instructions (e.g. programming style), and

how the computer executes these statements. Systems in this category attempt to make it easier for beginners to learn one of these three skills.

11.3.1.1 Expressing Programs

In most general-purpose languages, users create programs by typing sentences into a text editor. Beginning programmers often have trouble translating their intentions into syntactically correct statements that the computer can understand. The systems in this category explore two possible avenues for making this process easier for beginning programmers: improve the language such that beginners can more easily learn it or find alternate ways for beginners to communicate their instructions to the computer.

11.3.1.1.1 Simplify Entering Code

Many general-purpose languages have been influenced by the need for sufficient power to tackle arbitrary programming tasks and a desire to make the programming language easier to implement, making the resulting languages unnecessarily difficult for beginning programmers. The systems in this category examine three approaches to making languages more approachable for beginning programmers: 1) simplifying the language, 2) tailoring the language for a specific, small domain of programming problems, and 3) preventing syntax errors.

11.3.1.1.1.1 Simplify the Language

General-purpose languages typically include a large variety of syntactic elements that can be particularly difficult for beginners because these syntactic elements don't have an obvious meaning. The languages in this category use a few simple observations to decrease the number of potentially confusing syntactic elements encountered by beginning users, while trying to maintain as much similarity as possible to general-purpose languages. General-purpose languages often contain unnecessary syntax, use commands whose names are unfamiliar or have different meanings in the programming language than in standard English, have inconsistent uses for syntactic elements, or include features inappropriate for beginning programmers. Using these observations, it is possible to make a language syntactically easier for beginners to handle without fundamentally changing the common control structures found in general-purpose languages. Consequently, when a student moves from one of these languages to a

general-purpose language, they should be able to transfer their knowledge from the teaching language.

BASIC: J.G. Kemeny and T. Kurtz, Dartmouth College, 1963 (Kurtz 1981)

Basic was designed to teach Dartmouth's non-science students about computing through programming. FORTRAN and ALGOL, the commonly used languages at the time, were both large and complex. Kemeny and Kurtz believed that the students would "balk at the seemingly pointless detail" (Kurtz 1981). After considering using subsets of FORTRAN or ALGOL, Kemeny and Kurtz agreed they would have to create their own language. The BASIC (Beginners All-purpose Symbolic Instruction Code) language was designed to support a small set of instructions and remove unnecessary syntax. The environment was designed to have rapid turn-around time and sacrifice computer time for user time (in 1963, the computer science community was arguing against high level languages because the compilation time was seemingly wasted computation).

Statements in BASIC consist of three parts: a line number (e.g. 110), an operator (e.g. LET), and an operand (e.g. S = S + 1). All commands begin with an English word to make the language easier for the novice; the designers believed that LET S= S + I would be easier for students to understand than S = S + I. Figure 11.3(below) shows a simple summation loop in both FORTRAN and BASIC. While the statements have a similar structure, the BASIC program uses language more suitable for a novice, removes elements like labels (e.g. 30) that require a more detailed understanding of the program counter, and does not depend on spacing for syntactic meaning.

FORTRAN:	BASIC:
do 30 i = 1, 10	100 FOR I = 1 TO 10
m = m + I	110 LET S = S + I
30 continue	120 NEXT I

Figure 11.3:A *for* loop to compute the sum of the numbers from 1 to 10 written in FORTRAN and BASIC.

SP/k: R.C Holt et al, University of Toronto, 1977 (Holt, Wortman et al. 1977)

SP/k is a subset of PL/1 chosen for teaching introductory programming. The features of the SP/k language were chosen to remove redundant constructs, inconsistencies in the language that go against students' intuitions (in PL/1 the expression 25 + 1/3 evaluates to 5.3333), constructs that are easily misused such as pointers, and constructs like

concurrent programming that are suited for advanced programmers. The difficulty of compiling constructs was also considered. The result of pruning was a simpler language for introductory programming that both students and teachers generally preferred over FORTRAN. The authors also provided an order for introducing programming constructs as a sequence of subsets of SP/k. SP/1 introduces expressions and output. By SP/8, students have learned all of SP/k. By introducing things gradually, students can master a small piece of the language at a time, allowing them to devote more time to problem solving than memorizing the features of the language.

Turing: R.C. Holt and J.R. Cordy, University of Toronto, 1988 (Holt and Cordy 1988)

The Turing language was developed as both a general-purpose and instructional language for the Computer Science Department at the University of Toronto. Consequently, while the designers intended that Turing be used in teaching programming, the language design was influenced by a desire to help expert programmers by including powerful programming features. The Turing language contains all the features of Pascal (see section 1.1.2.1) and adds dynamic arrays, modules, and varying length strings. In addition, Turing simplifies the syntax by removing the requirement for headers declaring the name of the program and semi-colons at the end of each statement.

Blue Language: M. Kolling and J. Rosenberg, University of Monash, 1996 (Kolling and Rosenberg 1996)

Blue is an object-oriented language designed to be taught as a first language. After using Blue for a year, students are expected to move to an industrial language, such as C++. The designers of the language used four criteria in creating Blue: there should be only one way to do everything; the language should cleanly reflect the theoretical model; the language should be readable so students can learn by reading examples; and the language should explicitly support software engineering mechanisms like pre and post conditions. The Blue language is a pure object-oriented language that supports single inheritance, garbage collection, and strong static typing. Classes are defined in single files with a structure that clearly reflects which routines others can call and which routines are internal to the class by placing routines in separate *internal* and *interface* areas within the file. Routine definitions include explicit pre and post conditions. Blue provides a single

loop structure that consists of a set of statements followed by a list of conditions that should cause the loop to exit which can be used to create loops that function like traditional for and while loops. Each loop exit condition can include statements to execute if the loop exits on that particular condition. The designers of the language also created an environment for beginning programmers that will be discussed separately.

JJ: J. Motil and D. Epstein, California State University and California Institute of Technology, 1998 (Motil and Epstein 1998)

Full featured, general-purpose languages force beginning students to focus on the syntax rather than the problem they are trying to solve in writing a program. JJ (Junior Java) is a language designed to remove much of the syntactic complexity to allow students to focus on the concepts of programming. It removes much of the punctuation such as braces and semi-colons and has only one way to do anything; there is one integer type, one way to create a comment, etc. The language also provides an easy migration to Java after the first half of the semester. Students can either do this by hand or the environment can convert their JJ code to Java automatically. Figure 11.4 shows an example of computing weekly pay in JJ and the equivalent code in Java. Due to lack of adoption, the designers of JJ have moved towards improving students' classroom experiences with Java by providing better compilation error messages and allowing students to program over the web.

Computing weekly pay in JJ:	The same code in Java:

`If (hours <= 40) then`
`  Set pay = 10 * hours`
`Else`
`  Set pay =`
`  400 + 15*(hours – 40)`
`EndIf`
 	
`Output "The pay is "`
`Outputln pay`
	`if (hours <= 40) {`
	`  pay = 10 * hours;`
	`} else {`
	`  pay =`
	`  400 + 15 * (hours – 40);`
	`} // EndIf`
	
	`System.out.print ("The pay is " );`
	`System.out.println( pay );`

Figure 11.4:A short segment of code to compute a worker's weekly pay shown in both JJ and Java. Note the line by line correspondence.

GRAIL: L. McIver, Monash University, 1999 (McIver 1999; McIver 2001)

GRAIL was developed in response to the hypothesis that "it is the unfamiliarity of 'hieroglyphics' (i.e. the language syntax) and the sheer complexity of the full theory that

are the primary stumbling blocks for the novice" (McIver 2001). Three guiding principles governed the design of GRAIL: maintain a consistent syntax; use terms that novice programmers are likely to be familiar with and avoid standard programming terms that have different meanings in English; and include only constructs that are fairly simple and have a "single, obvious syntax" (McIver 2001). These guidelines led to an imperative language with many small differences from commonly used teaching languages such as Pascal (see section 3.1.2 under *New Programming Models*). The list of changes is too long to reproduce here, but we list a few to give the reader a feel for the kinds of changes made for the GRAIL language. Rather than using * for multiplication, GRAIL uses x because it is a symbol that novice programmers will understand from mathematics classes. Values are assigned using an arrow indicating where the answer will be placed since a = b is ambiguous. McIver removed pointers because they are difficult to use correctly; using pointers it is very easy for beginners to create problems they cannot easily understand or explain. The full details of the GRAIL language can be found in McIver's thesis.

11.3.1.1.1.2 Prevent Syntax Errors

One of the largest and most frustrating challenges for novice programmers is syntax. The systems in this category are programming environments for existing languages such as Pascal and Fortran that are designed to prevent users from making syntax errors using the hierarchical structure of programs.

Cornell Program Synthesizer: T. Teitelbaum and T. Reps, Cornell University, 1981 (Teitelbaum and Reps 1981; Reps and Teitelbaum 1989)

The Cornell Program Synthesizer was a structure editor designed to prevent students from making syntax errors. Using the synthesizer, students constructed programs by adding pre-defined templates for statements in a programming language (see Figure 11.5 below). A template often contains placeholders for statements, conditions, or phrases. These are essentially blanks for the user can fill in. To prevent syntax errors, the system presented only templates that would be syntactically valid at the cursor's current location. Students could use the arrow keys to move to the next or previous place in their program where they could add, remove, or edit a template based on the abstract syntax tree. While the designers of the Cornell Program Synthesizer originally wanted to require programs

to always be syntactically valid, they found this requirement made certain kinds of edits, such as changing a variable name, extremely difficult. In response, they changed the Cornell Program Synthesizer to allow syntactically invalid statements but highlight to draw the user's attention.

An If-statement template in the Cornell Program Synthesizer:

```
IF (condition)
    THEN statement
    ELSE statement
```

Figure 11.5:This is an If-statement template as it appeared in the Cornell Program Synthesizer. The words "condition" and "statement" are placeholders the user replaces with a condition (such as k < 1) or a programming statement, respectively.

GNOME: P. Miller et al, Carnegie Mellon University, 1984 (Miller, Pane et al. 1994)

The GNOME environments were an attempt to make a structure editor for novice programmers that was more versatile than the Cornell Program Synthesizer. GNOME displayed programs hierarchically, encouraging students to think about programs as hierarchical collections of procedures. Students navigated through their programs using arrow keys that corresponded to movements in the abstract syntax tree; GNOME displayed program segments in the familiar textual form. When the programmer attempted to move the cursor after an edit, GNOME analyzed the program, reported any syntax errors, and prevented the programmer from moving on until the program was syntactically correct. The programmer could also request an analysis of the program at any time. While this environment prevented syntax errors, it actually required students to think more about syntax than they previously had: they needed to have a mental model of the syntax tree to navigate through the system; the abstract syntax representation sometimes differed from the textual representation (particularly with mathematical equations); and the requirement for syntactic correctness sometimes prevented students from making desired changes in the program because the fastest route to a correct program required intermediate stages that were not syntactically correct. GNOME environments were created for Karel the Robot, Pascal, Fortran, and Lisp.

MacGnome: P. Miller et al, Carnegie Mellon University, 1986 (Miller, Pane et al. 1994)

The MacGnome project attempted to cleanly integrate structure-editing capabilities of GNOME with the text-editing model present in traditional programming editors. The GNOME project demonstrated that students have difficulty navigating in the abstract syntax tree; to alleviate this problem, MacGnome allowed students to navigate using point and click with a mouse. In GNOME, students often had trouble modifying code because of the requirement to maintain syntactic correctness. Rather than requiring syntactic correctness at all times, the MacGnome project editors converted the syntax tree into a textual representation to allow editing without syntactic constraints. Once the user finished editing, it converted the modified code back to tree representation using an incremental parser. By allowing students to edit code textually, the MacGnome environment could not prevent syntax errors. However, MacGnome detected and reported all syntax errors as soon as the code was parsed, allowing students to correct them before moving to other sections of the program. The novice programming environments produced as a result of the MacGnome project are called Genies.

11.3.1.2 Find Alternatives to Typing Programs

Despite the attempts to make programming languages simpler and more understandable, many novices still struggle with syntax: remembering the names of commands, the order of parameters, whether or not they are supposed to use parentheses or braces, etc. Another large set of systems are designed around the belief that to enable novices to understand what programming really is, we need to bypass the syntax problems altogether. The systems in this category represent three major approaches to bypassing syntax: creating objects that represent code that can be moved around and combined in different ways, using actions of the user within the interface to define programs, and providing multiple mechanisms for creating programs.

11.3.1.2.1 Construct Programs Using Graphical or Physical Objects

The systems in this group use graphical or physical objects to represent elements of a program such as commands, control structures, or variables. These objects can be moved around and combined in different ways to form programs. Novice programmers need

only to recognize the names of commands and the syntax of the statements is encoded in the shapes of the objects, preventing them from creating syntactically incorrect statements.

TORTIS – Slot Machine: R. Perlman, MIT Artificial Intelligence Lab, 1976 (Perlman 1976)

The TORTIS Slot Machine is a physical interface that allows young children to control a robotic turtle inspired by the Logo turtle (see section 4.1.2 under *Make the Language More Understandable*); since the robotic turtle is very slow, a simulated on-screen graphical version is provided for more advanced students. The Slot Machine consists of a set of command cards and rectangular boxes (called rows), which represent procedures and contain slots for command cards. Children created Slot Machine programs by placing cards in slots of the rows and having the turtle execute the cards in order. The Slot Machine provided several uniquely colored rows so that children could create different procedures in each row. Children could call their procedures using a colored card instructed the Slot Machine to execute the cards in the row corresponding to that color.

Pict: E. Glinert and S. Tanimoto, University of Washington, 1984 (Glinert and Tanimoto 1984)

Pict allows novice programmers to create simple programs by connecting graphical icons that represent commands. Pict allows users to build programs that do simple numeric calculation using the addition and subtraction of integers, variable assignment, and Boolean tests. To create a program, users select relevant icons (commands) from a menu screen area and position them on a workspace screen area using a joystick. After positioning icons on the workspace, the user can connect a pair of icons together by clicking on the two endpoints in turn. When a user runs a program, Pict animates the execution of the program by moving a white box along the execution path of the program. Users can run a Pict program at any point in its development, if the running program reaches a point where its behavior has not been specified, it will halt and notify the user that additional programming is necessary.

Play: S. Tanimoto and M. Runyan, University of Washington, 1986 (Tanimoto and Runyan 1986)

Play is a system designed to allow preliterate children to create graphical plays using an iconic language. Stories consist of a linear sequence of actions that is displayed at the top of the screen, above the story's stage, as a sequence of icons similar to a comic strip. The character, what the character should do, and one additional piece of information, typically a direction to move, all selected from menus, specify each action in the story. Play also provides a character editor where children can draw additional images of their characters and compose those images to create new animations. Play does not allow children to use more complicated control structures such as loops and conditionals or define procedures.

Show and Tell: T. Kimura et al, Washington University and Bell Labs, 1990 (Kimura, Choi et al. 1990)

Show and Tell is a data flow based visual language designed for children. A program in Show and Tell consists of a series of connected boxes. A box can represent a value or an operation on values. The program includes boxes that represent basic arithmetic functions, system input and output, and some special purpose boxes that play sounds or act as timers, etc. Children can build procedures by drawing their own icon for a box and defining what should happen in the procedure using other boxes. Procedures can call themselves. Because boxes are not permitted to form cycles or loops, users cannot construct for and while loops. However, Show and Tell provides an iteration box that provides bounded iteration, in other words, the function will continue repeating until a boundary value is reached. If two connecting boxes contain different values (e.g. 2 and 3), they and their parent box are marked "inconsistent" and become invisible to the other boxes. By checking for consistency and inconsistency in particular boxes, children can represent simple Boolean conditions.

My Make Believe Castle: Logo Computer Systems Incorporated, 1995 (1995)

My Make Believe Castle is a play program for children ages 4-7 that contains activities designed to help develop children's problem solving, critical thinking, sequential planning, and memory. The castle consists of a number of rooms, each containing an activity. In the courtyard of the castle, characters such as the dragon, prince, princess, and horse move around. When the user clicks on them with a particular tool, they will dance, slip on banana peels, do somersaults, etc. After children have played in the courtyard space, they can be introduced to a very simple, rule-based programming system. Editors

for each character allow children to specify which action a character should take when it meets another specific character. A typical rule might be "Nicky dances when it meets the horse" (see Figure 11.6). Rules are specified graphically; children select the action using icons and the character that should trigger the action by selecting a picture of that character.

Figure 11.6: A view of the My Magic Castle courtyard. The user is creating the rule "Nicky should dance when it meets the horse."

Thinkin' Things Collection 3- Half Time: Edmark Corporation, 1995 (1995)

Half Time is one of the activities in the computer game Thinkin' Things Collection 3. The activity revolves around creating a half time show (see Figure 11.7). Users can select characters from the top left and drag them onto the field; each half time show can have a total of thirty characters across three types (such as tuba, percussion, and trumpet players). At the bottom of the screen, there is a line for each of the three types of characters in which users can drop instructions for them to perform. The available instructions are similar to those of the Logo (see section 4.1.2 under *Make the Language More Understandable*) turtle: move forward, turn left and right, turn randomly, pause, pen down and up, etc. Programs are created by dragging the icons for instructions (shown below the football field) into the lines for a particular type of character. Counted loops are supported, but no other block statements are available.

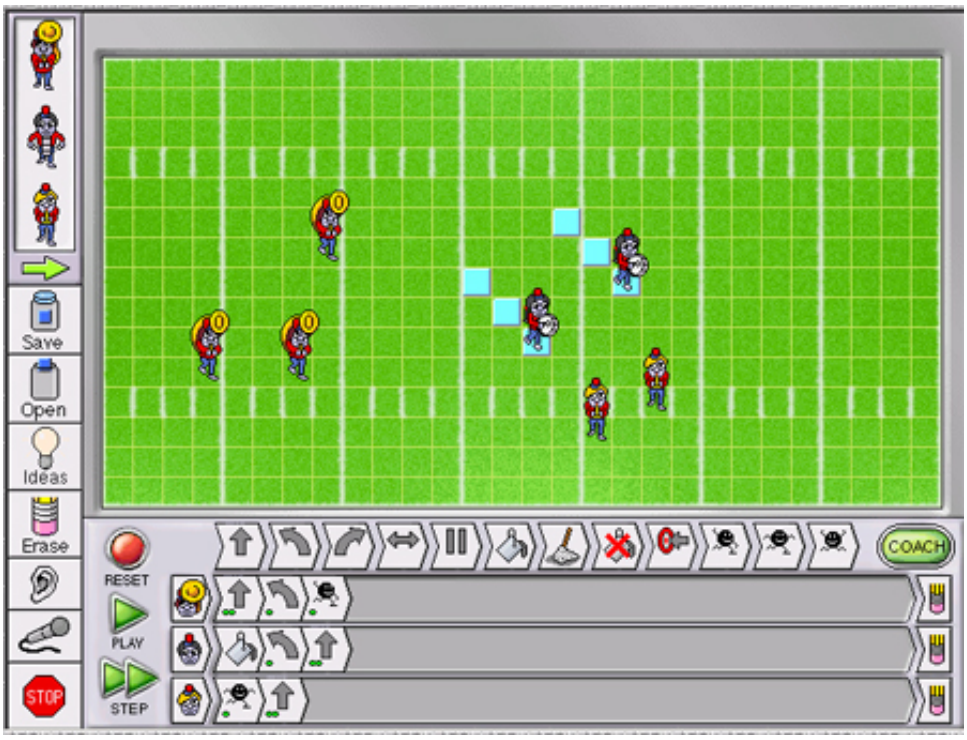


Figure 11.7:A screenshot of Half Time from Thinkin Things Collection 3

LogoBlocks: A Begel, MIT Media Lab, 1996 (Begel 1996)

LogoBlocks is a graphical programming language designed for the Programmable Brick, a precursor to the commercial Lego Mindstorms system (1998), developed by the MIT Media Lab (see Figure 11.8). In LogoBlocks, labeled graphical shapes represent commands in BrickLogo, an extension of Logo (see section 4.1.2 under *Make the Language More Understandable*) that provides commands for the Programmable Brick. These graphical blocks can be dragged off a tool palette on the side of the screen to a main work area where they can be placed next to other blocks to form programs. Like many visual programming environments, changes to programs may require the user to move existing statements to make room for new ones. The parts in the palette can take several forms, for example a block marked 'A' specifies the motor A as the recipient of commands following it, but, by clicking on the 'A' block, the user can turn it into a 'B' or an 'AB' block. Commands and conditionals also have multiple forms; the blocks in the tool palette represent kinds of objects rather than all available objects. Commands and conditionals requiring arguments have shapes with cutouts for placing the arguments so that it is clear both that the command requires an argument, and the type of the argument

which is specified by the shapes of blocks that will fit into the cutout. LogoBlocks includes support for procedures; users can attach commands to purple procedure blocks and name their procedures.

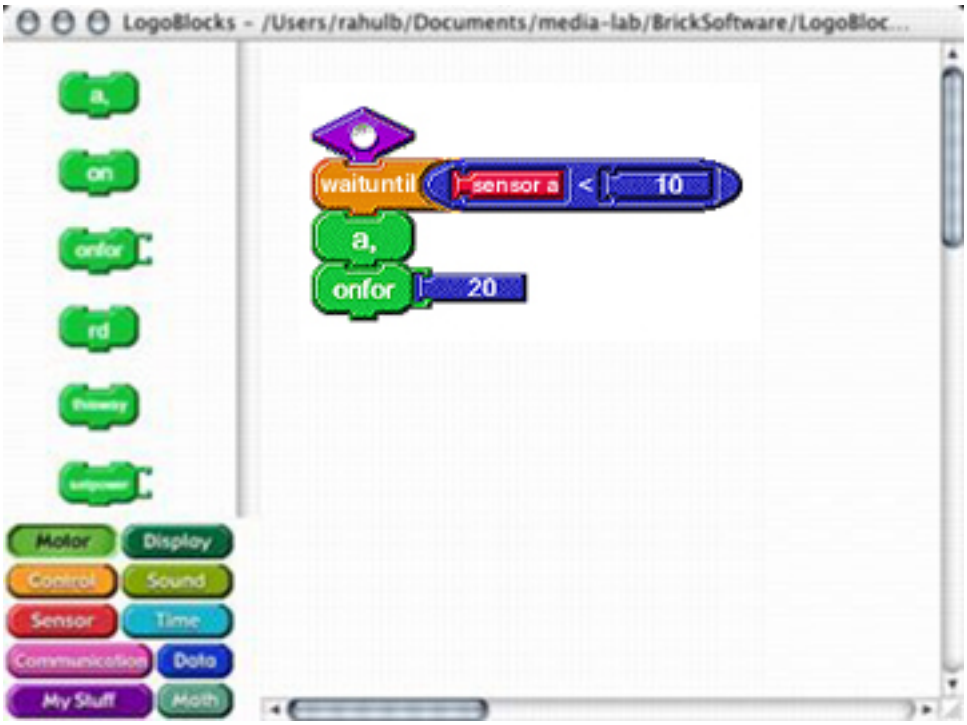


Figure 11.8:A LogoBlocks program that waits for a light sensor to get a reading of less than 10 and then turns motor A on for 20 seconds.

Pet Park Blocks: A. Cheng, MIT Media Lab, 1998 (Cheng 1998)

Pet Park Blocks is a graphical programming language, inspired by LogoBlocks, which was developed for the Pet Park collaborative environment (described in 3.2.1 under *Networked Interaction*). Animations are represented by notched squares that fit together. Conditionals are represented by squares with half oval cutouts where conditions can be added. Like LogoBlocks, programming constructs are kept in a palette from which users can drag them onto an active area. Pet Park Blocks provides a button that allows users to see their Blocks program as a textual program. This allows users to gradually transition to text-based programming.

Drape: M. Overmars, Universiteit Utrecht, 2000 (Overmars)

Drape is a programming environment that allows users to draw pictures (see Figure 11.9). There is a collection of pictorial icons on the left side of the interface that represent

different commands similar to the Logo (see section 4.1.2 under *Make the Language More Understandable*) turtle commands: pen up, pen down, move in different directions, move in shapes, etc. The icons can be dragged to the lines at the bottom of the screen that represent the program; commands are executed from left to right. There are extra lines associated with their own icons that can serve as procedure calls. The system does have support for some predefined blocks such as repeat 10 times (shown as x10) However, to apply the repeat 10 to more than a single object, the sequence needs to be enclosed in brackets, which introduces the possibility for syntax errors in the form of mismatched braces.

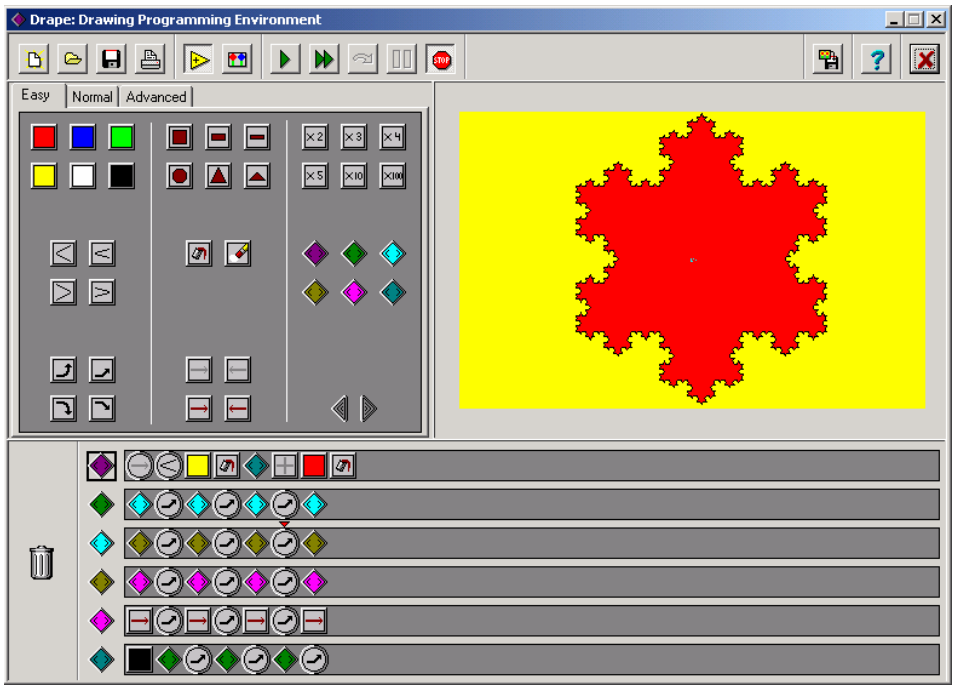


Figure 11.9:DRAPE Drawing and Programming Environment allows children to draw pictures.

Electronic Blocks: P. Wyeth and H. Purchase, University of Queensland, 2000 (Wyeth and Purchase 2000)

Unlike the graphical objects used to construct programs in other systems, Electronic Blocks are physical Lego blocks designed to allow young children (ages 3-8) to create Lego forms with interesting behaviors (see Figure 11.10). Preschool children can build block towers that flash when they talk or cars that move when a flashlight shines on them. Three types of blocks are provided: sensor blocks that can detect light, sound, and touch; logic blocks that can compute AND, NOT, TOGGLE, and DELAY; and action

blocks that can produce light, sound, and motion. The syntax of Electronic Blocks is very simple; the only requirements are that each stack includes a sensor block and an action block and that the action block be at the bottom of that stack. Action blocks are smooth on the bottom so they cannot be placed on top of other block types.

Figure 11.10:Electronic Blocks: the three sensing blocks are pictured on the left, the logic blocks in the middle, and the action blocks on the right

Alice 2: Carnegie Mellon University, 2002 (2003)

Alice is a programming system for building 3D virtual worlds, typically short animated movies or games. In Alice users construct programs by dragging and dropping graphical command tiles and selecting parameters from drop-down menus. Figure 11.11 shows an Alice screen as a user creates a simple animation. To add to the current animation, the user drags a graphical tile labeled with the name of the desired action from the selected object's methods, in this case the IceSkater's methods, displayed in the lower left panel. When the user drops the tile, the system automatically cascades to menus that allow the user to select valid parameters for the chosen method. In Figure 11.11 the user has just dragged and dropped *IceSkater turn* from the panel and has chosen to have *IceSkater* turn right one full turn. Students can also add standard programming control structures such as if-statements and loops by dragging *if* and *loop* tiles from the top bar. Where many no-typing programming systems present users with only a few of the standard programming constructs, Alice allows students to gain experience with all of the standard constructs taught in introductory programming classes without making syntax errors.

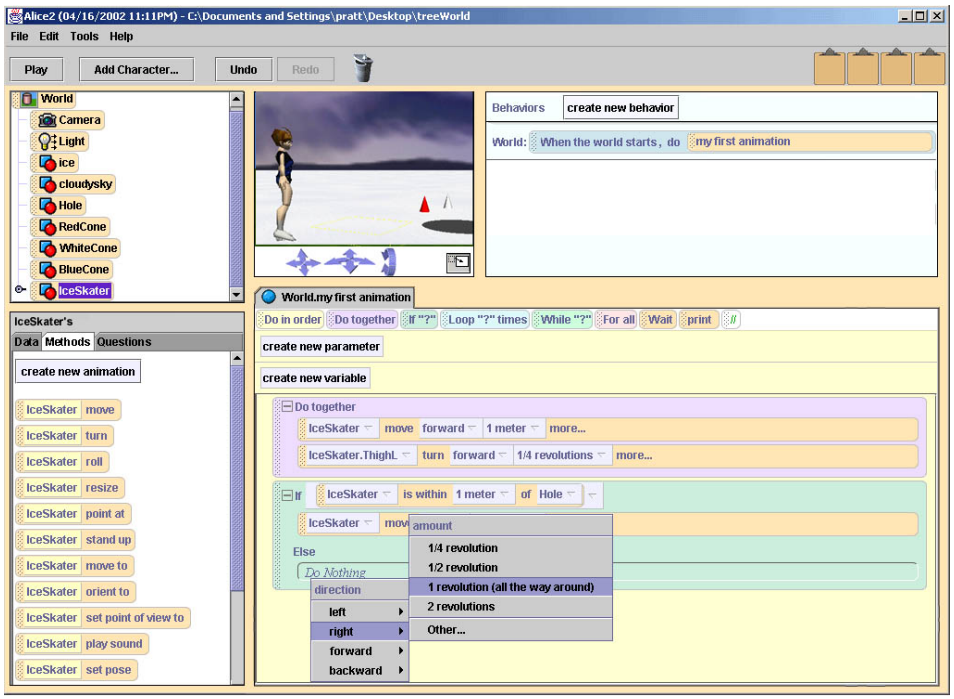


Figure 11.11:Building *my first animation* in Alice. In *my first animation, IceSkater* moves forward while she raises her leg. Then, if *IceSkater* is close to a hole in the ice, she falls through it.

Magic Forest: Logotron, 2002 (2002)

Magic Forest (see Figure 11.12) allows children ages four and up to play with, change, and create *Activities* that consist of 2D sprites that can move around, change appearance, and react to simple events. Each sprite can be given a set of *Rules* (represented by a scroll containing stones), a combination of an event and a list of things that should happen, in order, after that event occurs. Both events and actions are represented by graphical stones that can be identified by their icons, making it possible for children to learn how to use Magic Forest without needing to know how to read. Magic Forest supports a variety of events, such as mouse based events, events based on the relative positions of objects, and message passing events. Actions might change the direction or speed of an object, the appearance of an object, send a message, play sounds, or update the score. To add a new rule to a sprite, a child selects an event from a scrolling list of available event stones, clicks on it to pick it up, and then drops it onto a scroll associated with that sprite. The child can then attach action tiles to the end of the event. As in Logoblocks, some tiles can have multiple forms; a single tile can be used to increase the speed, heading, or size of an

object. Children can click on a tile to change which form it takes (increase speed, heading, or size).

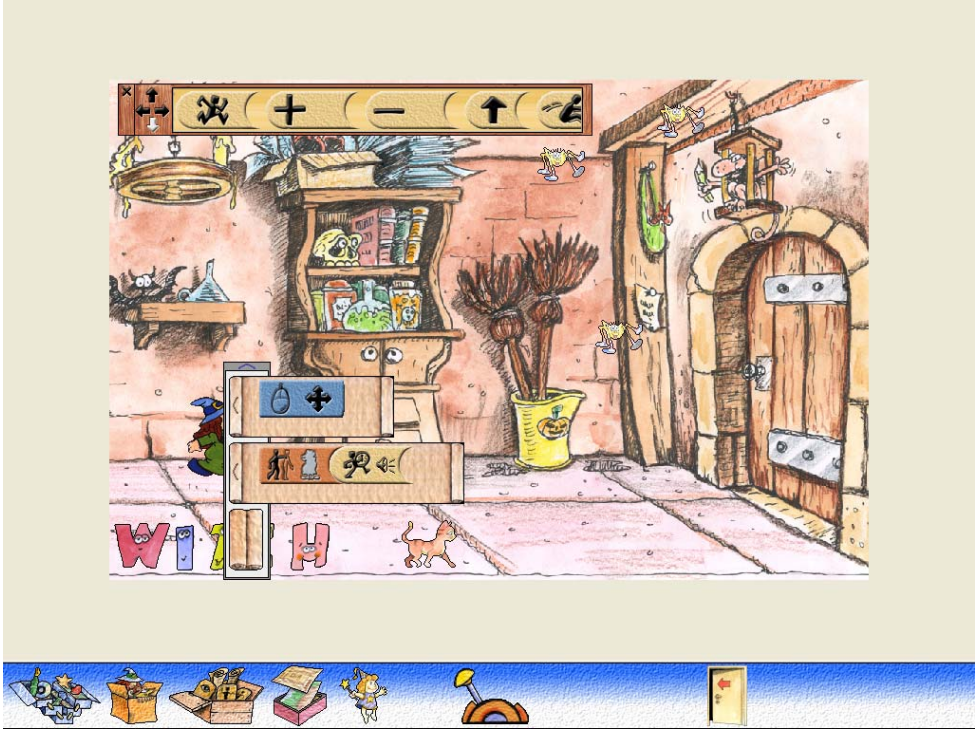


Figure 11.12:Magic Forest allows children to control the actions and appearances of 2D characters. This activity has five characters: a witch, a cat, and three spiders. The witch has two rules controlling her behavior. The top one (blue tile on a scroll) allows the user to move the witch around the scene. The second says that when the witch touches another object, she should make a sound (e.g. laugh). The witch also has an empty scroll to which the user can add new behaviors by selecting events and actions from the brown window at the top of the screen and placing them together on her scroll.

JPie: Washington University, 2003 (Goldman 2003; Goldman 2004)

Particularly at the college-level, there are two competing goals for introductory programming: 1) to introduce students to the ideas and concepts of computer science and 2) to train students in writing programs in a commercial programming language (commonly Java). JPie is a programming environment that allows users to construct Java programs (using any part of the Java 1.4. API) through direct manipulation, primarily drag and drop. JPie is unique in providing not only access to all the programming constructs typically taught in an introductory class, but also providing full access to the Java API rather than providing methods for actors in a micro-world.

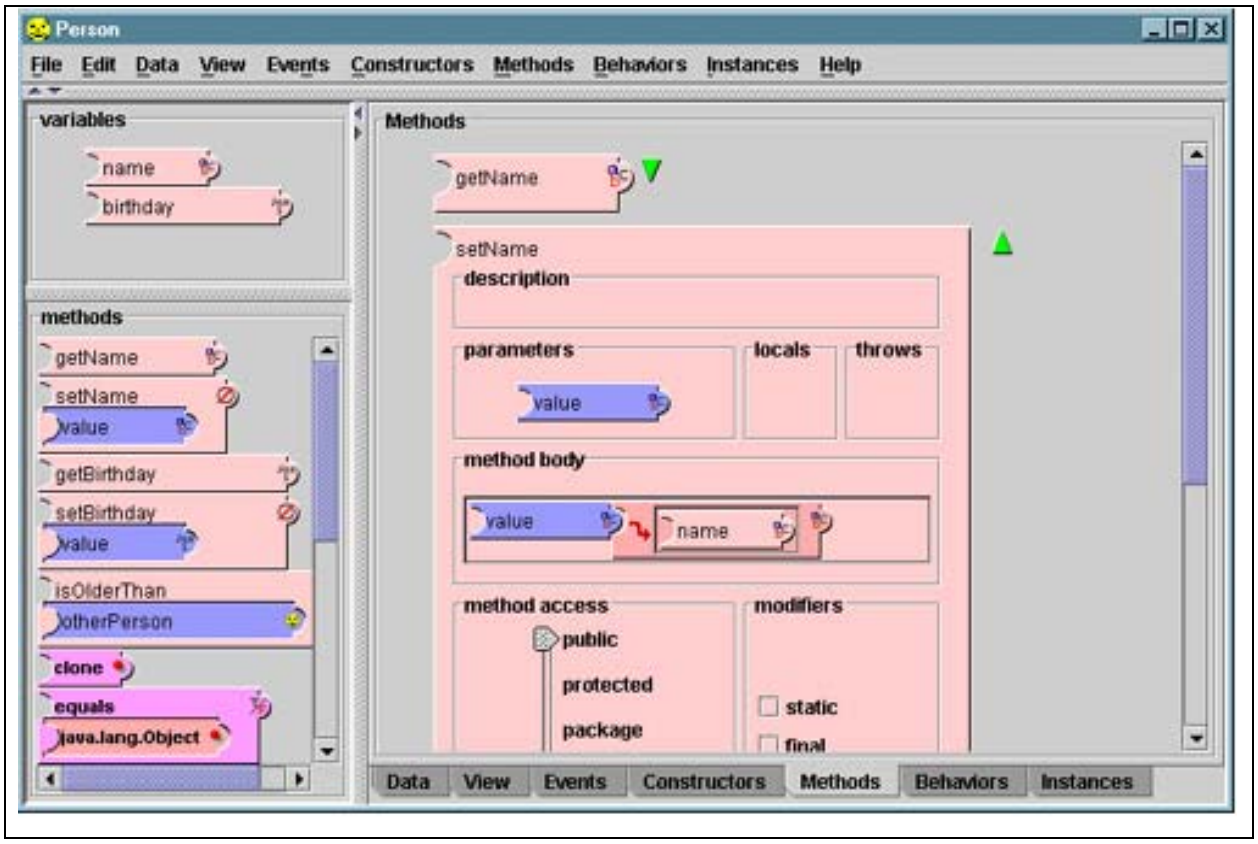


Figure 11.13: An image of the "Person" class within JPie. A person has a name and a birthday as well as methods that for getting and setting the person's name and birthday. In the methods panel, the user is editing the "setName" method which takes a string value and places the value in the "name" variable.

11.3.1.2.2 Create Programs Using Interface Actions

The systems in the previous category used the metaphor of constructing programs by arranging physical or graphical objects, the systems in this category use interface actions (such as button presses or motion through space) or sequences of interface actions as the building blocks of programs. Since most of these interfaces are on physical objects, the interfaces either tend to provide a limited number or commands or require the user to perform interface actions (such as pressing buttons) in a specific sequence, introducing the possibility for sequences of actions that do not correspond to valid program instructions.

TORTIS – Button Box: R. Perlman, MIT Artificial Intelligence Lab, 1976 (Perlman 1976)

The TORTIS Button Box is a physical interface that allows young children to control a robotic turtle inspired by the Logo turtle. The Button Box provides a set of four boxes for controlling the turtle that can be given to a child gradually. The first box provides buttons

that move and turn the turtle, pick up or put down the pen, turn a light on and off, and sound a horn. The second box adds numbers such that a child can repeat a command multiple times by pressing a number followed by a command. The third box adds a program area where children can get the turtle to "remember" commands and then play back remembered commands. The fourth and final box creates four procedures (named by colors) that can call each other. The button box system did not allow students to edit programs after creating them, making the gradual modification of programs difficult.

Roamer: D. Catlin, Valiant Technologies, 1989 (Catlin 1989)

Roamer is a programmable, mobile robot that has capabilities similar to those of the Logo turtle: the Roamer can move forward and back, turn left and right, wait, and make sounds. Programs are entered using a set of buttons, icons for the commands and a number pad to indicate how far to move or turn and what sound to play. Buttons are also provided for creating procedures and repeating statements. The Roamer can remember up to 59 instructions in either the main program (the GO program) or numbered procedures that can be called from the GO program or each other. An expansion set allows users to add on sensors, two-state outputs, and a stepper motor, allowing a greater variety of programs.

LegoSheets: Gindling et al, University of Colorado, 1995 (Gindling, Ioannidou et al. 1995)

LegoSheets attempts to provide a gentle introduction to programming for the MIT Programmable Brick by beginning with manual control of the elements of the brick and gradually progressing to writing programs. Users are presented with a simulated version of the Programmable Brick in which the parts can be manipulated; users can change the speed of a motor connected to the simulated brick by typing in a value or using arrow buttons to increase or decrease the value. Once users are comfortable with manipulating the values of motors and observing the values of sensors in response to different types of actions, they can double click on the representation of a motor or sensor and bring up a rule editor for that object. The rule editor provides buttons to add conditionals or initial values to control the behavior of the brick. Conditionals are provided in a template form where users only have to type the names of objects they want to use and arithmetic

operations. There are also buttons for increasing and decreasing the priority of the current rule.

Curlybot: P. Frei et al, MIT Media Lab, 2000 (Frei, Su et al. 2000)

Curlybot is an educational toy for children aged four years and older. It consists of a two-wheeled vehicle with electronics that allow it to record its motions. The Curlybot has a single button and a single LED. The LED is used to indicate whether it is in record mode (red) or playback mode (green). When a child wants to record a motion, he or she pushes the button, demonstrates the motion, and then pushes the button again, which stops recording and starts replaying the motion. The motion is repeated until the button is pushed again, turning Curlybot off. While Curlybot cannot provide the complexity of a full programming language, it does allow children to gain intuition about repeated motions. The designers describe how sensors could be added to Curlybot to allow children access to if and while statements, but these additions have not been implemented.

11.3.1.2.3 Provide Multiple Mechanisms for Creating Programs

Entering programs as text can be much harder than alternatives such as direct manipulation or form filling but often gives the student more power. In a system that provides multiple mechanisms for specifying programs and represents the resulting program in all program formats, students can use an easier method of program specification to help in learning a more complex, more powerful one. The system in this category provides multiple methods, including standard text, for specifying programs so that students can leverage the simpler methods to learn to program in a standard, textual format.

Leogo: A. Cockburn and A. Bryant, University of Canterbury, 1997 (Cockburn and Bryant 1997)

Leogo (see Figure 11.14) is a system that produces drawings similar to the Logo turtle (see section 4.1.2 under *Make the Language More Understandable*). However, rather than concentrating on one method for creating programs, it provides three: a typed syntax similar to Logo, a direct manipulation interface in which the turtle is dragged around and his actions are recorded, and an iconic language which contains templates for defining

structures and using common turtle commands. Motions are expressed in all code styles simultaneously; when the turtle is dragged forward 15 units, the text window shows forward 15, and the iconic window shows forward 15 in icons so it is possible to learn some of the iconic and typed languages using direct manipulation.

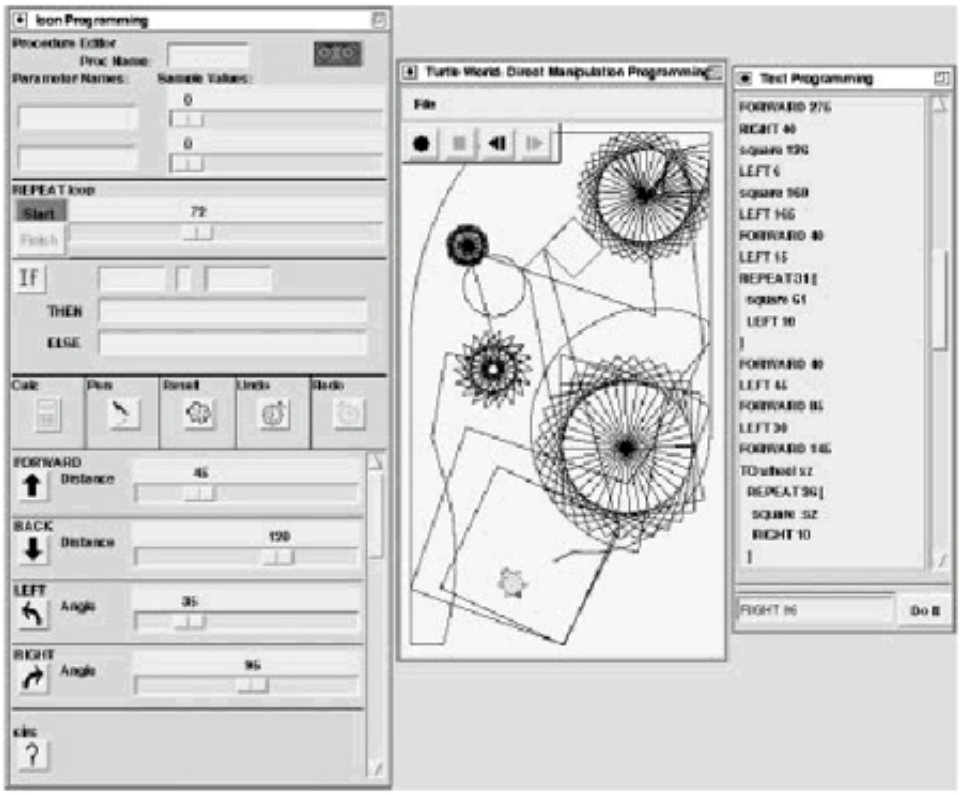


Figure 11.14: The Leogo interface showing iconic, direct manipulation, and textual programming.

11.3.1.3 Structuring Programs

These systems concentrate on the structure of code and how it is organized rather than on the syntax of short segments of code. This section includes systems that have tried *new* paradigms for programming. There are two groups here – ones that are changing the paradigm and ones that are trying to make changed paradigms more understandable

11.3.1.3.1 New Programming Models

These systems explore new paradigms for organizing code.

Pascal: N. Wirth, Institut fur Computersysteme, 1970 (Wirth 1993)

The first version of Pascal was created in 1970 for use in teaching programming, particularly systems programming. At the time, the other available languages were FORTRAN, COBOL, and Algol, none of which supported the Structured Programming

proposed by Dijkstra (Dijkstra 1969). Pascal was introduced in beginning programming classes in 1971 to enable professors to teach Structured Programming to their students in their first course. Although Pascal was designed with teaching in mind, the improvements in the language can be seen as general improvements in programming languages. Algol, one of the primary influences, had ambiguities in the ways nested ifs could be interpreted; Pascal removed these. In addition, Pascal added new basic types and the ability to define special purpose types through record statements.

Smalltalk: A. Kay and A. Goldberg, Xerox PARC, 1971 (Kay 1993)

The first version of Smalltalk was created in 1971 at Xerox PARC as the language for the KiddyKomputer, Alan Kay's original name for a portable computer designed for use by a child. Where BASIC attempted to provide a simpler programming language by reducing the number of commands and removing unnecessary syntax, the Learning Research Group (LRG) at PARC concentrated on the model of programming. The group wanted to create a programming language with a simple model of execution and a method of programming that could accommodate a wide variety of programming styles. Smalltalk was based around three ideas: (1) everything is an object, (2) objects have memory in the form of other objects, (3) and objects can communicate with each other through messages.

Playground: J. Fenton and K. Beck, Apple Computer, 1989 (Fenton and Beck 1989)

Playground is an object oriented programming environment designed to allow children to create their own graphical objects and give them behavior. The programming model was based on a biological metaphor in which all objects are independent "organisms"; the model was influenced both by Minsky's Society of Mind (Minsky 1986) and by classical ethology (the study and description of animal behavior). Each object has its own sensors, effectors, and processing elements so it can act independently. Programming in Playground is rule-based; rules describe both the action and the circumstances under which it should occur. Students specify rules for each object using a natural-language-influenced scripting language. One of the suggested projects for the system is a virtual aquarium with different species of fish and plankton that feed on each other. A fish might

have a rule that caused it to eat an algae cell if it saw one and was hungry. A larger fish might eat a smaller fish.

Kara: R. Reichert, W. Hartmann, J. Nievergelt, M. Braendle, T. Schlatter ETH Zurich, 2001 (Hartmann, Nievergelt et al. 2001)

Kara is a graphical programming language based on Karel the Robot that uses finite state machines to organize procedures (see Figure 11.15 below). Kara can move, turn, pick up and place clovers, and detect tree stumps and clovers; these commands and questions are represented graphically. In each state, the user can ask questions of Kara's current position and, based on the answers to these questions, supply a sequential list of instructions and the name of the next state in the machine. The finite state machine diagram of the program is provided to show the structure of the program and to allow the user to select a pre-existing state to edit. The use of the simple finite machine model for programming allows the Kara environment to be completely graphical; no typing is necessary, which is an advantage for beginning programmers. In addition, to aid the transition from introductory programming in Kara to "real programming" the authors have supplied JavaKara, an environment that provides a transition to Java, MultiKara, an environment that introduces concurrent programming, and TuringKara, an environment that allows students to experiment with Turing machines in a two dimensional plane.

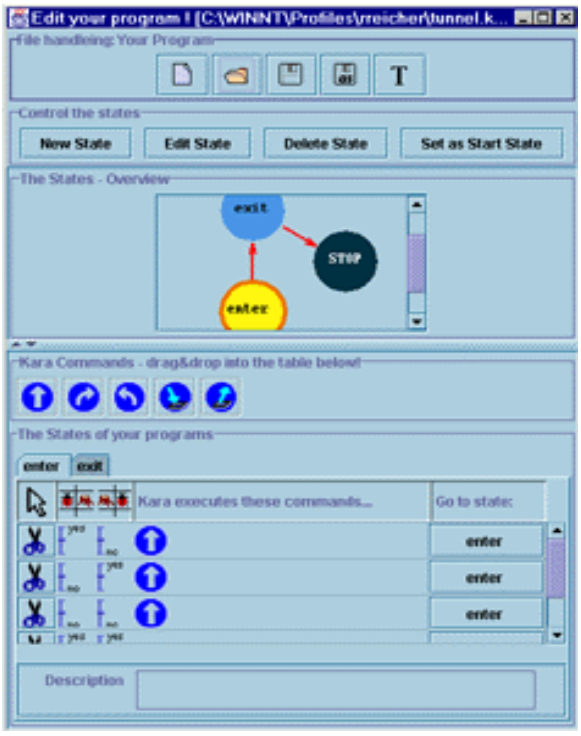


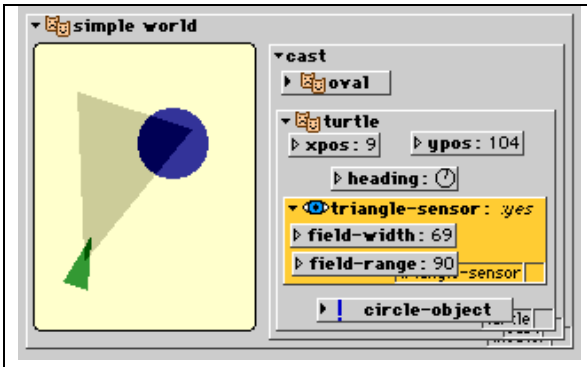
Figure 11.15: A screenshot of Kara showing a finite state machine with three states: enter, exit, and stop. Below the state machine are Kara's instructions based on whether there are tree stumps beside her. Each line contains instructions for a given scenario. For example, if there is a stump on Kara's right and not on her left, she should move forward and go to state enter.

11.3.1.3.2 Making New Models Accessible

Some programming styles, such as object-oriented programming, can be difficult for beginners to understand but can be helpful either in organizing larger programs or representing particular types of behaviors. Rather than requiring novice programmers to learn multiple styles of programming, the systems in this category attempt to make these more complex, but ultimately helpful, styles of programming accessible to novice programmers.

Liveworld: M. Travers, MIT Media Lab, 1994 (Travers 1994)

Liveworld is an object oriented programming environment built to improve on Playground (see section 3.1.2 under *New Programming Models*). In Playground, creating and interacting with graphical elements is very simple, but interacting with the rules and attributes that govern the behavior of the objects is much more difficult. Liveworld attempts to create a graphical interface for the rules and attributes of objects so they are more accessible to novice programmers. The interface is similar to a hierarchical browser (see Figure 11.16 below); parts of objects can be opened, revealing the details of those objects. The user can dive down and change the Lisp code controlling the behavior of objects or simply use the objects, depending upon how much detail the user of the system wants to see. This allows novice programmers to use more complicated objects as black boxes, which would have been difficult in Playground.




```
if (> (ask self :distance-
senser)
        (ask self :last-
distance))
(ask self: turn-left (arand 0
180))
(ask self: turn-left (arand 0
10)))
```

Figure 11.16: (a) A simple world in Liveworld containing two objects, an oval and a turtle. The turtle is open so that the user can see its details. (b) An example of Lisp code used in Liveworld to turn a turtle.

Blue Environment and BlueJ: M. Kolling and J. Rosenberg, University of Sydney, 1996 (Kolling and Rosenberg 1996) (Kolling, Quig et al. 2003)

The Blue environment and BlueJ are development environments designed to support object-oriented programming in the Blue language and Java, respectively. The authors of the Blue environment and BlueJ believe that Integrated Development Environments (IDEs) for object-oriented language should encourage users to develop and test individual classes rather than requiring users to always create complete programs. Yet, most common Integrated Development Environments (IDEs) for object-oriented languages such as Java and C++ still require students to build full programs that have a single entry point. In contrast, the Blue environment and BlueJ provide users with a class-testing bench, which they can use to instantiate individual objects, call their methods, and inspect their internal data. This allows users to test individual objects outside of the context of the running program, better supporting an object-based design. The Blue environment and BlueJ also support object-oriented programming by explicitly representing the relationship between the objects in a graphical tree. Users can click on a particular class to view the code for that class. Compiling and debugging are also supported in the environment, similar to other commercially available IDEs.

Karel++: J. Bergin et al, Pace University, 1997 (Bergin, Stehlik et al. 2001)

Karel J Robot: J Bergin et al, Pace University, 2000 (Bergin, Stehlik et al. 1996)

J. Karel: B. Becker, University of Waterloo, 2004 (Becker 2004)

Karel J Robot, J.Karel, and Karel++ are versions of Karel the Robot that concentrate on preparing students for object-oriented programming rather than procedural programming. Karel J Robot and J Karel use Java-style syntax; Karel++ uses C++ style syntax. Rather than creating procedures to teach Karel to turn right, students subclass a basic robot to create a right-turning robot. These systems all leverage off the success of the original Karel the Robot to attempt to introduce object-oriented programming early such that thinking and programming in an object-oriented manner will seem more natural to students.

11.3.1.4 Understanding Program Execution

A syntactically correct program may not perform the actions that the student author intended. For beginning programmers, understanding how programs are executed and how to find mistakes in their programs can be difficult. The systems in this category try to help students understand what happens during the execution of programs, either by placing programming into a concrete setting or by providing a physically based model of how programs are executed in more general-purpose languages.

11.3.1.4.1 Tracking Program Execution

Atari 2600 BASIC: W. Robbinett, Atari, 1979 (Robinett 1979)

The Atari BASIC Cartridge allowed children to write short programs in a variant of the BASIC language and watch them as they executed. Atari BASIC divided the screen into six regions: the Program region, which displayed the child's program; the Stack region, which displayed expressions as they were evaluated; the Variables region, which displayed each variable and its current value; the Output region, which displayed all program output; the Graphics region, a 2D graphical region with sprites; and the Status region, which displayed the current execution speed of the interpreter and the amount of remaining memory. Atari BASIC contained simple support for observing what was happening as the program executed, similar to the supports found in many debuggers. As a child's program ran, several parts of the display changed to reflect the current state of the program: a program cursor showed the current line of code being executed; the stack updated as expressions were added or evaluated; the values of variables changed as appropriate; sprites might move in the graphics region; and the program might play a sound.

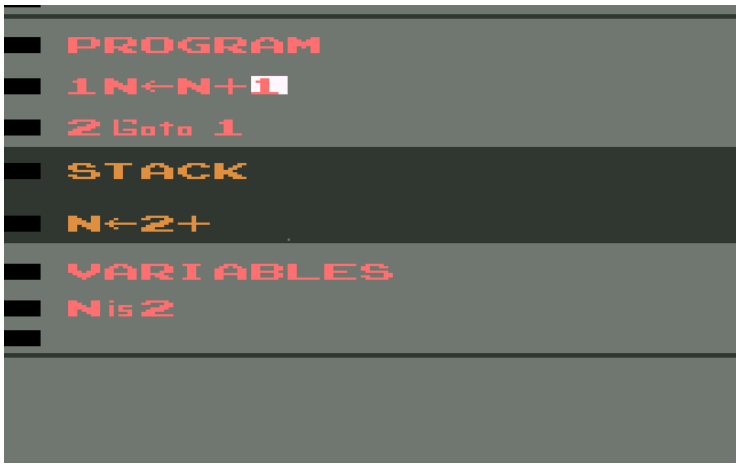


Figure 11.17: A simple program in Atari 2600 BASIC. The areas of the screen update to show the current position and state of the program.

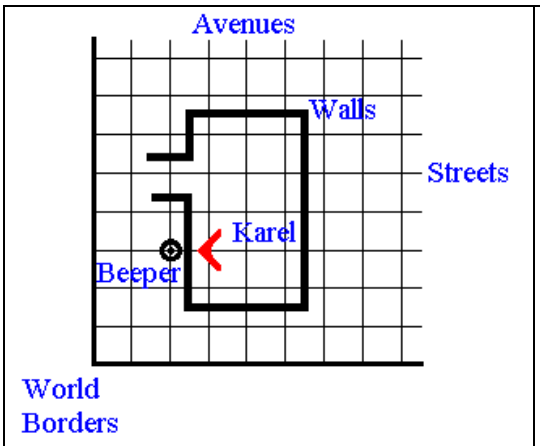
11.3.1.4.2 Make Programming Concrete: Actors in Microworlds

Most introductory programs in general-purpose languages are fairly abstract; the computer performs arithmetic operations on numbers and stores the results in invisible registers, making it difficult for students to understand and correct problems in their programs. The micro-world, inspired by the Logo turtle (see section 4.1.2 under *Make the Language More Understandable*), attempts to make programming more concrete by introducing students to programming constructs through controlling the behavior of an actor in a simple, physically based world. The actors usually perform only a few actions, resulting in small languages that students can master more quickly than general-purpose languages. Micro-world based systems also typically include simulators that allow students to watch the progress of their programs. These simulators require the states of micro-worlds to be graphically visible. Using micro-worlds, students can quickly gain familiarity with many of the control structures like if-statements and loops, allowing them to devote more time and energy to mastering the syntax and new commands when they move on to general-purpose languages.

Karel: R. Pattis, Carnegie Mellon University, 1981 (Pattis 1981)

Karel the Robot is one of the most widely-used mini-languages, originally designed for use at the beginning of a programming course, before the introduction of a more general-purpose language. Karel is a robot that inhabits a simple grid world (see Figure 11.18) with streets running east-west and avenues running north-south. Karel's world can also

contain immovable walls and beepers. Karel can move, turn, turn himself off, and sense walls half a block from him and beepers on the same corner as him. A Karel simulator allows students to watch the progress of their programs step by step. Unlike many of the systems discussed in this paper, Karel is supported by a short textbook, making it easier for teachers to incorporate Karel in their classes.




```
BEGINNING-OF-PROGRAM
  DEFINE-NEW-INSTRUCTION turnright AS
    ITERATE 3 TIMES
      turnleft;

  BEGINNING-OF-EXECUTION
    turnright;
    ITERATE 2 TIMES
      move;
    turnleft;
    ITERATE 2 TIMES
      move;
    turnleft;
    ITERATE 2 TIMES
      move;
    turnleft;
    move;
    pickbeeper;
    turnoff;
  END-OF-EXECUTION
END-OF-PROGRAM
```

Figure 11.18: Left, a simple Karel world with Karel in a room and a beeper outside the door. On the right, a program that will move Karel to the beeper's location and have him pick up the beeper.

Students can create procedures using DEFINE-NEW-INSTRUCTION (see Figure 11.18), but variables and data structures are not supported in the language. The syntax was designed to be similar to Pascal (see section 3.1.2 under New Programming Models) to ease the transition from Karel to Pascal after the first few weeks of an introductory programming course. There are a number of other robot-based micro-worlds that are described in a survey of mini-languages (Brusilovsky, Calabrese et al. 1997).

Josef the Robot: I. Tomek, Acadia University, 1983 (Tomek 1983)

Like Karel, Josef is intended to introduce programming to beginners using a robot, Josef, in a simulated world. Josef lives in Wolfville, which is represented by an ASCII map; users can replace the map of Wolfville with one of their own choosing. He knows how to turn left and right, and move forward. The user can also set the speed at which Josef moves. However, unlike Karel, Josef can say and listen for text strings, enabling input -

output programs. Additionally, he can drop text markers (e.g. the string "cat") similar to Karel's beepers anywhere in his world. Unlike Karel, Josef was intended for use in a full semester of programming for non Computer Science majors. To support a full semester of use, it includes many more programming constructs than Karel, such as parameters, variables, and recursion.

Turingal: P. Brusilovsky, University of Pittsburgh, 1991 (Brusilovsky 1991)

Turingal is micro-world based language in which the actor is a Turing machine and the world is the infinite tape designed to give students exposure to the standard programming constructs as well as the classic Turing machine. The instructions in the language allow the actor to move left and right along the infinite tape as well as read and write symbols on the tape. Like Karel, the basic instructions are easy to visualize. The Turingal language supports conditional, loop and case statements and procedures so that students can gain experience with them in a visual setting. The language uses Pascal syntax (see section 3.1.2 under New *Programming Models*) to ease the transition from Turingal to Pascal. In support of a computer literacy course for Russian high school students, Brusilovsky also created Tortoise, a micro-world based on Turingal which uses a two-dimensional field of symbols to make it more attractive to younger students (Brusilovsky, Calabrese et al. 1997).

11.3.1.4.3 Models of Program Execution

Rather than creating a language that has a simple, physical interpretation, the systems in this category provide physically based metaphors for explaining actions in a more general-purpose language. These metaphors can help students both to imagine the execution of their programs and perhaps more clearly understand why their programs do not perform as expected.

ToonTalk: K. Kahn, Animated Programs, 1996 (Kahn 1996)

ToonTalk uses a physical metaphor for program execution. In ToonTalk, cities and the creatures and objects within those cities represent programs (see Figure 11.19). Most of the computation takes place inside of houses where trainable robots live. Robots can communicate with robots in other houses using birds that carry objects back to their nests. Using interaction techniques commonly found in videogames, users can navigate

around the cities, pick up tools, and use those tools to affect objects. Users can construct programs by entering the thought bubbles of robots and showing them what they should do using standard ToonTalk tools.

Figure 11.19: A view of ToonTalk from inside a house. Marty the Martian provides information about objects and what they can do.

Prototype 2: D. Gilligan, Victoria University, 1998 (Gilligan 1998)

Prototype 2 personifies the flow of control in a computer using a clerk following instructions. The clerk can interact with calculators, I/O devices, worksheet machines, and his clipboard in executing a program. Calculators represent the computer's math processor, I/O devices represent communication with the computer user, the clipboard represents the program stack, and the worksheet machines produce stacks of worksheets that represent the instructions in user-defined subroutines. Rather than imagining the internals of a computer, a novice programmer can imagine the clerk walking around a room interacting with calculators, I/O devices, worksheet machines, and his clipboard, and executing the instructions specified on his clipboard. This model was used in the creation of a programming by demonstration-based system in which the user plays the part of the clerk and demonstrates the actions the clerk should take. The system records these actions. While Prototype 2 uses an anthropomorphic metaphor, the system does not

include a graphical representation of the clerk and the objects in his world; instead it is a standard graphical user interface with sections of the interface that represent each of the objects in the clerk's world (e.g. the calculator, I/O devices, etc.) that the novice programmer can use to demonstrate how the clerk should behave.

11.3.2 Learning Support

Systems in the previous category examined ways to make the process of learning to program easier by simplifying the mechanics necessary to write a program. The systems in this category try to ease the process of learning to program by providing basic educational supports such as progressions of projects that gradually introduce new concepts or ways for students to connect with and learn from each other.

11.3.2.1 Social Learning

Some of the most effective learning is done in a social context where more than one person is working with a problem. Since programming is known to be hard and children often learn more effectively in groups, perhaps it may help the learning process to provide a social context in which learning can occur. The systems in this category investigate different methods for allowing students to work together: co-located and over a network connection.

11.3.2.1.1 Side By Side

Most computer interfaces are designed for single users. Consequently, when groups of children use a standard mouse, monitor, and keyboard setup in learning, one child tends to dominate the process. The systems in this category use tangible interfaces to allow multiple students in informal groups to work together in solving programming problems. Because of the difficulty of representing the wide variety of programming constructs in a tangible form, these systems concentrate on small subsets of programming.

AlgoBlock: H. Suzuki and H. Kato, NEC Information Technology Research Laboratories, 1995 (Suzuki and Kato 1995)

The authors of AlgoBlock wanted to create an active learning community among children learning to program in which children can share notes and techniques, and learn from each other. They created AlgoBlock, a set of blocks, each of which corresponds to a simple command in Logo. The blocks can be connected together to form programs that

control the movements of a submarine in a maze. The blocks are tangible and large enough that they can be arranged on a desk that several students can work around. This allows students to work with the blocks in a social context, learn from each other, and communicate what they are learning. The tangible nature of the blocks made it easy for children to take turns manipulating the blocks and communicate about which pieces should be placed where. The AlgoBlock project demonstrates that, in a suitable environment, children will work together in building programs. However, the blocks supported a limited set of programming constructs; the children were not able to explore concepts like procedures, parameters, or control structures.

Tangible Programming Bricks: T. McNerney, MIT Media Lab, 2000 (McNerney 2000)

Tangible Programming Bricks are physical Lego blocks that can be stacked together to form programs. The designer's intent in creating these was to provide a simple interface to appliances and toys and to create a programming environment that would allow children to collaboratively explore ideas. While the work concentrated on the hardware implementation of the Lego blocks, the designer created three prototype environments using Lego blocks that represent commands. To allow a greater variety of commands, users could insert a small card (e.g. microchip) into a block. Each block could accept a single card, allowing users to communicate with other blocks via IR transmission, supply parameters to commands, sense the environment, or display variables. The three prototype languages allowed children to teach toy cars to dance, kitchen users to program microwaves, and toy trains to react to signals along the side of the tracks in unique ways. By stacking blocks together with accompanying cards, if necessary, users could construct simple programs.

11.3.2.1.2 Networked Interaction

Rather than trying to move away from the common single user, single computer paradigm, the systems in this category attempt to allow students using different machines to work together over the network. While the systems designed for students working side-by-side can assume all children can see the state of the current program and what other

children are doing, programming systems designed for network use need to explicitly support the exchange of this kind of information.

MOOSE Crossing: A. Bruckman, MIT Media Lab, 1997 (Bruckman 1997)

MOOSE Crossing is a networked programming environment built for children. It is an adapted text-based MUD (multi-user dungeon) in which children can use an object-oriented scripting language to create spaces and characters that inhabit a textual world (see Figure 11.20). Children often create spaces and characters similar to those found in text adventure games such as castles complete with secret passages that other children can explore. Once their projects are completed, any child in the MOOSE Crossing environment can interact with them. In addition, the environment allows children to view the scripts controlling any object or character in the environment and chat with children that are currently logged onto MOOSE Crossing. In general, children work alone on projects but one child will often use another child's project as an example. Children may also ask another user for help or advice. The MOOSE Crossing community has provided a source of help, role models, and positive feedback for users of the system as they create their own projects.

```
on pet this
   tell player "You pet Rover."
   if player member_of my friends
        emote "wags his tail."
end
```

Figure 11.20: A MOOSE Crossing script that allows MOOSE users to pet Rover. When a user pets Rover, they are told "You pet Rover." If they are one of Rover's friends, then Rover wags his tail.

Pet Park: A. DeBonte, MIT Media Lab, 1998 (DeBonte 1998)

Pet Park is an exploration of the ideas of MOOSE Crossing in a 2D graphical domain rather than a textual one. Children can choose one of 5 dogs to be their pet. Each dog comes with a few animations, such as wagtail, jump, walk, laugh as well as basic ones like wait, turnLeft, say, etc. Users can combine these simple commands to create their own animations using a textual scripting environment or a set of graphical blocks representing each command. As in MOOSE Crossing, Pet Park is a networked programming environment in which children can talk, ask each other for help, and show off their creations. While in MOOSE Crossing, children create spaces by describing them with text; in Pet Park, creating a space requires graphical objects. In response, the system

provides a variety of furniture, objects, and rooms. Furniture and rooms can be programmed to react to simple events such as avatars coming near them.

Cleogo: A. Cockburn, University of Canterbury, 1998 (Cockburn and Bryant 1998)

Cleogo is a networked version of Leogo (described earlier) that allows children to see and interact with the same Leogo workspace. Rather than concentrating on building a community of programmers, Cleogo creates a shared environment, the current program being edited, and allows multiple children to see and manipulate that environment. Cleogo does not attempt to provide children with a way to communicate with each other about their project. Instead, it assumes that they are either in the same room or can talk to each other using the phone or some equivalent.

11.3.2.2 Providing a Motivating Context

Motivation can be a key element in learning; if students want to accomplish a particular goal, obstacles they encounter while learning to program will not deter them as much. The systems in this category attempt to provide beginning programmers with goals to achieve through programming that the designers believe novice programmers will find motivating.

Rocky's Boots / Robot Odyssey: W. Robbinett, The Learning Company, 1982 (Robinett and Grimm 1982)

Rocky's Boots was one of the first educational software products for personal computers to successfully use an interactive graphical simulation as a learning environment. The game allows children to connect logic gates (AND, OR, NOT and flip-flop) together to create circuits using a joystick (see Figure 11.21). When the circuits are active, users can watch the wires turn from white to orange as the electricity passes through them. The game provides a series of puzzles of increasing difficulty in which the player is supposed to separate the shapes matching a certain criteria from those that do not using logic gates, sensors that can detect certain kinds of shapes, and a boot that, when activated by a true value, kicks the current shape out of the line and off to one side. Robot Odyssey follows the same basic pattern; the player connects gates together to solve problems. However, Robot Odyssey includes a larger selection of objects that perform animated actions when

they are activated (like the shape-kicking boot), creating a wider set of possibilities for the behaviors of circuits.

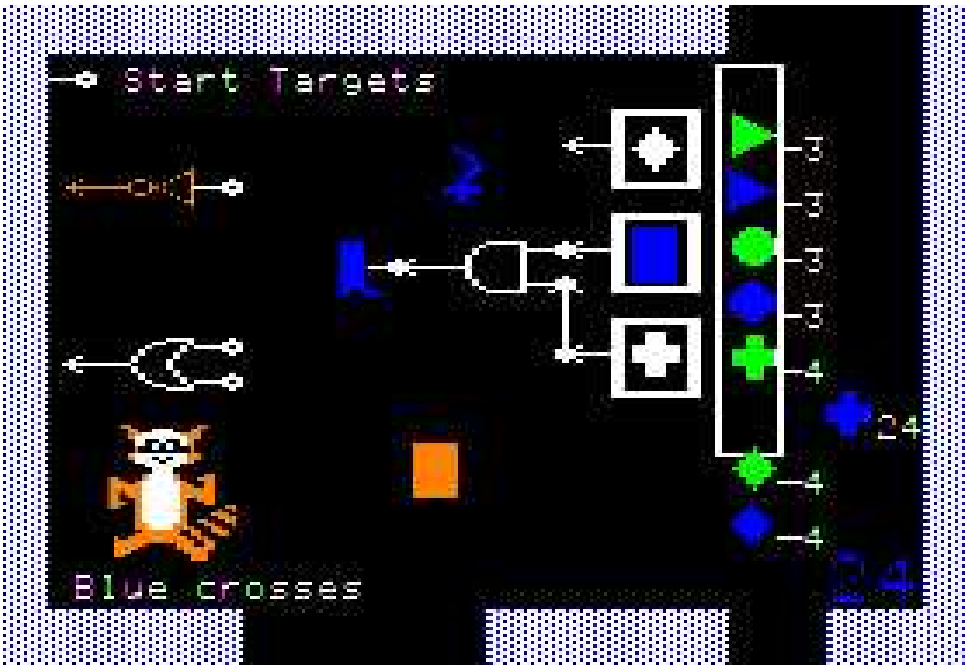


Figure 11.21: A puzzle from Rocky's Boots in which the player is asked to create a circuit that separates blue crosses from the other shapes. When the circuit is switched on, shapes move up the right side of the screen. When they enter the white rectangle, the shape sensors to the right of the rectangle can detect them. The player is asked to attach a sequence of logic gates to the sensor that will activate the boot (center) when a blue cross enters the box. The boot, when activated, will kick the shape out of the rectangle.

AlgoArena: H. Kato and A. Ide, NEC Information Technology Research Laboratories, 1995 (Kato and Ide 1995)

In AlgoArena, players write programs to control the behavior of sumo wrestlers fighting tournaments. The programs are written in a language based on Logo. When a player has completed a program, the player can log onto a website and have his or her wrestler fight against another student's wrestler. Over time, by analyzing the circumstances in which the player's sumo wrestler loses tournaments, the player is expected to learn more complex ways to control the wrestler, perhaps querying the position and posture of their opponent before deciding which moves to execute.

Robocode: M. Nelson, IBM Advanced Technology, 2001 (Nelson 2001)

Robocode is designed to help novices learn Java through programming a robotic battletank for a "fight to the finish". The tutorial teaches novices to subclass an existing battletank robot and extend the robot's capabilities using standard Java and a set of

classes written for the Robocode environment. Upon completion of a robot, users can upload their creation to a number of websites or join a robotic battle league. The designer of the system believes that the ability to program robotic battles will provide enough motivation to get a novice programmer over the hurdles of beginning to program.

Scratch: M. Resnick et al., MIT Media Lab, 2006 (Maloney, Burd et al. 2005)

Scratch was designed to introduce programming to students in after-school computer centers. Over the past ten years, approximately 2000 community technology centers (CTCs) have opened throughout the United States. The goal of these CTCs is to provide access to technology in economically-disadvantaged communities. However, while many of the CTCs have developed communities of children interested in creating digital art using Photoshop, few of the CTCs have developed a community of children who are interested in computer programming. Because computer programming has the potential to help children develop the ability to think about the potential for and challenges associated with technology in our current society. Scratch is designed to leverage childrens' motivation to use and manipulate digital images (as they do in Photoshop) to introduce programming. Users can import digital images to Scratch and create programs that animate and modify (similar to photoshop filters) those images. Users construct programs in Scratch by dragging and dropping blocks that represent program elements, similar to the method of program construction used in LogoBlocks.

Figure 11.22: A View of the Scratch interface. In the left-most panel are the blocks (commands, functions, control structures, and variables) that users can use in their programs. The center panel is the scripts panel, where users can compose their programs. The right-most panel shows the 2D world that the user's program controls.

RAPUNSEL: K. Perlin, M. Flanagan, J. Plass, A. Hollingshead., NYU and Hunter College, 2005 (Flanagan, Nissenbaum et al. 2005)

The goal of the RAPUNSEL project is to develop a programming environment that makes learning to program an engaging activity for girls. In working with inner-city teenagers at a computer center, the RAPUNSEL investigators created and tested a variety of prototype programming environments. The development and testing of these prototypes culminated in a programming-based game based in an imaginary world inhabited by Peeps. Girls teach their Peep characters new dance moves by writing programs in Java (a sequence of tutorials guides them through this process).

Figure 11.23: A screenshot of a RAPUNSEL prototype.

11.4 Empowering Systems

The systems in this category are built with the belief that the important aspect of programming is that it allows people to build things that are tailored to their own needs. Consequently, the designers of these systems are not concerned with how well users can translate knowledge from these systems to a standard programming language. Instead, they focus on trying to create languages and methods of programming that allow people to build as much as possible.

11.4.1 Mechanics of Programming

The systems in this category are designed around the hypothesis that the primary barrier for people attempting to use programming as a tool is the mechanical difficulties of creating programs. Systems in this category examine ways of improving programming languages and alternative ways for creating programs.

11.4.1.1 Code Is Too Difficult

Many researchers have examined the problem of making languages more understandable and usable for novices. While progress has been made making programming languages more understandable, there still are many barriers for novices trying to build their own

programs. These systems examine creating programs either through demonstrating correct behavior or selecting actions through the interface.

11.4.1.1.1 Demonstrate Actions in the Interface

The systems in this category examine ways that users can program a system by showing the system what to do through manipulating the interface, without relying on a programming language.

Pygmalion: D. Smith, Stanford University, 1975 (Smith 1993)

Pygmalion was the first programming by demonstration system. Unlike many of the systems that came after it which concentrated on graphical objects, Pygmalion attempted to get people to write more abstract programs, such as a program to compute the factorial of a number. However, rather than building factorial by typing statements in a programming language, Pygmalion relied on editing an artifact. To create a factorial program, the user creates an icon with two sub-icons, one for the input and one for the output, and draws a symbol to represent factorial. The user can then enter remember mode, in which all of the actions made by the user are remembered by the system. Consequently, the user can program the computer by working out an example of how to compute factorial. However, the user must anticipate the handling of the value one and test whether or not the current value, say three, is equal to one, something that novices may not be well prepared to do. If the user does not demonstrate his or her current actions as the case for the current value not being equal to one, Pygmalion will not know that one should be handled differently and, consequently, will not prompt the user to demonstrate how one should be handled.

Programming by Rehearsal: W. Finzer and L. Gould, Xerox PARC, 1984 (Finzer and Gould 1984)

Programming by Rehearsal was built to help non-programmers create educational software. It is designed around a theater metaphor in which components of the interface are performers that interact with one another on a stage by sending and responding to cues. A user of the system would begin creating a piece of software by auditioning performers to use as building blocks, selecting their cues via a pop-up menu and observing their responses to those cues. The user would then copy the chosen performers

onto the stage, placing and sizing them appropriately. The rehearsal portion of development consists of showing the performers what actions they should take in response to user input or cues sent by other performers. Objects that accept user input, such as buttons, have cue sheets that allow users to fill in their responses to those user inputs. Users can press a closed eye icon to tell the system to begin observing their actions. Then, by selecting cues from the menus of other performers, they can show the system how to react to those cues. By pressing the eye icon again, users indicate they have finished. The system comes with 18 basic performers that users can audition and use in their own creations. Additionally, the system allows users to create new performers by combining existing performers and teaching them new cues. While Programming by Rehearsal does allow users to access the underlying programming languages (Smalltalk), the system was designed to allow non-programmers to create educational software without requiring them to program at the Smalltalk level.

Mondrian: H. Lieberman, MIT, 1992 (Liebermann 1993)

Mondrian is a programming by demonstration system for drawing and graphical editing in which commands are shown with "domino" icons that depict the before and after states for that command. To execute a command, users select the command icon and select the object or area to which the command should be applied. The user can create new commands in a storyboarding style by showing how to do each step in the new command. These steps are displayed at the bottom of the screen in comic book format with a short caption describing each step. Drawing a rectangle on the screen would show a box with the new screen state captioned by "rectangle". If the user then moves the rectangle, a "move" domino would appear beside the "rectangle" domino in the definition of the new command. New commands created by the user are displayed in the same domino style as the commands built into the system. In addition, the system provides speech synthesis capabilities to give an English description of what a command does.

11.4.1.1.2 Demonstrate Conditions and Actions

Like the previous category, the systems in this category try to avoid forcing users to express their intentions in code. However, instead of demonstrating programs by performing actions in the user interface, as the systems in the previous category did, the

systems in this category allow users to depict the conditions in which they want the program to perform an action and the results of that action.

AgentSheets: A. Repenning, University of Colorado, 1991 (Repenning 1993; Repenning and Ambach 1996)

In AgentSheets (see Figure 11.24), users can create simulations by specifying the behavior of sprites in a 2-dimensional grid-based world. Sprites can move to new grid positions, make sounds, and change appearance. Users can create programs using graphical rewrite rules; users select conditions (configurations of icons in the world or relative to each other) and show the system what should happen under these conditions by moving the agents to their new positions. In addition, Agentsheets provides tools for creating analogies between agents. For example, if a user wants a train to follow a set of train tracks in exactly the same way that a car follows roads, he or she can use an analogy tool to easily specify this. Use of analogies provides an easy way to reuse code.

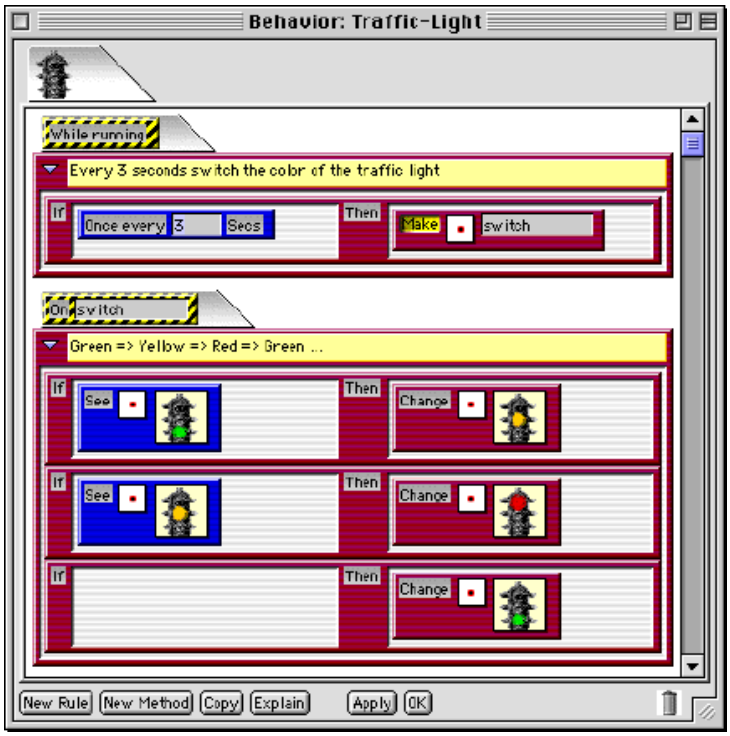


Figure 11.24:A screenshot of a traffic light simulation in AgentSheets containing two rules. The first rule runs continuously: every three seconds it triggers the second rule. The second rule looks at the current color of the traffic light and changes it to the next one in the sequence green, yellow, red.

ChemTrains: B. Bell and C. Lewis, US West Advanced Technologies, University of Colorado, 1993 (Bell and Lewis 1993)

ChemTrains is a pictorial rule-based language that attempts to make it easy for people to create a wide variety of "behaving pictures". ChemTrains is similar to Stagecast (see below) in that users show both the conditions and results of a rule through pictures. In ChemTrains the pictures used to specify conditions and results are interpreted as patterns of connections rather than collections of pixels. For example, in simulating an AND gate, if there is any box with a zero connected to the AND gate (from any direction and any distance away), the output of that gate should become zero. A similar statement in Stagecast would only work if the zero connected to the AND gate was always in the same relative position to the AND gate. As in Stagecast, the order of the ChemTrains rules dictates how they are applied; only the first matched rule is applied in each time slot. Additionally, the ChemTrains pattern matcher can use variables; in ChemTrains, variables are specially marked pictorial elements that can match any element of the simulation display. The addition of variables allows users to create a wider range of simulations.

Stagecast: D. Smith, A. Cypher, and J. Spohrer, Apple Computer, 1995 (Smith, Cypher et al. 1994)

Stagecast, a commercial version of KidSim (see Figure 11.25), is an environment for creating simulations. Children are presented with a grid-based world in which they can create their own actors. Users define rules for the simulation by selecting a before condition from the grid world and then demonstrating how that condition should change (see Figure 11.25). When the simulation is started, when a section of the grid matches a condition of one of the rules, the rule is applied. Stagecast applies only the first rule (in top to bottom order) that matches a section of the grid.

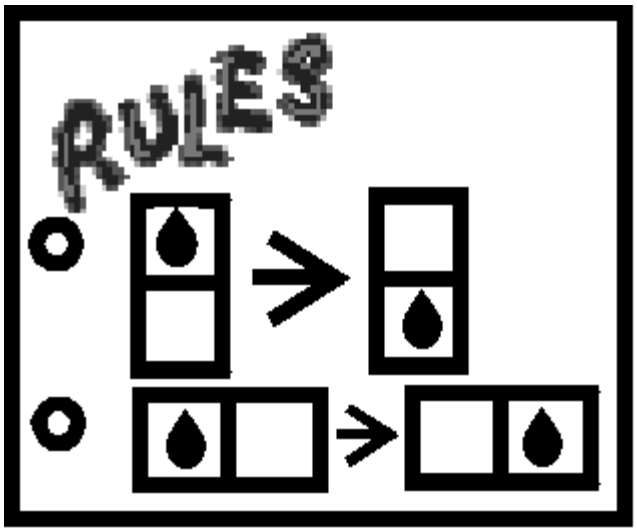


Figure 11.25: This drawing shows an example of how users create rules in Stagecast. On the left side are the conditions in which each rule should be applied. On the right, the results of each rule are shown. In this drawing, if there is a raindrop with an empty space between below it, the raindrop should move down. Otherwise, if there is a raindrop with an empty space on its right, it should move right.

11.4.1.1.3 Specify Actions

In these systems, the user creates programs by using the interface to specify the desired behavior. The user does not see any code, but unlike in programming by demonstration systems, the user does not show the computer what to do, he or she selects the program's actions.

Alternate Reality Kit: R. Smith, Xerox PARC, 1987 (Smith 1987)

The Alternate Reality Kit (ARK) is an environment in which users can build interactive simulations. Users interact with objects built on a physical-world metaphor; each object has an image, position, velocity, and can be influenced by forces. Users can pick up objects, move them, drop them, or throw them using mouse gestures. Users can query or change the state of objects by sending messages, represented by buttons, to those objects. To connect a button to a particular object, the user drops the button onto that object. If the object understands the message the button represents, the button "sticks" to the object, otherwise it falls through. Buttons that require a parameter have a little "plug" where users can hook up a value for the parameter.

Klik N Play: F. Lionet and Y. Lamoureux, Europress, 1994 (Lionet and Lamoureux 1994)

Klik N Play is designed to allow the user to create simple level-based games. The application has three modes: a storyboard editor, which allows the user to see all levels as thumbnails, a level editor, and an event editor. The level editor allows the user to select the background, add predefined objects to the level, and provides users with the ability to create their own objects and animations for those objects. Users create animations frame by frame with a bitmap editor and use controls to set the speed and motion of objects. The event editor uses a table format and allows the user to specify actions for a variety of predefined events (see Figure 11.26). Klik N Play's events are based on collisions between objects, mouse and keyboard input, time, the state of players, and the states of variables and objects in the level. Corel distributed an updated version of Klik N Play that granted users the rights to sell their games under the name Click and Create.

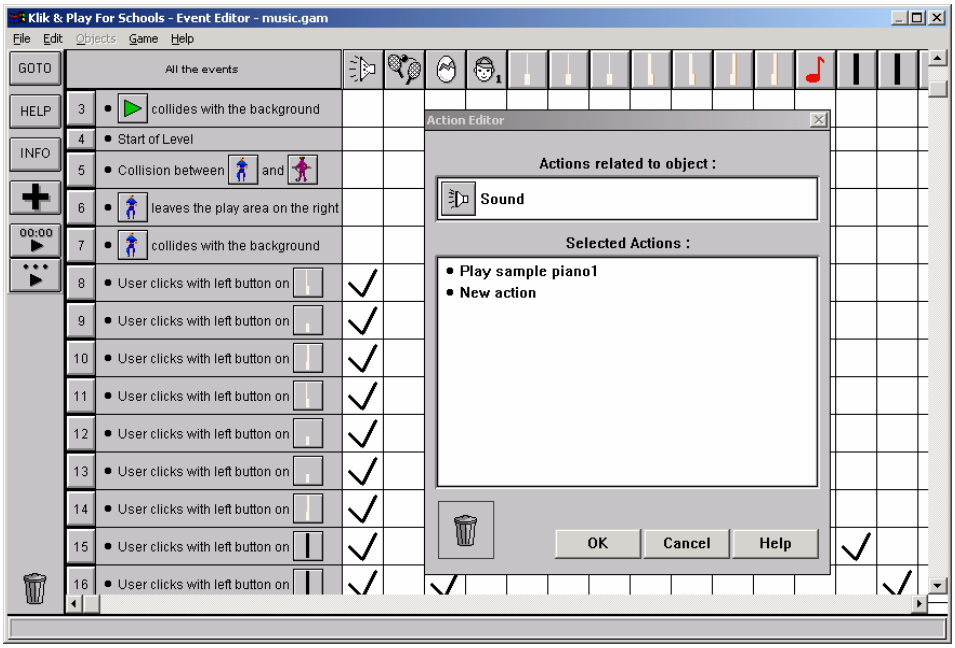


Figure 11.26: A view of the event editor in Klik N Play while the user builds a graphical piano program. The user is currently specifying that when the "User clicks with left button on white piano key," the game should play "sample piano1." The events are organized in table form based on their effects: all sound events are in the first column, events on the user's objects, piano keys in this screenshot, begin at column 5.

Emile: M. Guzdial, University of Michigan, 1995 (Smith 1987; Guzdial 1995)

Emile is a programming environment written in Hypercard(Goodman 1987) that allows high school aged students to create physics simulations (see Figure 11.27 below). The environment provides support or scaffolding (Merrill and Reiser 1993) that makes the process of programming (everything from defining the problem and breaking it into goals to defining the behavior of a button within the interface) easier for beginning students. As

the students become more comfortable with the environment, they can choose to use less support. In Emile, beginning programmers create programs by assembling components: buttons, textfields, and predefined actions. Using menus and dialog boxes, students can select one or more actions that should happen when a given button is pressed and fill in any necessary parameters for those actions. As they become more advanced, students may begin to use mathematical expressions, create their own actions by combining other actions, and eventually edit HyperTalk (Hypercard's scripting language) code themselves.

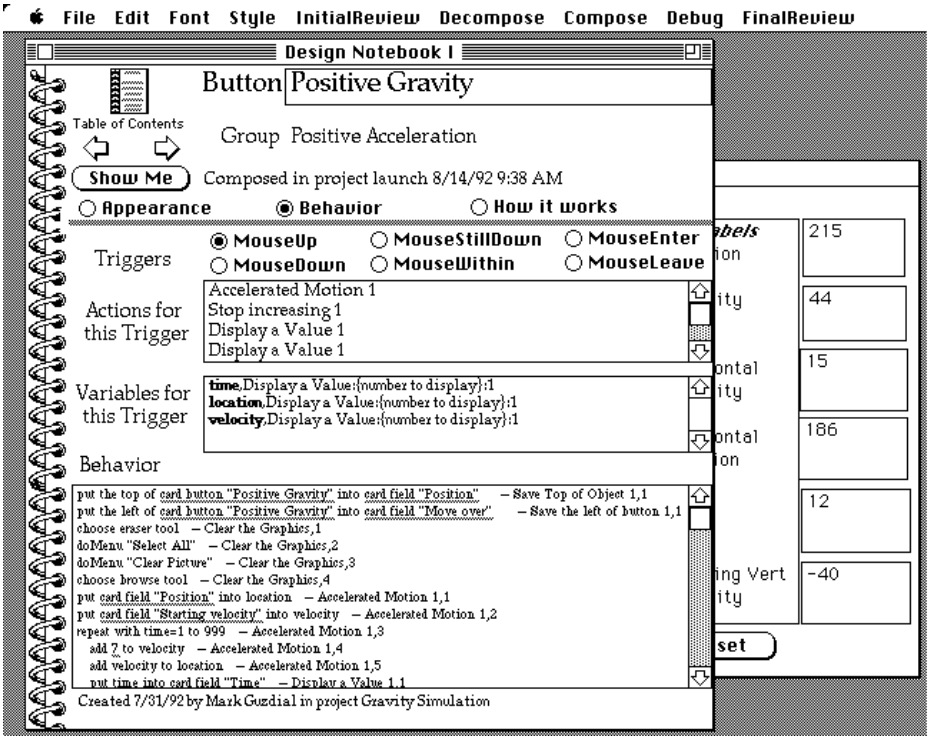


Figure 11.27: An editor for the Positive Gravity Button. When the mouse goes up, Emile will execute 4 actions: Accelerated Motion 1, Stop Increasing 1, and Display a Value 1 (2 times). At the bottom of the screen, we can see the code that Emile will execute. Underlined text corresponds to parameters (or slots) that the user can fill in using menu options and dialog boxes.

11.4.1.2 Improve Programming Languages

The designers of many of the teaching languages are concerned with how well students can transfer the knowledge they gain in the teaching language to more general-purpose languages. Consequently, the designers of teaching languages have been hesitant to deviate very far from these general-purpose languages. However, the systems in this category endeavor to empower their users to create interesting programs; whether the

users of these systems can transfer their programming knowledge to more general-purpose languages is not important. Consequently, the designers of these systems can make changes to standard programming languages that the authors of teaching languages might hesitate to make.

11.4.1.2.1 Make the Language More Understandable

These systems include languages that were developed with a focus on the language and words novices use to describe situations. Most previous languages have been developed with a focus on consistency between languages or on mathematical simplicity. These languages instead focus on choosing words that the users of the system understand and can use effectively without having to translate their words in their everyday vocabularies into the words that the computer language uses for the same concept.

COBOL: C. Phillips et al, Department of Defense, 1960 (Sammet 1981)

COBOL is the COmmon Business Oriented Language, designed to support the creation of business applications. It was intended to be usable by novice programmers and readable by management; spoken English influenced many of the programming constructs (see Figure 11.28). The designers also added "noise" words to increase the readability of the language: *ADD X TO Y* rather than *ADD X,Y.*

```
IF X = Y <…>
IF GREATER <…>
OTHERWISE <…>
```

Figure 11.28: A conditional statement in COBOL. Conditionals can use implied subjects and objects as seen in the second and third lines of the conditional statement.

Logo: Seymour Papert, MIT, 1967 (Papert 1980)

The Logo programming language is a dialect of Lisp with much of the punctuation removed to make the syntax accessible to children. It was intended to allow children to explore a wide variety of topics, from mathematics and science to language and music. The most well known part of Logo is the Logo turtle, which began as a robotic turtle that could draw on the ground. It was later replaced by a simulated actor in a two dimensional graphical world that can move, turn, and leave trails. The turtle's directions are object-centric; if a child tells the turtle to "forward 10", the turtle will move in his own forward direction rather than a direction defined by the screen. Many children have been

introduced to programming through making the turtle draw simple pictures. However, the Logo language includes a wider variety of possibilities. Classes of children have written music programs, programs that translate English to French, and many others. The Logo language is an interpreted language with descriptive error messages. For example, if a student typed "foward 10" instead of "forward 10" the system would respond with "I don't know how to foward."

Alice98: M. Conway et al, Carnegie Mellon University, 1997 (Conway 1997)

Alice98 is a programmable 3D authoring tool, designed to make authoring interactive 3D graphical worlds accessible to college-level, non-science majors. The authoring tool consists of a scene layout editor in which the user can create their opening scene, and a script tab in which the user can specify the behavior of the world. The programming language in Alice is Python, with a few changes suggested by user testing: it is not case sensitive and ½ evaluates to 0.5 rather than 0. However, Alice provides domain-specific commands for manipulation of objects in 3D. The structure and naming of these domain-specific commands were influenced greatly by user testing. As in Logo, commands utilize object-centric notation: forward, backward, up, down, left and right are used to describe direction. This description is equivalent to XYZ notation, but is much easier for novices to understand. Similarly, the names of commands are drawn from the language that users would choose to describe those actions; for example, translate became move, scale became resize, and rate became speed. Alice commands can also be accessed with varying degrees of detail. At the simplest, *bunny.move* only needs a direction. The user can also specify how far bunny should move, how long the animation should take, what speed he should move at, whether he should move in someone else's coordinate system, and different interpolation styles. This allows novices to begin by learning a very simple command for moving the bunny and, as they gain more experience, learn to express greater control over how the bunny moves through additional options. To help users understand the behavior of their programs, Alice98 animates all changes to the state of the program.

HANDS: J. Pane, Carnegie Mellon University, 2001 (Pane 2002)

The HANDS system was designed to allow children in 5th grade and older to create games and simulations similar to the ones with which they play (see Figure 11.29 below). The design of the system was informed by studies of the language that children with no programming experience use in expressing solutions to programming problems. The environment provides a concrete model of computation, represented by an agent, HANDY the dog, who manipulates a deck of cards. All information used in a program is stored on two-sided cards. The front of each card contains object-related data; the back displays a picture of the object. The user can place cards on the surface of the table, which represents the end-users' view of the program. It includes queries and aggregate operations that reduce the need for data structures and iteration through lists of items. Children using the HANDS system perform better than children using a version of the HANDS system that does not include queries and aggregate operations.

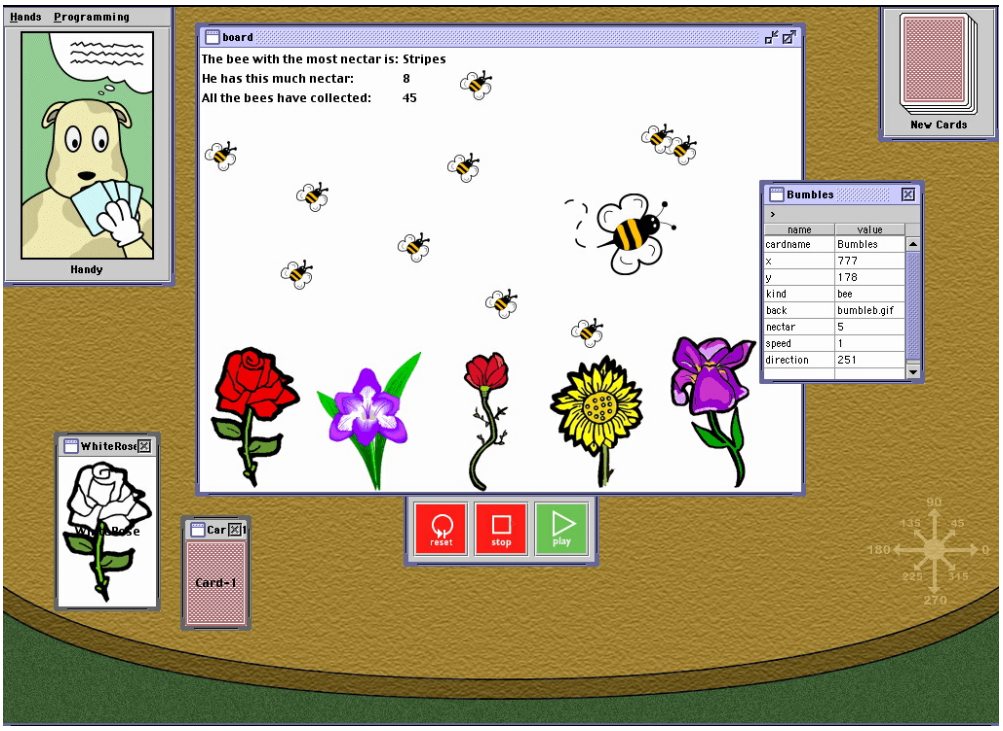


Figure 11.29: All data in HANDS is stored in cards, which the user can draw from a pile shown on the top right of the screen. All the graphics (flowers and bees) and text on the screen are represented as facedown cards. One card on the right has been flipped to face up so that the user can see and edit its properties. When cards are on the board (in the center of the screen), only the image on their backs are visible. Users of HANDS can add code into Handy's thought bubble by clicking on his picture in the upper left corner.

11.4.1.2.2 Improve Interaction with the Language

In addition to changing the language and the words used to describe programming commands and constructs, another area for improvement is in the ways that people

interact with language. The systems in this category examine different methods for creating programs in ways that are easier for novice programmers to understand and less prone to errors. The systems use a variety of techniques from dataflow metaphors, to menu selection, to physical proximity to allow users to express their intentions without having to type traditional programming statements.

Body Electric: J. Lanier, VPL (Blanchard, Burgess et al. 1990)

Body Electric was designed as an authoring tool for a two-person virtual reality system. Programs in Body Electric are data driven; raw data from sensors (such as positional sensors on people) can be passed to the representation of the virtual world through modules that are capable of transforming the data or generating events. These modules are represented in the authoring environment as boxes connected by arrows in a flow diagram. Users can create programs that modify and react to sensor data by sending the sensor data through a sequence of modules. Programs are always live, allowing the author to immediately see the results of changes. This allows worlds to be quickly prototyped, tested, and modified.

Fabrik: Ingalls et al, Apple Computer, 1988 (Ingalls, Wallace et al. 1988)

Fabrik is a computational construction kit in which pieces of functionality (procedures) appear as boxes with connectors. These boxes can be wired together to create a variety of programs (see Figure 11.30). The user is supplied with a parts bin that includes simple computational elements, such as string and integer manipulation, as well as interface elements such as buttons, images, and lists. By dragging boxes into a working area and connecting them together, the user can create programs. As in Body Electric, Fabrik programs are always live so users can test as they are building. During development, user interface elements and computational elements share screen space. However, once a program is finished, the user can choose to view only the interface elements. In addition, finished programs can be used as elements in subsequent programs, so the user can extend the capabilities of the construction kit.

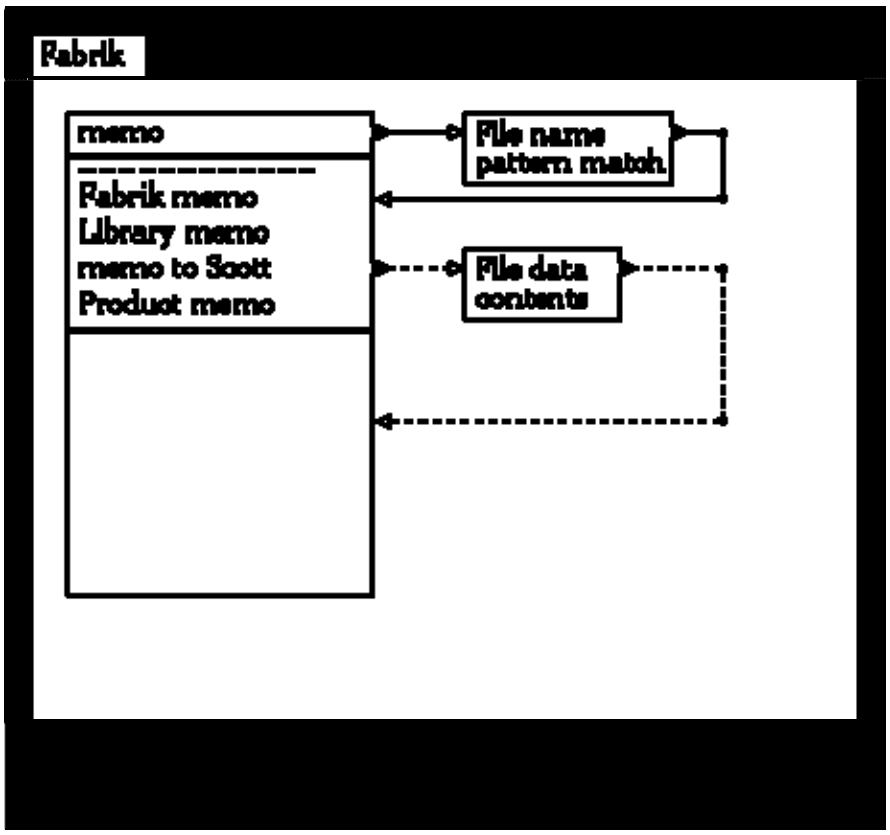


Figure 11.30: A Fabrik program to create a simple text file editor. In the top left text field, the user can enter a search string for file names. The user's string is passed to a file name pattern matcher and then to a GUI list element. The user can then select the file they want to edit. When a file is selected, the name of the file is passed to a module to retrieve its contents and the contents are passed into a text field for the user to edit.

Forms/3: M.Burnett et al, Oregon State University, 1995 (Burnett, Atwood et al. 2001; Hays and Burnett 2001)

Forms/3 is a visual programming language based on the spreadsheet paradigm, which is designed to give end-users access to more powerful programming while maintaining the ease-of-use associated with spreadsheets (see Figure 11.31 below). In Forms/3, users create cells and provide mathematical expressions (which may rely on the values of other cells) that the system will use to compute the value of those cells. To extend the kinds of programs that users can write in Forms/3, the system provides users with the ability to create their own data types (including graphical data types), use a system clock to create time-based calculations and animations, and link spreadsheets together to allow encapsulation of data and functionality.

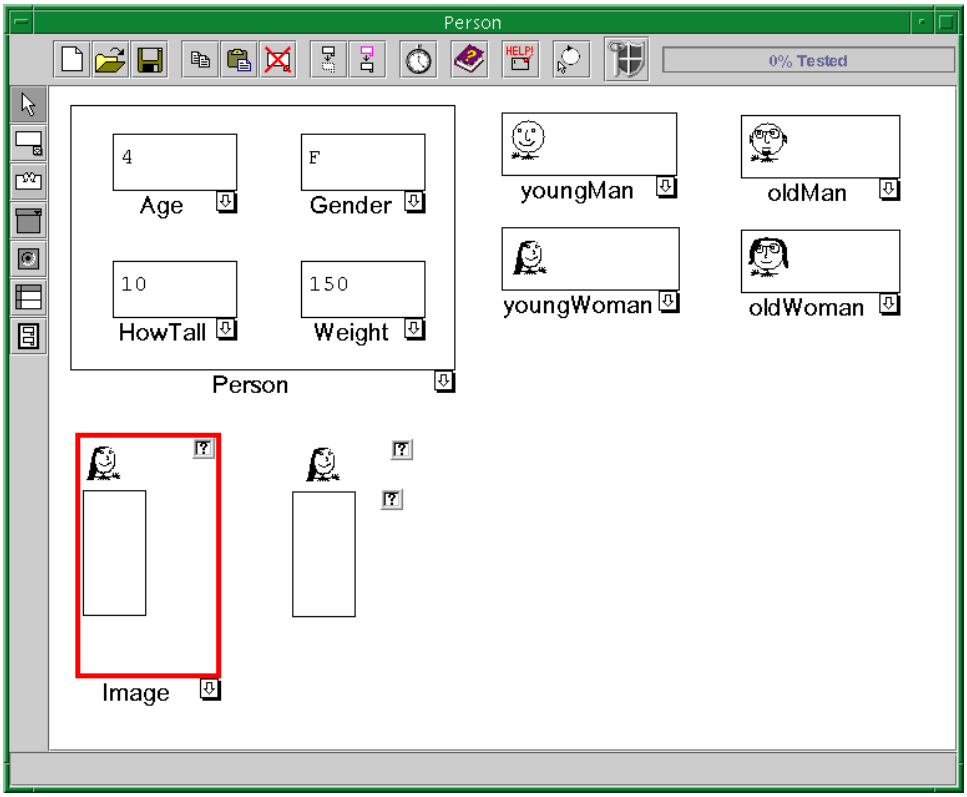


Figure 11.31: A Forms/3 program which creates a graphical representation of a Person. The value for the head is computed with a nested if-statement that selects an appropriate face based on the age (young < 20) and gender of Person. The width and height of the body box are based on the Person's weight and height. To view or edit the equation associated with a given cell, the user can press the arrow symbol below the bottom right corner of the cell.

Tangible Programming with Trains: F. Martin et al, MIT Media Lab, 1996 (Martin, Colobong et al. 1999)

Tangible Programming with Trains is a train set and collection of active train toys that influence the behavior of the train. The Tangible Programming with Trains system was designed to allow children to explore "pre-programming concepts – causality, interaction, logic, and emergence" (Martin, Colobong et al. 1999). For example, a stop sign that causes the train to stop or a sign that asks the train to turn on its lights. The active train toys and the train can communicate via IR signals such that when the train is close to one of these toys, the train will change its behavior appropriately. Children can place these objects around the path of the train such that it will stop at a station or turn its lights on when it goes through a tunnel.

Squeak Etoys: A. Kay et al, Disney, 1997 (Kay)

Squeak Etoys are designed to allow children to learn ideas by "building and playing around with them" (Kay) either through interacting with simulations others have built or creating their own simulations (see Figure 11.32). The Etoys environment provides students with a variety of pre-made objects, from simple shapes to trashcans, and a simple drawing tool with which students can create their own objects. All objects have viewers that contain object-specific information as well as tiles that the student can drag out of the viewer to build programs that control the behavior of the object. Programs can change the position, orientation, size, and appearance of objects as well as play sounds. Users can create simple if-statements in their program, but no other standard control structures are included in the Etoys system. Users can trigger object behaviors based on a variety of mouse events, or the behaviors can be started, stepped and stopped with a set of pre-made buttons users can add to their simulations.

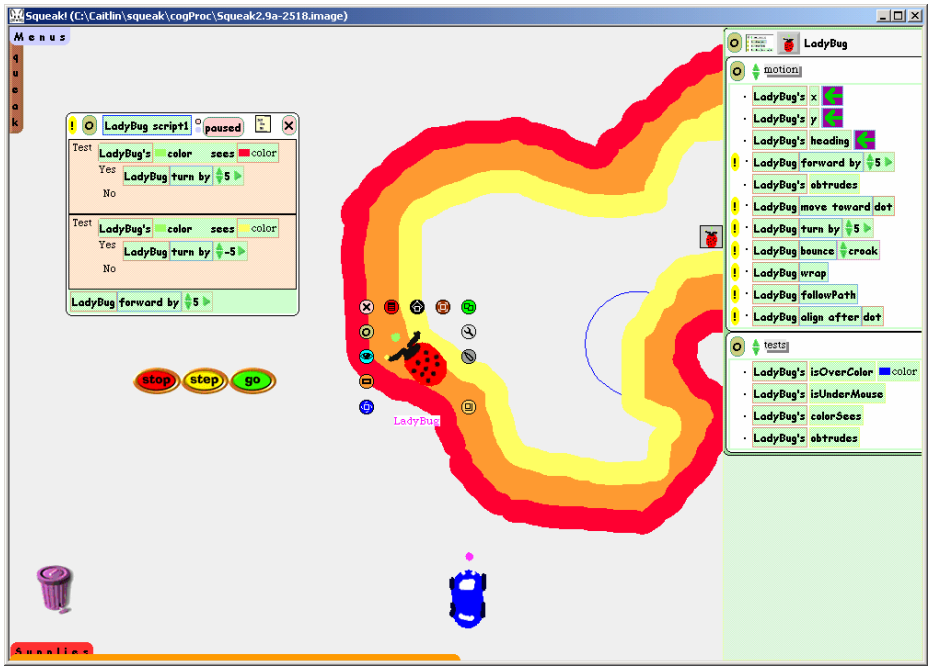


Figure 11.32: An Etoys simulation that makes the LadyBug follow the track. The user has dragged statements from the LadyBug's viewer (right) into a script (left) so that the LadyBug continually moves forward, turning right when she is over red and left when she is over yellow. The script is currently paused, but if the user pressed the "go" button, the LadyBug would start following the track.

Alice99: Carnegie Mellon University, 1999 (1999)

The developers of Alice98 (see section 2.1.2 under *Make the Language More Understandable*) noticed that typing was difficult for many users. This system is a

follow-on system to Alice98 that focuses on exploring ways to reduce the amount of text users have to type. In Alice98, users create both animations and events by typing statements in a programming language. In Alice99, users create basic animation using drag and drop: the user selects the character of interest from the tree of characters on the left of the screen and drags that character into the animations window. When the user drops the character in the animations window, a series of menus appears showing the actions the character can take, such as move, turn, resize, etc, and the options for each of those choices; a character can move forward, backward, left, right, etc. The drag and drop system in Alice99 does not provide support for many of the traditional programming constructs present in the Alice98 system; to create more complex programs, users must still type. The animation editor can create only fully specified, linear animations. The scripting system was left in place to allow advanced users to build complex worlds. Alice99 also introduced an event editor that allowed users to specify events in a table form in which they selected the event and the animation they wanted to trigger in response to that event.

AutoHAN: A. Blackwell and R. Hague, University of Cambridge, 2001 (Blackwell and Hague 2001)

The AutoHAN project grew out of the desire to provide a single programming interface for the many home appliances that are being shipped with customization or programming features. The goal of the project is to provide a language and interface that home users can use to program their appliances to do simple tasks such as recording a particular TV show, switching on an outside light when the doorbell rings, or starting the coffee pot when the alarm goes off in the morning. This language must be usable by people who can operate remote controls. The AutoHAN project elected to create a variety of physical "media" cubes for this purpose. At their simplest, they operate as single button remote controls that can be associated with a wide variety of appliances. For example, a play cube can be associated with a CD player by holding it close to the CD player. Once the association has been created, the user can press the cube's button to play a CD. The user can later associate that same play cube with a VCR and use it to play a movie. Additionally, the cubes can be composed together to form programs, such as starting the coffee pot when the alarm goes off. These programs can be stored by the AutoHAN

system for later use. The designers proposed two languages for the media cubes: one based on ontological abstraction, the other based on linguistic abstraction. The ontological language includes event cubes which reference changes of state in the home, channel cubes which grant access to different channels of information, and aggregate cubes which allow cubes to be grouped together to form a set (a set of events to react to, for example). The linguistic language includes cubes that are linked to particular words in English, for example, stop, go, and play. Cubes that support more abstract data roles such as variables and lists are also included.

Physical Programming: J. Montemayor, University of Maryland, 2002 (Montemayor, Druin et al. 2002)

The Physical Programming work describes a method for children ages 4-6 to build interactive story spaces using StoryRoom Kits that provide sensors and actuators that can be used to augment everyday objects, such as chairs or teddy bears. The StoryRoom kits allow children to create stories in which objects in the real world represent characters or elements in the story the children are telling. Seeking stories in which one character is asking a series of other characters where to find an object, character, or piece of information work very well in this context. The Physical Programming method was prototyped using Wizard of Oz techniques and the following tools: a foam hand to indicate touch, a light for lighting up objects to draw attention to them, a sound box which had a different sound associated with each side of the box, and a magic wand for users to indicate when they were programming and when they wanted to tell a story using their augmented story room. To create a program, a child associates sensors, actuators, and props using the magic wand. For example, to have the teddy bear say something when it is touched, the child would tap the hand and the teddy bear to indicate that the bear should respond when touched, and one side of the sound box to indicate which sound should be played when the teddy bear is touched. When the wand is put away, the StoryRoom goes into "story" mode and the rules the child created are active.

Flogo: C. Hancock, MIT Media Lab, 2001 (Hancock 2001)

Flogo is a visual dataflow language designed to enable children to build more complex robotic behaviors with their lego robotics kits. The designers of the system believe that visualizing the temporal structure of a program is helpful in understanding how it works

(or why it does not work). The visual dataflow model is well suited to showing the temporal structure of a program. Consequently, Flogo programs use a visual dataflow model. Sensor outputs can be connected in the box and wires style to arithmetic operations, Boolean tests, and motor controls. Flogo programs are always live; a change in the inputs to the sensors will be immediately reflected in the representation of the program, making Flogo a tinkering-friendly language even when the program a child is working on is incomplete.

JiVE: J. Hintze and M. Masuch, Otto-von-Guericke University of Magdeburg, 2004 (Hintze and Masuch 2004)

JiVE is a programming environment inspired by Squeak Etoys that was designed to allow children to easily create 3D interactive virtual worlds while learning mathematical concepts. The authors of the system believe that if children draw their own characters, they will be more motivated to animate them. Instead of providing a library of 3D objects, JiVE allows users to draw 2-dimensional sketches of characters. The system then inflates these drawings into 3-D objects for the world using a modification of the Teddy algorithm(Igarashi, Matsuoka et al. 1999). As in Etoys, all objects have viewers that contain information about the object and tiles the user can drag out to create programs. While the Etoys system only allows users to create if-statements, JiVE includes for, while, and repeat loops.

11.4.1.2.3 Integration with Environment

To write a program in most general-purpose languages, a user must type their program into a text editor, compile the program, fix any syntax errors, build the program, and then run it. For a novice programmer, this is a lot of steps and the time and effort involved in making changes to a program can discourage experimentation. The systems in this category integrate the environment in which users write programs with the environment in which users run programs. Many of these systems also allow users to test the effects of individual program statements so that they can experiment while building programs.

Boxer: A. diSessa and H. Abelson, University of California at Berkeley, 1986 (diSessa and Abelson 1986)

Boxer presents a hierarchical world composed of boxes that can contain other boxes (see Figure 11.33). Rather than separating the act of programming, programming is integrated into an environment that a typical person might use, primarily for text editing and graphical layout. Boxer programs contain three types of boxes: standard boxes which can contain text or program code, data boxes which contain string literals for use in programs, and graphics boxes which contain graphical displays. The composition of the boxes has meaning; it indicates that sub-procedures are parts of procedures and records are part of databases. In general, sub-boxes are only accessible from inside a box. The boxes provide the novice programmer with a simple mechanism for abstracting program and data elements. Boxes also allow the novice to view program elements as black boxes that they can use in their programs without fully understanding them. As users gain experience, they can return to these black boxes and open them to discover how they work.

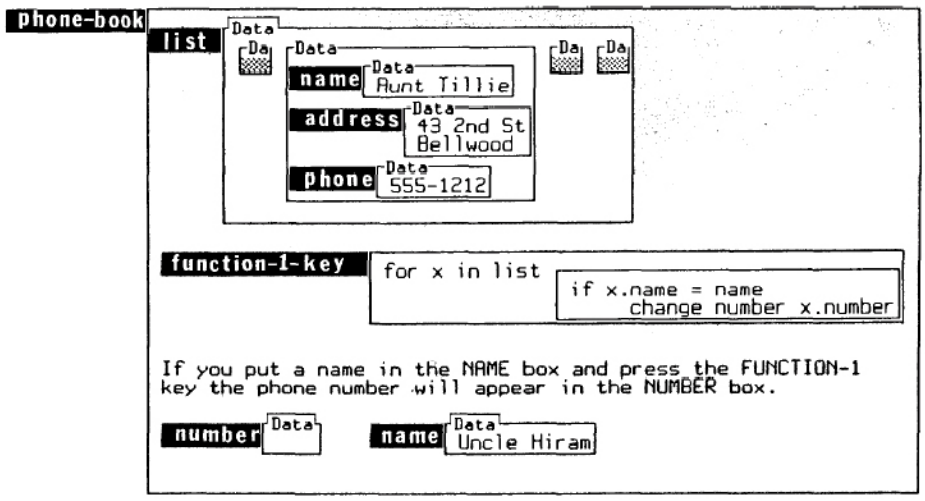


Figure 11.33:A phone number look up program written in Boxer. If a user enters a name in the "name" box and presses the Function-1 key, Boxer will search through the entries in "list", another box shown at the top of the screen, and display the phone number associated with that name.

Hypercard: Bill Atkinson, Apple Computer, 1987 (Atkinson 1987; Goodman 1987)

Hypercard is described by its creator Bill Atkinson as "an authoring tool and a sort of cassette player for information." The application itself allows users to create stacks of cards, somewhat like a Rolodex program, that contain images, text, and buttons. At their simplest, buttons can trigger visual changes, make sounds, or show a new card. A scripting language called Hypertalk is provided to allow users to build more functionality

into the stacks they author. Spoken English heavily influenced the Hypertalk language itself; the language provides constructs such as the "first card" and the "last card", descriptors that are easily understandable to most users. In designing the system, Atkinson concentrated on the user's first experience with the tool. He focused on supporting the user's immediate success using Hypercard and tried to reveal features gradually. A beginning user could learn to create cards and used text-editing tools before moving on to graphics editing. The user could learn about using the message box as a calculator before moving onto placing values in fields. By the time the user was ready to write a full script, he or she would already be familiar with how to access information in different parts of the interface.

cT: B. Sherwood and J. Sherwood, Carnegie Mellon, 1988 (Sherwood and Sherwood 1988)

This system attempts to simplify the process of creating graphics-oriented programs by providing higher-level primitives. Programs are created in an integrated environment where users can see the results of their programs immediately. The cT environment also provides a method for users to specify shapes using mouse clicks on the screen. Finished programs can be executed as separate programs.

Visual AgenTalk: A. Repenning and J. Ambach, University of Colorado, 1996 (Repenning 1993; Repenning and Ambach 1996)

Visual AgenTalk is a programming environment based on an approach the designers of the system call "Tactile Programming" which focuses on allowing users to manipulate code in multiple contexts to aid comprehension, the construction of more complex programs, and sharing between programmers. The designers of AgenTalk believe that users should be able to drop code pieces (either commands or conditional statements) in three contexts: the program editor, the programming world (the grid-based world in which the program runs), and the collaboration world. Allowing users to drop code in the programming world allows users to test the behavior of individual pieces of code without running the whole program. This gives users a way to explore and begin to understand code that they did not create. Visual AgenTalk also allows users to easily share code with other users through the web.

Chart N Art: C. Digiano, University of Colorado, 1996 (DiGiano 1996)

Chart N Art is a graphical editor similar to MacDraw that reveals a programming language. As designers manipulate the interface to create drawings and charts, the equivalent programming statements are printed in a scrolling history area at the bottom. These statements can be copied from the history area into an interaction pane, edited, and executed. The interface provides operations on sets of objects as well as single objects, allowing designers to learn how to specify sets of objects to manipulate using the scripting language. The goal of the interface is to allow designers to automate the creation of custom designed charts, giving them more control than graphing and charting packages, but removing the necessity to draw every aspect of the chart by hand.

11.4.2 Activities Enhanced by Programming

The systems in this group look at programming as a way to enhance activities, either by allowing greater control or creating opportunities to explore particular domains. Rather than trying to create full general-purpose programming environments, the designers of these systems have tailored the functionality in the programming languages to specific domains.

11.4.2.1 Entertainment

These systems use programming to support entertaining activities. These systems use programming models inspired by earlier systems to make programming more realizable to novices and provide activities that the designers believe users will find enjoyable.

Pinball Construction Set: B. Budge, Exidy Software, 1983 (Budge 1983)

The Pinball Construction Set was written in 1983 to allow users to design and build their own pinball machine simulations (see Figure 11.34). It provided a construction space, a set of pinball parts, and bitmap editing capabilities to allow users to build themed pinball machine simulations. Physical laws and behaviors were written into each part; each part provided could be seen as acting on balls that collide with it in defined ways. In this system, users can program by placing pinball parts in well-defined relationships. For example, users may want to specify that when a ball hits a certain target, it is diverted onto a ramp, and its path affected by a magnet.

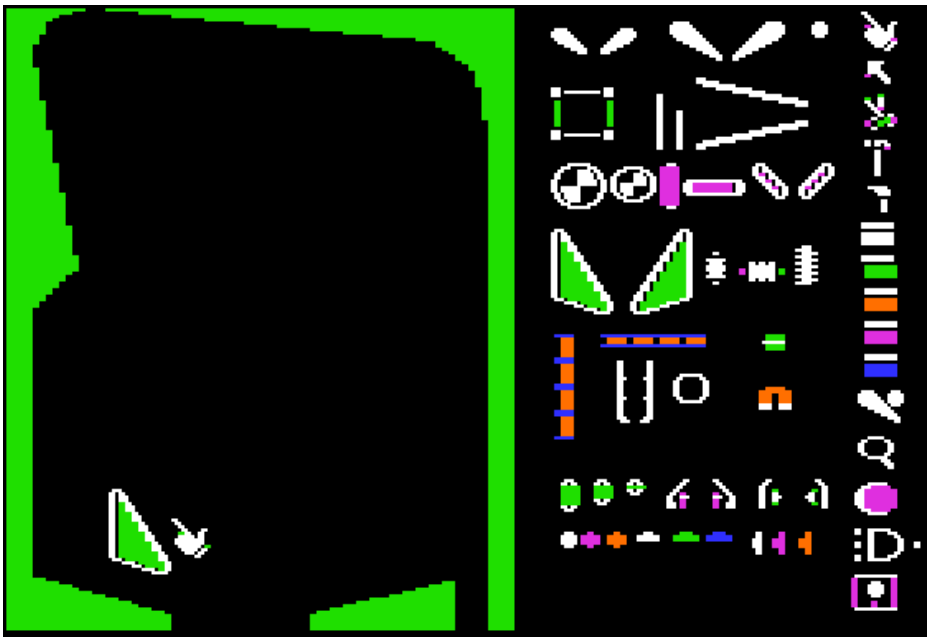


Figure 11.34: A screenshot of the Pinball Construction Set. On the right is an empty pinball game; on the left are a variety of parts that users can put into their pinball games.

The Incredible Machine: Sierra Entertainment, 1993. (1993)

In the Incredible Machine, the player is given a series of Rube Goldberg style challenges (see Figure 11.35). For example, the player may be asked to construct a way to get a ball to fall into a basket. Each challenge includes a short description and all the parts necessary to create the machine described. Players can select parts and position them in the world and then start the simulation to test their machine. When the simulation is running, the parts respond as they would in the physical world. If users run into trouble, they can ask for hints. More advanced users can use a free play mode to create their own machines.

Figure 11.35: An easy challenge in The Incredible Machine: the player needs to help Mel (top left) get back to his house. The puzzle has been solved by positioning the grey pipe, ramp, and a trampoline so that Mel will go through the pipe, slide down the ramp, and bounce off the trampoline and over the barrier to get home.

Widget Workshop: Maxis, 1995 (1995)

Widget Workshop provides a series of puzzles that players attempt to solve by connecting different components together using graphical wires. Each puzzle poses a specific question (e.g. what colors of light do you add together to get white) and provides a context in which to experiment with that question (e.g. red, green, and blue lights controlled by switches that connect to a "light box" where they are combined). Widget Workshop also provides a free play mode in which users can create their own widgets by connecting pre-made parts together.

Bongo: A. Begel, MIT Media Lab, 1997 (Begel 1997)

Bongo enables children to create their own video games and share them with others through the web. Bongo builds upon Starlogo (see section 4.2.2), and adds primitives for playing sounds, changing shapes, and detecting collisions between characters on the screen; it customizes Starlogo for use in the domain of games programming. High-level movement of objects in the system can be done using drag and drop, but procedures are

created with text-based programming. Bongo supplies a command center that allows users to test out code and observe its results.

Mindrover: Cognitoy, 2001 (2001)

Mindrover is a commercial game in which the user is a researcher on Europa, one of the moons of Jupiter. In the researcher's free time, he or she programs robotic rovers to race around hallways and battle other rovers. The game allows users to program their rovers using a drag and drop programming system, inspired by a data-flow visual programming model and The Incredible Machine (see section 4.2.1). Users select pre-built components (such as thrusters and steering wheels) and sensors, place them in a limited number of slots on their rovers, and wire the components and sensors together to give their vehicles certain behaviors. The programming model is similar to the box and wires approach seen in Fabrik, Flogo, and Body Electric. Wires contain information about when signals are sent from sensors to components and the actions triggered by those signals. Boolean gates are provided to allow users to create more complex behaviors.

11.4.2.2 Education

These systems use programming to allow users to build, explore, and experiment with models from different domains of knowledge to gain a stronger understanding of those models. The programming languages are tailored for these specific domains.

SOLO: M. Eisenstadt, The Open University, 1983 (Eisenstadt 1983)

SOLO is a Logo-inspired, interpreted textual programming language designed for cognitive psychology modeling. The typical psychology student has little computer experience, no programming experience, occasional access to a computer, and often works on projects in groups. The SOLO language provides psychology students with a simple way to model cognitive processes through accessing and manipulating a simple database of triples. Each triple represents a relationship: for example, "Fido ***isa*** dog". The language provides 10 commands that allow students to store triples, remove triples, test for relationships via pattern matching, define procedures, iterate through triples, and view and edit procedures. Students are able to quickly create simple models of human memory and reasoning, similar to those discussed in introductory psychology classes, and use these programs to reason about how cognition works.

Gravitas: R. Sellman, The Open University, 1992 (Sellman 1992)

Gravitas is an object-oriented discovery-learning environment that allows students to experiment with Newtonian Gravitation. The environment includes both a graphical interface controlled by the mouse and a textual Logo-based programming interface. Students can control the x and y position, x and y velocity, x and y accelerations, and the mass of the spherical objects in the world. Students typically start with the graphical interface to Gravitas, and then, as they gain more experience they progress to typing Logo commands.

Starlogo: M. Resnick, MIT Media Lab, 1996 (Resnick 1996)

Starlogo is a programmable modeling environment designed to allow students to explore decentralized systems, such as ant colonies and traffic patterns. Users can write simple rules that control thousands of objects and observe the patterns that arise as a result of these rules. The Starlogo programming language is based on Logo (see section 4.1.2 under *Make the Language More Understandable*). However, instead of controlling a single turtle, users control thousands of turtles. The Starlogo turtles have improved senses: they can detect each other, nearby turtles, and scents in the world. Each pixel in the world has additional capabilities. Rather than containing a single piece of information (color), each pixel is modeled as a turtle that cannot move; it can contain an arbitrary amount of information. Pixels in the world can affect the state of other pixels, causing growth or dispersal of scent, for example.

Hank: Mulholland and Watt, The Open University, 1998 (Mulholland and Watt 1998)

Hank is a visual programming language designed for the same audience as SOLO: psychology students who are constructing cognitive models of human behavior. Consequently, the Hank language was designed with five goals in mind: support the creation of cognitive models; consider the requirements of the non-programmer; support group work; clearly show the execution path; and support paper-based use of the language. Based on findings that spreadsheets tend to allow a number of interested people to understand how the spreadsheet is being developed, Hank is a spreadsheet-based language. The architecture of Hank is similar to the information processing architectures taught to psychology students. There are three components: a database where information

can be stored and represented (i.e. long term memory), a workspace where information can be worked upon (i.e. short term memory), and an executive component that carries out processing, input, and output. Data is represented with fact cards that typically represent relationships between entries, similar to a typical spreadsheet. Programs are expressed on instruction cards using queries for entries on cards and arrows to indicate what to do when entries are found or not. The execution model is explained using a dog named Fido who performs programs according to a few simple rules. The authors designed Fido to be similar to the Logo turtle, in the sense that he gives students a physical being to imagine executing their programs, increasing the likelihood that they will be able to accurately simulate their programs on paper. In addition, the environment provides a comic strip representation of the execution of each program; by double clicking on a cell in the comic strip, at student can view the related part of the program.

Starlogo TNG: E. Klopfer and A. Begel, MIT Teacher Education Program, 2006 (Klopfer and Yoon 2005)

Starlogo TNG combines the modeling aspects of Starlogo with the drag-and-drop program creation of LogoBlocks to create a programming environment designed for formal education. The system is designed to be used in classes ranging from computer science to biology and mathematics in order to foster students' development of critical thinking skills and technology fluency. Unlike Starlogo, Starlogo TNG turtles are 3D objects that can move through a 3D virtual world. Turtles can perform basic actions like moving along the ground, turning, and changing colors. Users can define different types or "breeds" of turtles and define new behaviors for them.

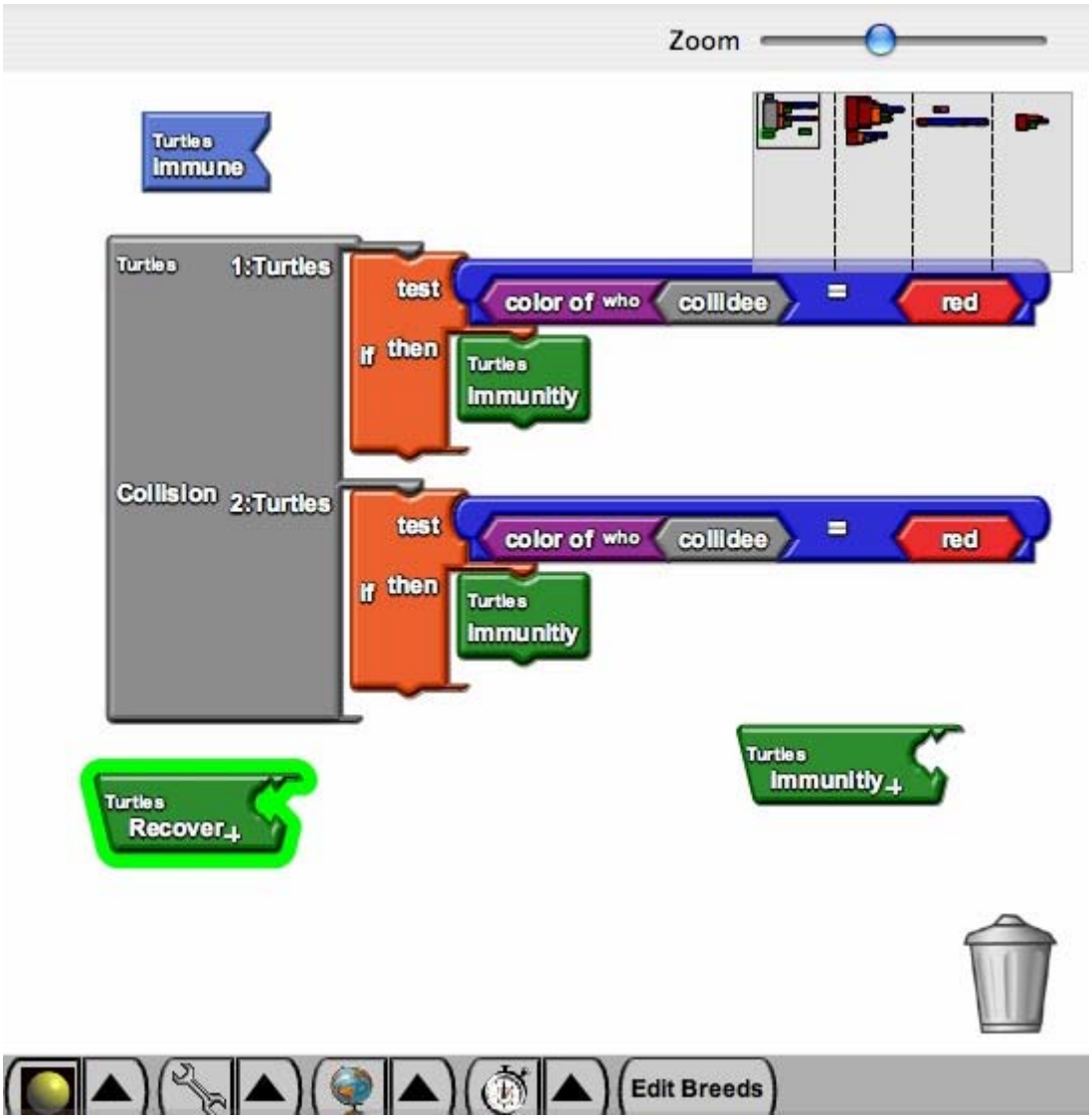


Figure 11.36: Part of a disease simulation program in StarLogo TNG. When two turtles collide, each turtle checks to see whether the turtle it collided with is red. If the turtle's collide is red, then it calls "Immunity."

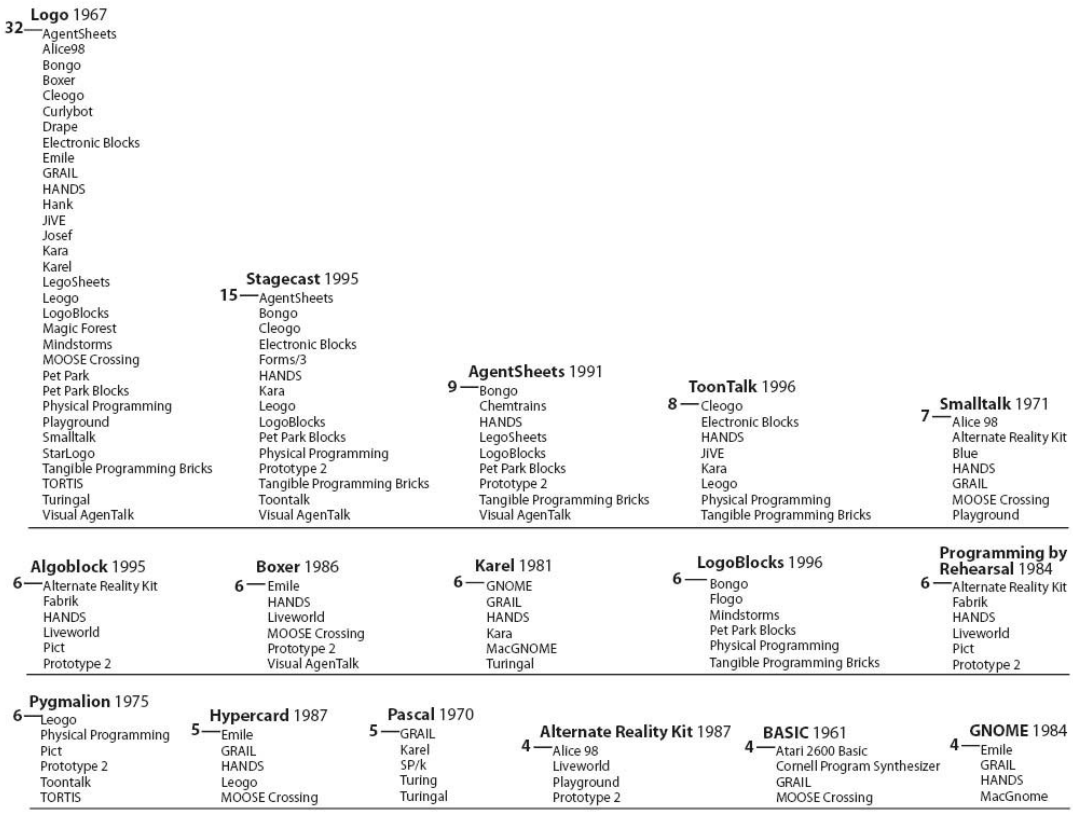
11.5 Additional System Information

We placed systems in our taxonomy based on the primary problem that particular system was trying to address. However, many of the systems described in this paper have incorporated ideas drawn from earlier systems. In this section, we try to pinpoint some of the most influential systems, identify which approaches to making programming more accessible each system has incorporated, and provide information about which programming constructs are included.

11.5.1 System Influences

Table 11.1: System influences attempts to provide some insight into which systems have most influenced the design of later programming systems for novice programmers using the number of citations. The system with the most citations (from papers referenced by this survey) appears first. Underneath the system name is the list of all references to it.

Table 11.1: System influences



11.5.2 System Attributes

Each system appears in our taxonomy only once but many have built on the lessons of systems that have come before. This table attempts to show the major design influences, including those that were not the primary contribution of the system. Figure X is intended to address the following questions:

1. What style of programming does the programming environment or language support?

The systems in the taxonomy fell into six categories: procedural, functional, object-oriented, object-based, event-based, and state-machine based.

2. What programming constructs are available?

We categorized each programming language as having a particular programming construct only if the language included a single statement corresponding to that construct. This excludes languages which do not explicitly support a given construct even if users can replicate the behavior of that construct using a combination of other elements in the language. For example, in a system that does not include a "for" loop, a user can create the behavior of a "for" loop using a "while" loop and a variable. This system would be classified as supporting "while" loops but not "for" loops.

3. How does code look in the programming environment or language?

The systems in our taxonomy represent programs using text, pictures, flow charts, animation, forms users can fill in, finite state machines, and physical objects users can manipulate.

4. What actions do users take to construct programs?

Users can construct programs by typing code, assembling graphical objects, demonstrating actions through an interface, selecting from valid options or filling values into a form, and assembling physical objects.

5. Does the programming environment provide additional support to enable users to better understand the behavior of their programs?

Environments in our survey used several techniques to help users understand the behavior of their programs. These included: 1) back stories designed to explain the world in which programs execute and what actions are possible within those worlds, 2) debugging support, 3) choosing commands with a physical interpretation (for example, move forward or turn right) such that users can "act out" their programs, 4) allowing users to make changes to a running program so that users can immediately see the effects of those changes (liveness), and 5) the ability to generate example programs that correspond to users' interface actions.

6. Does the programming environment attempt to prevent syntax errors in any way?

Environments help to prevent users from making syntax errors by: 1) using the shape of objects to suggest to users which program elements can be connected together (physical

shape affordance) 2) allowing users to select from valid options based on their current position within the program 3) using syntax directed editing 4) allowing users to drop graphical objects only in places where they would be syntactically correct and 5) providing better syntax error messages to enable users to more easily recover from syntax errors that do occur.

7. Have the designers of the language made any explicit attempt to make the language easier to learn?

Language designers used a number of techniques to make programming languages easier for novices to learn. These included: 1) limiting the domain so that there are fewer commands for users to learn, 2) selecting user-centered keywords, 3) removing unnecessary punctuation, 4) making statements in the programming language as close to natural language as possible, and 5) removing any redundancy in the language.

8. Does the environment support users collaborating on programs?

Environments enabled three types of collaboration between users: 1) side by side based collaboration in which two or more users were manipulating the same program on computers that were located in the same room, 2) networked shared manipulation in which users were in different locations but connected to a common network and could collaborate while building a program and 3) networked shared results in which users were in different location but connected to a common network and could share completed programs or program fragments.

9. What were the primary considerations behind what the authors of the system envisioned users creating with it?

The systems in the taxonomy fell into three categories: 1) fun and motivating systems were designed to support a task the creators of the system believed users would find enjoyable 2) useful systems were designed to enable users to solve a particular type of problem 3) educational systems were created specifically to aid in teaching either programming or another topic.

Table 2: System Attributes - part 1

			Simplify the language						Prevent syntax errors			Construct Programs Using Objects												Create Programs Using Interface Actions				Multiple Methods for Creating
			1963 BASIC	1977 SPik	1988 Turing	1996 Blue	1998 JJ	2001 GRAIL	1981 Cornell Program Synthesizer	1984 GNOME	1986 MacGnome	1976 TORTIS - Slot Machine	1984 Pict	1986 Play	1990 Show and Tell	1995 My Make Believe Castle	1995 ThinkIn Things Collection 3: Half Time	1996 LogoBlocks	1998 Pet Park Blocks	2000 Drape	2000 Electronic Blocks	2002 Alice2	2002 Magic Forest	1976 TORTIS - Button Box	1989 Roamer	1995 LegoSheets	2000 Curlybot	1997 Leogo
1	Style of Programming	procedural	x	x	x		x	x	x			x	x		x		x	x		x			x	x	x		x	x
		functional																											
		object-based																				x							
		object-oriented				x	x												x										
		event-based														x		x	x		x		x			x			
		state-machine based																											
2	Programming Constructs	conditional	x	x	x	x	x	x	x			x	x					x	x		x	x			x	x		x
		count loop						x	x			x					x	x		x		x		x	x			
		for loops	x	x	x		x																						x
		while loops	x	x	x	x	x		x													x							
		variables	x	x	x	x	x	x	x			x	x		x							x							
		parameters	x	x	x	x	x	x	x				x		x							x							x
		procedures/methods	x	x	x	x	x	x	x			x	x		x					x		x			x				x
		user-defined data types	x	x	x	x		x																					
		pre and post conditions				x	x																						
3	Representation of Code	text	x	x	x	x	x	x	x	x						x		x	x			x			x	x		x
		pictures										x	x		x	x	x		x	x			x	x	x	x		
		flow chart											x		x														
		animation																									x		
		forms																											x
		finite state machine																											
		physical objects										x									x								
4	Construction of Programs	typing code	x	x	x	x	x	x		x	x														x				x
		assembling graphical objects											x		x		x	x		x		x	x						
		demonstrating actions																						x			x	x	
		selecting/form filling							x							x										x			x
		assembling physical objects										x										x							
5	Support to Understand Programs	back stories																											
		debugging																											
		physical interpretation															x			x			x	x	x				
		liveness													x									x		x			
		generated examples																											x
6	Preventing Syntax Errors	physical shape affordance															x	x	x		x	x	x						
		selection from valid options							x				x									x							x
		syntax directed editing							x	x	x																		
		dropping only in valid location																x				x	x						
		better syntax error messages					x																						
7	Designing Accessible Languages	limit the domain							x			x	x		x	x	x	x	x	x	x		x	x	x	x	x	x	x
		select user-centered keywords																				x							x
		remove unnecessary punctuation			x		x															x							
		use natural language																											
		remove redundancy		x		x	x	x																					
8	Support Commu-nication	side by side										x									x			x	x		x	
		networked- shared manipulation																											
		networked - shared results																	x										
9	Choice of Task	fun & motivating										x			x	x	x	x	x	x	x	x	x	x	x	x	x	x	
		useful	x	x																									
		educational	x	x	x	x	x	x	x	x	x	x	x		x	x	x	x	x	x	x	x	x	x	x	x	x	x	x

Table 3: System Influences - part 2

			New Programming Models				Making New Models Accessible					Tracking Execution	Make Programming Concrete			Models of Program Execution		Side by Side		Networked Interaction			Providing a Motivating Context			Demonstrate Actions in the Interface			Demonstrate Conditions and Actions			Specify Actions		
			1970 Pascal	1971 Smalltalk	1989 Playground	2001 Kara	1994 Liveworld	1996 Blue Environment/BlueJ	1997 Karel++	2001 Karel J Robot	2001 J Karel	1979 Atari 2600 BASIC	1981 Karel	1983 Josef	1991 Turingal	1996 ToonTalk	1998 Prototype 2	1995 AlgoBlock	2000 Tangible Programming Bricks	1997 MOOSE Crossing	1998 Pet Park	1998 Cleogo	1982 Rocky's Boots/ Robot Odyssey	1995 AlgoArena	2001 Robocode	1975 Pygmalion	1984 Programming by Rehersal	1992 Mondrian	1991 AgentSheets	1993 ChemTrains	1995 Stagecast	1987 Alternate Reality Kit	1994 Klik N Play	1995 Emile
1	Style of Programming	procedural	x									x	x	x	x	x	x	x	x			x	x			x	x						x
		functional																				x												
		object-based			x																													
		object-oriented		x			x	x	x	x	x									x	x				x									
		event-based		x	x		x													x	x						x		x	x	x	x	x	x
		state-machine based				x																												
2	Programming Constructs	conditional	x	x	x	x	x		x	x	x	x	x	x	x	x	x			x	x	x	x	x	x	x			x	x	x		x	x
		count loop							x	x			x	x					x		x			x										x
		for loops	x	x			x				x	x			x					x		x			x									x
		while loops	x	x			x		x	x	x	x	x	x	x		x			x				x	x	x								x
		variables	x	x	x		x				x	x		x		x			x	x	x			x	x								x	x
		parameters	x	x			x				x	x		x		x	x		x		x			x	x							x		x
		procedures/methods	x	x	x	x	x		x	x	x	x	x	x	x	x	x			x		x			x	x	x	x	x	x	x			x
		user-defined data types	x	x			x				x	x													x									
		pre and post conditions																																
3	Representation of Code	text	x	x	x		x	x	x	x	x		x	x	x		x			x	x	x			x		x					x		x
		pictures				x		x															x			x		x	x	x	x			
		flow chart																					x			x								
		animation														x																		
		forms																				x					x						x	
		finite state machine				x																												
		physical objects																x	x															
4	Construction of Programs	typing code	x	x	x		x	x	x	x	x		x	x	x					x	x	x			x	x								x
		assembling graphical objects				x																	x			x		x	x	x	x	x		
		demonstrating actions														x	x					x				x	x		x	x	x			
		selecting/form filling															x					x		x									x	x
		assembling physical objects																x	x															
5	Support to Understand Programs	back stories							x	x	x		x	x		x	x																	
		debugging																													x			
		physical interpretation				x							x	x		x		x						x								x		
		liveness						x																		x						x		
		generated examples																				x												
6	Preventing Syntax Errors	physical shape affordance																					x											
		selection from valid options																				x											x	
		syntax directed editing																																
		dropping only in valid location																															x	
		better syntax error messages																																
7	Designing Accessible Languages	limit the domain			x	x			x	x	x		x		x			x				x		x							x	x	
		select user-centered keywords			x				x	x	x		x									x												
		remove unnecessary punctuation																																
		use natural language			x																						x							
		remove redundancy																																
8	Support Communication	side by side																x																
		networked- shared manipulation																				x												
		networked - shared results																		x	x													
9	Choice of Task	fun & motivating			x											x		x		x	x		x	x	x					x	x	x	
		useful	x	x			x	x											x				x	x	x									
		educational	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x				x	x	x	x	x	x

Table 4: System Influences - part 3

			Make the Language More Understandable				Improve Interaction with Language										Integration with Environment					Entertainment					Education			
			1960 COBOL	1967 Logo	1997 Alice 98	2001 HANDS	1985 Body Electric	1988 Fabrik	1995 Forms/3	1996 Tangible Programming with Trains	1997 Squeak e-Toys	1998 Alice 99	2001 AutoHAN	2001 Physical Programming	2001 Flogo	2004 JiVE	1986 Boxer	1987 Hypercard	1988 cT	1996 Visual AgenTalk	1996 Chart N Art	1983 Pinball Construction Set	1993 The Incredible Machine	1995 Widget Workshop	1997 Bongo	2001 Mindrover	1983 SOLO	1992 Gravitas	1996 Starlogo	1998 Hank
1	Style of Programming	procedural	x	x			x	x							x	x	x	x	x		x		x	x		x	x		x	x
		functional							x																						
		object-based				x					x	x																			
		object-oriented			x																				x			x			
		event-based			x	x	x	x		x	x	x	x	x	x	x	x	x		x		x		x	x	x			x	
		state-machine based																													
2	Programming Constructs	conditional	x	x	x	x		x	x		x				x	x	x	x	x	x	x	x	x		x	x	x	x	x	x	x
		count loop														x		x			x							x			
		for loops	x	x	x	x											x	x	x		x				x				x		
		while loops	x	x	x											x	x	x			x				x			x	x		
		variables	x	x	x			x	x		x					x	x	x	x		x				x		x	x	x	x	x
		parameters	x	x	x			x					x				x	x	x		x				x		x	x	x	x	x
		procedures/methods	x	x	x	x		x			x		x			x	x	x	x	x	x				x		x	x	x	x	x
		user-defined data types	x		x	x			x																						
		pre and post conditions																													
3	Representation of Code	text	x	x	x	x			x		x	x				x	x	x	x		x				x		x	x	x		
		pictures																		x		x	x	x		x					
		flow chart					x	x							x								x	x							x
		animation																													
		forms							x																			x			
		finite state machine																													
		physical objects								x			x	x																	
4	Construction of Programs	typing code	x	x	x	x			x								x	x	x		x				x		x	x	x	x	x
		assembling graphical objects					x	x	x		x	x			x	x				x		x	x	x		x					x
		demonstrating actions																		x											
		selecting/form filling				x						x							x												x
		assembling physical objects								x			x	x																	
5	Support to Understand Programs	back stories				x																									
		debugging																													
		physical interpretation		x	x	x					x					x						x	x	x				x			
		liveness				x	x	x						x	x		x			x							x				
		generated examples																			x							x				
6	Preventing Syntax Errors	physical shape affordance											x										x								
		selection from valid options									x	x				x															
		syntax directed editing				x																					x				
		dropping only in valid location									x					x															
		better syntax error messages																													
7	Designing Accessible Languages	limit the domain		x					x		x	x	x			x						x		x			x	x		
		select user-centered keywords			x	x					x					x															
		remove unnecessary punctuation		x																											
		use natural language	x			x					x																				
		remove redundancy				x																									
8	Support Commu-nication	side by side											x	x																	
		networked- shared manipulation																													
		networked - shared results																		x											
9	Choice of Task	fun & motivating		x	x	x					x	x		x		x						x	x	x	x	x					
		useful	x		x	x	x	x	x	x		x	x	x			x	x	x		x				x	x	x		x	x	
		educational		x	x						x			x	x	x				x	x	x	x	x			x	x	x	x	

11.6 Summary and Future Directions

The systems presented in this paper have tried to make programming accessible in three main ways: simplifying the mechanics of programming, providing support for learners, and providing students with motivation to learn to program. The majority of the systems have focused on the mechanics of programming. Clearly, beginners need to feel that they can make progress in learning to program. However, pure difficulty is not the only reason that people hesitate to learn to program. There are a variety of sociological factors (including students not seeing the relevance of programming or perceiving computer science as being a socially isolating career path) that can prevent people from learning to program. Creating environments that address some of these sociological barriers to programming by supporting learners or providing interesting reasons to program have the potential to attract a more diverse group of people to computer science. If the population of people creating software is more closely matched to the population using software, the software designed and released will probably better match users' needs. In addition to the potential benefits to society of having a diverse Computer Science population, we believe that learning to program will benefit individuals both as a mode of thought and as preparation for interacting with technology in daily life.

11.6.1 Mechanical Barriers to Programming

Most of the programming systems built for children and novice adults have focused on making the mechanics of programming more manageable. Systems have removed unnecessary syntax, designed languages that are closer to spoken English, introduced programming in visible contexts (such as the Logo turtle) in which students can see the immediate results of their commands, and explored alternatives to typing programs. Using these ideas, it is possible to create a system that will allow a wider audience of people to begin programming. While these systems do not take all of the challenges out of programming, they can allow students to focus on the logic and structures involved in programming rather than worrying as much about the mechanics of writing programs. However, even with these improvements to a beginner's first programming experience, there are a number of questions that remain.

Many of the teaching languages have been heavily influenced by the prevalent general-purpose languages of their time. Designers of these systems chose to make the programming constructs and syntax very similar to those of the general-purpose languages to ease the transition from teaching languages to general-purpose languages. While it seems obvious that students need to understand the parallels between the programming constructs in teaching and general-purpose languages, it is not clear how closely and in what ways teaching languages must resemble general-purpose languages.

11.6.2 Sociological Barriers to Programming

In some ways, sociological barriers can be harder to address than mechanical ones because they are harder to identify and some cannot be addressed through programming systems. However, by studying particular groups of people who choose not to learn to program, identifying the reasons behind their decisions, and trying to address those reasons in our programming systems and textbooks, we may be able to attract a broader audience of people to programming and Computer Science. The systems in the taxonomy have identified and are beginning to address two kinds of sociological barriers to programming: the lack of a social context for programming and the lack of compelling contexts in which to learn programming.

11.6.2.1 Social Support

It can be easier and more fun to learn with a group of people. MOOSE Crossing (Bruckman 1997) and, later Pet Park (DeBonte 1998) added support for social interaction so that students using these systems can share projects, provide examples for each other, and chat. Future communities might provide support for students helping each other learn the interface and programming constructs, support students working on projects together, or try to capture and strengthen the positive feedback that members of the community give to each other through looking at and using each other's work.

11.6.2.2 Reasons to Program

Several systems have tried to provide motivating contexts such as building robots, fighting battles, and constructing machines in which to learn programming. While these systems have been very effective for a segment of the population, they do not have broad

appeal. Newer systems are beginning to search for motivating contexts to introduce programming that have broad appeal or appeal to groups that are under-represented in computer science.

Appendix A: Storyboarding Worksheets

11.7 Worksheet 1

The first storyboarding worksheet that I used in formative testing was a guide that was developed by Adam Shulman and is included on a website of curricular information for "Project-Based Learning with Multimedia." It is available at http://pblmm.k12.ca.us/TechHelp/Storyboarding.html

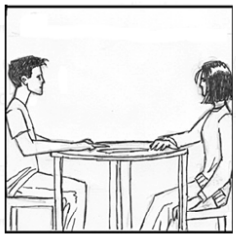
This Storyboarding Guide was developed by Adam Shulman, IRL summer intern, and adapted from *The Rowland Animation Guide to Storyboarding*

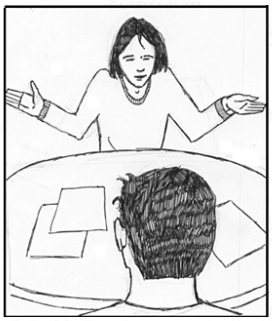
Storyboarding Activity (Page 1 of 12)

Thursday afternoon at Aristos' house

...so then at the end of the film, the guy realizes that it wasn't his sister, but his mother that planned the whole thing all along.

Cool! Let's do it.





How do we do it?

I'm not really sure. We should probably plan it out first before we actually get started.

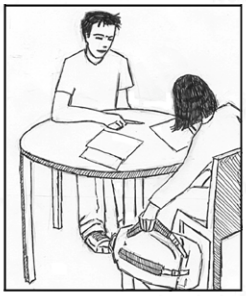
I agree, but how should we go about planning it.

I don't really know. We could just get started and see how it goes from there.

hmmmmm

Oh! my aunt works at Z productions. I bet she could give us some pointers on how to plan out our film.

Storyboarding Activity (Page 2 of 12)



Cool, let's call her up. Do you know her number?

It's in my bag, just a sec.

LATER...

Hello

Hi Alice this is your niece Rebecca

Oh hi Rebecca, how are you?

I'm pretty good. Wait a sec, I'm going to put you on speaker phone...

Listen, my friend and I are working on a film project for school and we know what we want to do but we're not sure how to get started. We were wondering if you could give us some pointers.

Of course. OK where do I start. Well the first thing that you have to do is to create a storyboard.

What's that?

A storyboard is a visual script for your project. It's your project in outline form.

Storyboarding Activity (Page 3 of 12)

I don't understand.

OK ummmm, a storyboard is a series of visual images that simply and briefly illustrate the film's key scenes and events.

I get it.

Be careful though. A storyboard doesn't illustrate every moment in the film. It is not a frame by frame breakdown of the story, but rather a scene by scene breakdown.

OK I think I understand.

In each of your storyboard drawings, you want to single out the essential details needed to communicate the information in that particular scene. There are two ways to do this, a visual description and a written description.

So basically, for each scene, we have a drawing or a series of drawings that describe what's going on.

Storyboarding Activity (Page 4 of 12)

Yes, sort of, but in your storyboard you're not only describing the plot but all information that's important to your scene such as the mood, the setting and anything else that you think will help the audience understand your story.

To create a storyboard, you should follow three basic steps. First is analysis which is breaking down your story into its component parts. Second is evaluation which is judging and choosing what shots angles and frame sizes you will put in your project, and third is synthesis which is the process of actually developing and putting your project together.

It's a good thing we called you because I was planning on skipping right to synthesis when we first started.

Yes it is a good thing you called me because planning in a film is very very important. You save countless hours of uneccesary editing by doing a storyboard. It is especially important in animation. Unlike live-action filming where the filmaker shoots tons of footage and then edits it later, an animator wants to throw away as little of his/her work as possible because it is more work to animate a scene than it is to film it in live action. If you plan it all out in advance you don?t have to worry about wasting time animating scenes that you?ll never use.

Another reason why a storyboard is important is because it is a way to uncover problems and to fix them while they are still easy to fix.

Storyboarding Activity (Page 5 of 12)

We'll probably run into tons of problems.

Yes you definitely will. However, the mistakes you make don't become serious problems if you can fix them while your project is in the planning stage.

A storyboard is a representation of a story. It is a guide, a plan and a blueprint from which you will direct you film. It is taking your ideas and translating them into visual images.

A storyboard also makes experimentation possible. You can try out new things and easily change your film when it is in the storyboard stage. It is also much easier to revise a storyboard than it is to revise the actual film.

Cool, so if we don't like something and want to cut it or want to add something, we can without a lot of effort.

Exactly.

There are four things that a storyboard does for you. First, it is a way to work out and discuss your ideas. Second, is a visualization of how your film will look. Third, it is a description of how the film is sequenced and put together, and fourth, it is a step by step guide to making and shooting your film.

Storyboarding Activity (Page 6 of 12)

OK I think we understand what a storyboard is, but can you tell us what making a storyboard will achieve?

With your storyboard, you will achieve a better sense of what is going on in your project and you will be able to plan every step and mold your project into exactly what you want it to be. There are certain goals that you want to accomplish in creating your storyboard that will help you do this.

The first is put your shots and scenes of your storyboard in an order that tells your story clearly. Second, plan your story so that the visual images and the script can be clearly understood by reading your storyboard.. Third, plan your film in the most interesting and appealing way possible for the audience. Fourth, plan not only what happens in each shot, but also how fast or how slow you want it to happen. Fifth, eliminate unnecessary or repetitive shots and add missing shots. Cut long boring shots and break them down into shorter more interesting shots. Lastly, make sure there is a smooth, clear, logical flow from shot to shot and scene to scene.

Kids, I'm sorry but have to go. I'd like to give you some more help though. When is your project due?

In about a month. 

Storyboarding Activity (Page 7 of 12)

OK, how about you two get started and I can come over and meet with you in a couple days and review what you've done. I can see if you're on the right track and give you a few more pointers. How about Saturday at around 11:00 in the morning.

OK sounds good, we'll see you then and thanks for the help. Bye.

Bye.

OK let's get started....

Two days later at Rebecca's house...

You have a lot of good stuff here. Your story is very good and you've planned out many of your shots well. Your transitions are good and the scenes flow nicely together. However, I would recommend more variation in your transitions. Don't make them all the same. Keep changing them to keep the audience engaged.

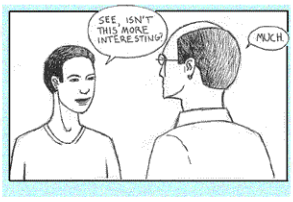
I like how you used many different perspectives though at times it seems that you didn't pay too much attention to what mood different perspectives create. Usually, a low angle gives the feeling that the object is dominant or superior and an angle looking down gives the feeling that the object is weak and inferior.

Storyboarding Activity (Page 8 of 12)

Another thing that you need to think about is using the whole depth of the frame. Often times in your storyboard, the movement of the characters is limited to left-right. To make it more interesting and to add a third dimension, have your characters moving towards and away from the characters at times.

Storyboarding Activity (Page 9 of 12)

Also, when your characters are having interactions, you want to experiment with different perspectives instead of just having the two characters side by side. Put one character at a different angle from the other character to make the scene more interesting and three dimensional.



Storyboarding Activity (Page 10 of 12)

It is nice the way you made the center of interest in each scene stand out. You didn't camouflage them in the background or hide them in the shadows and they grab my attention.

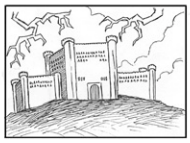
I also like the way that you used anticipation in your scenes. There is always a physical or emotional response to every action. This is very important because the scene makes no sense if there is no reaction to each action.

Storyboarding Activity (Page 11 of 12)

One thing that you need to indicate underneath your pictures that I see none of, is the kind of camera movement that you want such as pan, zooming in and zooming out and whether the movement is actual or apparent.

Another thing that you need to indicate is how long you want each shot to last. This is important for the timing of your movie. The timing will affect the mood of the scene and the flow of the film.

Something that you really need to work on is choosing the correct frame size for each scene. You tended to stay the same distance from your subject in every frame. You need to vary the frame size depending on what kind of shot you are taking and what kind of mood you want to convey. A long shot establishes location and displays mood.



A midrange shot shows interactions between characters. A medium close-up shot communicates gestures. A close-up shot communicates expressions. An extreme close-up shot communicates a strong emotional impact.

Storyboarding Activity (Page 12 of 12)

I think that that's all the advice that I can give you. I know it sounds like you've done a lot wrong but you really haven't. You're off to a great start, I'm just being nit picky. If you need any more help, feel free to call me up again.

OK, thank you Alice.

Yeah you've been really helpful.

It was my pleasure. I must be off, I have a lunch meeting with Steven Spielberg. Good luck with your project.

Thanks. See you later Alice.

Bye

Bye

Let's get to work...

11.8 Worksheet 2

The second worksheet included the full text of a folk tale ("The Tinker and the Ghost") and an example script and storyboard based on it. The worksheet was intended to serve as an example to guide participants in creating a good storyboard.

The Tinker and the Ghost

Spain

From Favorite Folktales from Around the World, edited by Jane Yolen

On the wide plain not far from the city of Toledo, there once stood a great grey castle. For many years before this story begins no one had dwelt there, because the castle was haunted. There was no living soul within its walls, and yet on almost every night in the year a thin, sad voice moaned and wept and wailed through the huge, empty rooms. And on All Hallow's Eve a ghostly light appeared in the chimney, a light flared and died and flared against the dark sky.

Learned doctors and brave adventurers had tried to exorcise the ghost. And the next morning they had been found in the great hall of the castle, sitting lifeless before the empty fireplace.

Now one day in late October there came to the little village that nestled around the castle walls a brave and jolly tinker whose name was Esteban. And while he sat in the marketplace mending the pots and pans the good wives told him about the haunted castle. It was All Hallows Eve, they said, and if he would wait until nightfall he could see the strange ghostly light flare up from the chimney. He might, if he dared go near enough, hear the thin, sad voice echo through the silent rooms.

"If I dare!" Esteban repeated scornfully. "You must know, good wives, that I – Esteban – fear nothing, neither ghost nor human. I will gladly sleep in the castle tonight, and keep this dismal spirit company."

The good wives looked at him in amazement. Did Esteban know that if he succeeded in banishing the ghost the owner of the castle would give him a thousand gold pieces?

Esteban chuckled. If that was how matters stood, he would go to the castle at nightfall and do his best to get rid of the thing that haunted it. But he was a man who liked plenty to eat and drink and a fire to keep him company. They must bring him a load of faggots, a side of bacon, a flask of wine, a dozen fresh eggs, and a frying pan. This the good wives gladly did. And as the dusk fell, Esteban loaded these things on the donkey's back and set out for the castle. And you may be very sure that not one of the village people went very far along the way with him!

It was a dark night with a chill wind blowing and a hint of rain in the air. Esteban unsaddled his donkey and set him to graze on the short grass of the deserted courtyard. Then he carried his food and his faggots into the great hall. It was dark as pitch there. Bats beat their soft wings in his face and the air felt cold and musty. He lost no time in piling some of his faggots in one corner of the huge stone fireplace and in lighting them. As the red and golden flames leaped up the chimney Esteban rubbed his hands. Then he settled himself comfortably on the hearth.

"That is the thing to keep off both cold and fear," he said.

Carefully slicing some of the bacon he laid it in the pan and set it over the flames. How good it smelled! And how cheerful the sound of its crisp sizzling!

He had just lifted his flask to take a deep drink of the good wine when down the chimney there came a voice – a thin, sad voice – and "Oh me!" it wailed, "Oh me! Oh me!"

Esteban swallowed the wine and set the flask carefully down beside him.

"Not a very cheerful greeting, my friend," he said, as he moved the bacon on the pan so that it should be equally brown in all its parts. "But bearable to a man who is used to the braying of his donkey."

And "Oh me!" sobbed the voice, "Oh me! Oh me!"

Esteban lifted the bacon carefully from the hot fat and laid it on a bit of brown paper to drain.

Then he broke an egg into the frying pan. As he gently shook the pan so that the edges of his egg should be crisp and brown and the yolk soft, the voice came again. Only this time it was shrill and frightened.

"Look out below," it called. "I'm falling."

"All right," answered Esteban, "only don't fall into the frying pan."

With that, there was a thump, and there on the hearth lay a man's leg! It was a good leg enough and it was clothed in the half of a pair of brown corduroy trousers.

Esteban ate his egg, a piece of bacon and drank again from the flask of wine. The wind howled around the castle and the rain beat against the windows.

Then, "Look out below," called the voice sharply. "I'm falling!"

Then there was a thump, and on the hearth there lay a second leg, just like the first!

Esteban moved it away from the fire and piled on more faggots. Then he warmed the fat in the frying pan and broke into it a second egg.

And "Look out below" roared the voice. And now it was no longer thin, but strong and lusty. "Look out below! I'm falling"

"Fall away" Esteban answered cheerfully. "Only don't spill my egg!"

There was a thump, heavier than the first two, and on the hearth there lay a trunk. It was clothed in a blue shirt and a brown corduroy coat.

Esteban was eating his third egg and the last of the cooked bacon when the voice called again, and down fell first one arm and then the other. "Now," thought Esteban, as he put the frying pan on the fire and began to cook more bacon. "Now there is only the head. I confess that I am rather curious to see the head."

And, "LOOK OUT BELOW!" thundered the voice. "I'M FALLING –FALLING!"

And, down the chimney there came tumbling a head!

It was a good enough head, with thick black hair, a long black beard and dark eyes that looked a little strained and anxious. Esteban's bacon was only half cooked. Nevertheless, he removed the pan from the fire and laid it on the hearth. And it is a good thing that he did, because before his eyes the parts of the body joined together, and a living man – or his ghost – stood before him! And that was a sight that might have startled Esteban into burning his fingers with the bacon fat.

"Good evening," said Esteban. "Will you have an egg and a bit of bacon?"

"No, I want no food," the ghost answered. "But, I will tell you this, right here and now. You are the only man, out of all those who have come to the castle, to stay here until I could get my body together again. The others died of sheer fright before I was half finished."

"That is because they did not have sense enough to bring food and fire with them," and Esteban replied coolly. And he turned back to his frying pan.

"Wait a minute!" pleaded the ghost. "If you will help me a bit more, you will save my soul and get me into the Kingdom of Heaven. Out in the courtyard, under a cypress tree, there are buried three bags – one of copper coins, one of silver coins, and one of gold coins. I stole them from some thieves and brought them here to the castle to hide. But no sooner did I have them buried than the thieves overtook me, murdered me, and cut my body into pieces. But they did not find the coins. Now you come with me and dig them up. Give the copper coins to the Church, the silver coins to the poor, and keep the gold coins for yourself. Then I will have expiated my sins and can go to the Kingdom of Heaven."

This suited Esteban. So he went out into the courtyard with the ghost. And you should have heard how the donkey brayed when he saw them!

When they reached the cypress tree in a corner of the courtyard: “Dig,” said the ghost.

“Dig yourself,” answered Esteban.

So the ghost dug, and after a time the three bags of money appeared.

“Now will you promise to do just what I asked you to do?” asked the ghost.

“Yes, I promise,” Esteban answered.
“Then,” said the ghost, “strip my garments from me.”

This Esteban did, and instantly the ghost disappeared, leaving his clothes lying there on the short grass of the courtyard. It went straight up to Heaven and knocked on the gate. Saint Peter opened it, and when the spirit explained that he had expiated his sins, gave him a cordial welcome.

Esteban carried the coins into the great hall of the castle, fried and ate another egg and then went peacefully to sleep before the fire.

The next morning when the village people came to carry away Esteban’s body, they found him making an omelet out of the last of the fresh eggs.

“Are you alive?” they gasped.

“I am,” Esteban answered. “And the food and the faggots lasted through very nicely. Now I will go the owner of the castle and collect my thousand gold reales. The ghost has gone for good and all. You will find his clothes lying out in the courtyard.”

And before their astonished eyes he loaded the bags of coins on the donkey’s back and departed.

First he collected the thousand gold pieces from the grateful lord of the castle. Then he returned to Toledo, gave the copper coins to his church, and faithfully distributed the silver ones among the poor. And on the thousand gold pieces he lived in idleness and great contentment for many years.

Scene 1:

In the village – Esteban the gypsy is talking to villagers.

Esteban: Who lives in that huge castle?

Villager 1: No one – it's haunted.

Esteban: Oh please.

Villager 2: No really, lots of people have DIED trying to get rid of that ghost.

Villager 1: If you're brave enough to venture close to the castle, you can hear the ghost at night wailing and carrying on.

Esteban: If I'm brave enough? I fear nothing and nobody. I'll do better than going close to the castle; I'll sleep there.

Villager 1: If you can get rid of the ghost, the owner of the castle will give you a 1000 gold pieces as a reward.

Esteban: Then it's settled. Just bring me some food and drink and some wood for a fire.

Scene 2:

Esteban is sitting in front of his fire in the castle cooking something in a frying pan.

Ghost: "Oh me! Oh me! Oh me!"

Esteban: "That's not a very cheerful greeting, but I guess it's alright for someone used to having a donkey for company"

Ghost: "Oh me! Oh me! Oh me!"

Esteban eats from his pan.

Ghost: "Look out below, I'm falling"

Esteban: "Fall away, just don't spill my egg!"

Ghosts leg falls from the sky and lands next to Esteban.

Esteban looks at the leg, shakes his head and goes back to eating.

Ghost: Look out below, I'm falling

Esteban: All right, just don't fall into the frying pan

Ghost's leg falls from the sky and lands next to Esteban

Esteban looks at the leg and then goes back to cooking over the fire

Ghost: Look out below, I'm falling

Torso falls.

Esteban doesn't bother to look, he just eats his food.

Ghost: Look out below, I'm falling

An arm falls

Ghost: Look out below, I'm falling

Another arm falls

Esteban looks over at the body

Esteban thinks: Now he just needs a head – wonder what that will look like.

Ghost: Look out below, I'm falling

Head falls. Esteban looks over at body
Ghost stands up

Esteban: Good evening, would you like some food?

Ghost: No, I want no food. But I'll tell you this, you're the first to stay here until I could get my body together. The others have all died of shock before I was even halfway through.

Esteban: Guess they didn't have the good sense to bring fire and food.

He turns back to the fire

Ghost: Wait – if you'll help me, you can save my soul and get me into Heaven. Out in the courtyard are three buried bags – one copper, one silver, and one gold. I stole them from some thieves, but they caught me and cut me to pieces. They never found the coins though. Give the copper to the church, the silver to the poor and keep the gold ones for yourself.

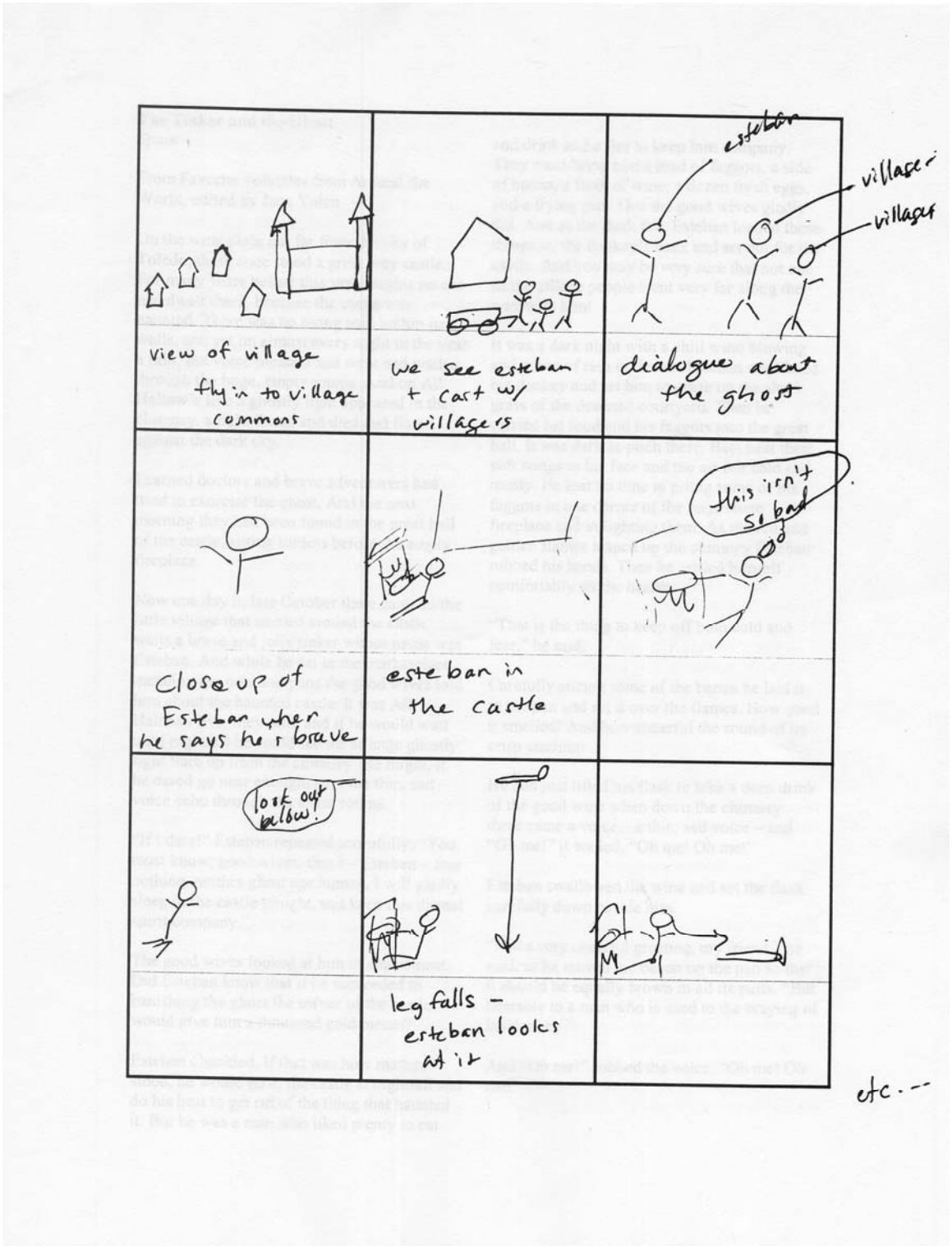
Epilogue –

Esteban did as the ghost said and lived out his days as a rich man.

The ghost went to heaven.

The owner of the castle finally got to move in.

The end.


esteban
villager
villager
view of village
fly in to village commons
we see esteban + cast w/ villagers
dialogue about the ghost
this isn't so bad
Close up of Esteban when he says he's brave
esteban in the castle
look out below!
leg falls - esteban looks at it
etc. --

Worksheet 3

The third storyboarding worksheet guided participants through a 3-step storyboarding process. In the first step, they created a DVD-box description of their story. In the second step, they broke their story into scenes and created a more detailed description of each scene. Finally, participants drew a storyboard for each scene.

Name:
Computer:

Storyboarding

Step 1- Story Description: Write a short (2-3 sentences) high-level description of your movie below. You may want to think of this as the description that would appear on the DVD box for your movie.

Examples:

Jennifer and her friends have never been part of the popular crowd. Despite their cruelty to Jennifer and her friends, Jennifer has always wanted to be friends with the popular girls. When her wish is unexpectedly granted, will she remain true to her old friends?

Melly and her dog are training to compete in a human-dog talent competition. But, when Melly's enemy and main competition see how far they've come, she decides to steal Melly's dog to prevent them from entering.

Patrick and Tina meet one day in the park and fall madly in love. Unfortunately, both are dating other people. By sheer coincidence, both Patrick's girlfriend and Tina's boyfriend can't go to the dance on Friday, and they decide to go together. Both are shocked by who they meet.

A group of soccer players on a trip to the state playoffs decide to go hiking in a nearby park to relax before the big game. Will they manage to find their way back in time for the state final or will their team have to forfeit?

Step 2- Scene Breakdown: Now that you have a basic overview of your story, you need to decide what scenes you'll use to tell your story. For the purposes of this class, your story should have no more than four scenes. List out the scenes in your story and briefly describe each one. Each of your scenes should have a specific purpose in the overall context of your story.

As an example, let's look at the wish story from the previous page:

Jennifer and her friends have never been part of the popular crowd. Despite their cruelty to Jennifer and her friends, Jennifer has always wanted to be friends with the popular girls. When her wish is unexpectedly granted, will she remain true to her old friends?

One way to present this story is….

Scene 1

Purpose: show that Jennifer and her friends are unpopular
Setting: school hallway
Action: Jennifer asks one of the popular girls if she's going to go to the school play on Friday. The girl refuses to acknowledge her. Jennifer's best friend, Hilary comforts her saying "You're a great person, why would you want to be friends with someone so mean"

Scene 2

Purpose: show Jennifer wishing to become popular
Setting: in the park
Action: Jennifer stands at a wishing well and wishes that she could be popular. She tosses in a penny.

Scene 3

Purpose: show Jennifer having become popular
Setting: school hallway
Action: Jennifer and Hilary are talking. One of the popular girls approaches Jennifer and asks if she wants to come to the mall with them after school. Jennifer realizes that her wish has come true.

Scene 4

Purpose: show Jennifer ignoring her former best friend because she's not popular
Setting: skate park
Action: Hilary sees Jennifer with her new group of friends. Hilary walks over to them and asks Jennifer if they can talk for a minute. Another girl in the group asks "Why would she want to talk to you?" Jennifer answers with "Yeah, why would I?" and turns away. Hilary looks sad and walks away.

Please use this sheet to list and describe the scenes in your movie.

Scene 1

Purpose:

Setting:
Action:

Scene 2

Purpose:

Setting:
Action:

Scene 3

Purpose:

Setting:
Action:

Scene 4

Purpose:

Setting:
Action:

Scene 5

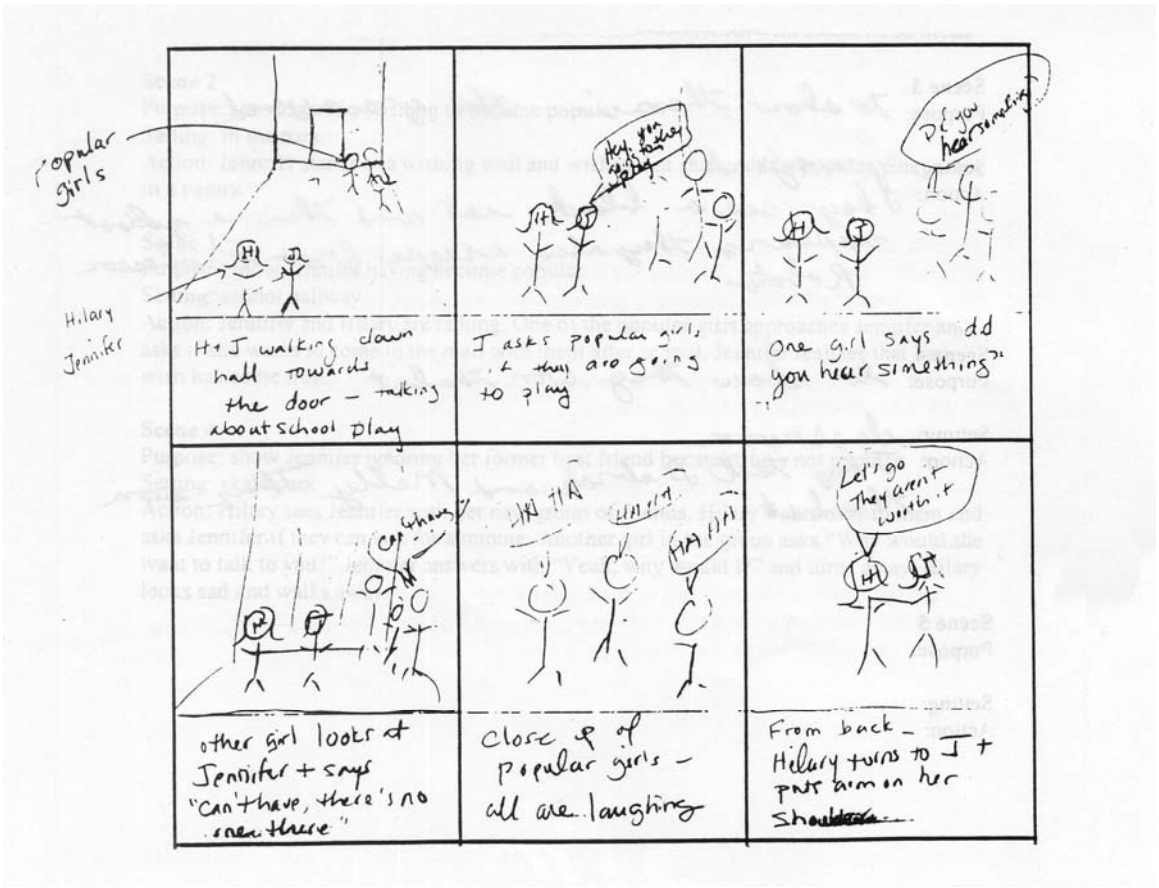
Purpose:

Setting:
Action:

Step 3 – Drawing Storyboards: At this point, you should be ready to begin sketching out what you think your movie should look like. In the movie industry, this process is known as creating a storyboard. Before studios invest money in actors and sets, either digital or real, studios create storyboards. Storyboards can help movie directors and producers to work out problems with the story or how it will be presented without needing actors and sets.

When you create your storyboards you should be thinking about where your characters start in the scene, how they move, what they do, and what they say. You should also think about where you want the camera to be – shots from far away can help your audience to better understand where the scene takes place, close-up shots may help your audience to feel more connected with the characters.

Below is what the first scene of the wish story might look like in storyboard form. Please note: these are all stick figures, not major artistic endeavors. The point of making a storyboard is to help you plan out how your movie will look. Your storyboards don't need to be beautiful: stick figures are fine; and, if you run into something that's hard to sketch quickly, you can describe it with words. As in the example below, you should write a quick textual description underneath each picture to indicate what's going on, who's talking, etc.


popular girls
Hilary
Jennifer
H+J walking down hall towards the door – talking about school play
J asks popular girls if they are going to play
One girl says "did you hear something"
other girl looks at Jennifer + says "can't have, there's no room there"
Close up of popular girls – all are laughing
From back – Hilary turns to J + puts arm on her shoulder
Let's go They aren't worth it

Appendix B: Surveys and Programming Quiz

11.9 Pre-Workshop Survey

This was given to all participants of the summative evaluation at the beginning of the evaluation workshops.

Pre-Workshop Survey

Name:
Age:
Grade in School:

What kind of school do you go to? (Please circle one answer)

a) a public school
b) a private school
c) I am home-schooled

How would you describe the grades on your report card? (Please circle one answer)

a) Mostly A's
b) A's and B's
c) Mostly B's
d) B's and C's
e) Mostly C's
f) C's and D's
g) Mostly D's and below.
h) I don't get grades.

What are your favorite subjects in school? (Please circle all that apply)

a) English
b) History
c) Math
d) Science
e) Foreign Language
f) Government
g) Art
h) Music
i) Other: _____

During the last week (counting yesterday and backwards 6 days), how often did you use a computer for any purpose?
Hours last week _____

What do you use computers for?

a) Only for schoolwork
b) Mostly for schoolwork and some for fun
c) About equally for schoolwork and fun
d) Mostly for fun and some for schoolwork
e) Only for fun.

What is your skill level at using computers?

a) Poor or nonexistent
b) Fair
c) Good
d) Very good
e) Excellent

Have you ever written a computer program?

a) Yes
b) No
c) Don't know

Have you ever made your own web page?

a) Yes
b) No
c) Don't know

When something goes wrong with your computer, how frequently do you ask friends or family members for help fixing it?

a) Very frequently
b) Somewhat frequently
c) Neither frequently nor infrequently
d) Somewhat infrequently
e) Very infrequently

When you want to install a new computer program, how frequently do you ask friends or family members to help you install it?

a) Very frequently
b) Somewhat frequently
c) Neither frequently nor infrequently
d) Somewhat infrequently
e) Very infrequently

Do you think you could learn a computer language like Java or C++?

a) Definitely not
b) Probably not
c) Maybe yes, maybe no
d) Probably yes
e) Definitely yes

Would you be interested in taking a computer science class in high school?

a) Definitely not
b) Probably not
c) Maybe yes, maybe no
d) Probably yes
e) Definitely yes

11.10 Post-Workshop Survey

After participants completed their 135 minutes working with their assigned version of Alice, they were asked to complete a second survey.

Post-workshop Survey

Name:

There are several statements about using the computer during the workshop today. Please put an 'X' in one of the boxes to indicate whether you agree or disagree with each statement.

	Strongly Disagree	Disagree	Not Sure	Agree	Strongly Agree
1 Using the computer during the workshop today was **fun**.					
2 Using the computer during the workshop today was **interesting**.					
3 Using the computer during the workshop today was **frustrating**.					
4 Using the computer during the workshop today was **boring**.					

There are several statements about the computer animation program you used during the workshop today. Please put an 'X' in one of the boxes to indicate whether you agree or disagree with each statement.

	Strongly Disagree	Disagree	Not Sure	Agree	Strongly Agree
1 The computer animation program I used today is **confusing**.					
2 The computer animation program I used today is **cool**.					
3 The computer animation program I used today is **annoying**.					
4 The computer animation program I used today is **easy to learn**.					
5 The computer animation program I used today is **entertaining**.					

If you used Alice (the computer animation program you used today) again, how long do you think you could use it at one time without getting bored? (Please circle one answer)

a) Less than 1 hour
b) 1-2 hours
c) 2-3 hours
d) 3-4 hours
e) More than 4 hours

Alice is currently being used to teach high school and college students. In what kinds of classes is Alice used most frequently?

a) Science classes
b) Computer science classes
c) Art classes
d) Communications classes

If you had the computer animation program you used today ("Alice") on a computer at home, how often during the next month do you think you would use it? (Please circle one answer)

a) Never
b) Once during the next month
c) Twice or three times during the next month
d) Once a week during the next month
e) More than once a week during the next month

Would you be interested in taking another Alice class?

a) Definitely not
b) Probably not
c) Maybe no, maybe yes
d) Probably yes
e) Definitely yes

Do you think you could create a world in Alice that you would be proud to show your friends?

a) Definitely not
b) Probably not
c) Maybe no, maybe yes
d) Probably yes
e) Definitely yes

Do you think you could learn to use advanced features in the Alice program?

a) Definitely not
b) Probably not
c) Maybe no, maybe yes
d) Probably yes
e) Definitely yes

Do you think you could learn a computer language like Java or C++?

f) Definitely not
g) Probably not
h) Maybe yes, maybe no
i) Probably yes
j) Definitely yes

Would you be interested in taking a computer science class in high school?

f) Definitely not
g) Probably not
h) Maybe yes, maybe no
i) Probably yes
j) Definitely yes

Can you imagine growing up to be a computer scientist?

a) Definitely not
b) Probably not
c) Maybe no, maybe yes
d) Probably yes
e) Definitely yes

How likely is it that you will tell anyone about your experience in the workshop today?

a) Very unlikely
b) Somewhat unlikely
c) Neither unlikely nor likely
d) Somewhat likely
e) Very likely

If you do plan to talk about your experience today, what will you say? (Please write your answer below.)

What are the 3 best things about Alice?

1.

2.

3.

What are the 3 worst things about Alice?

1.

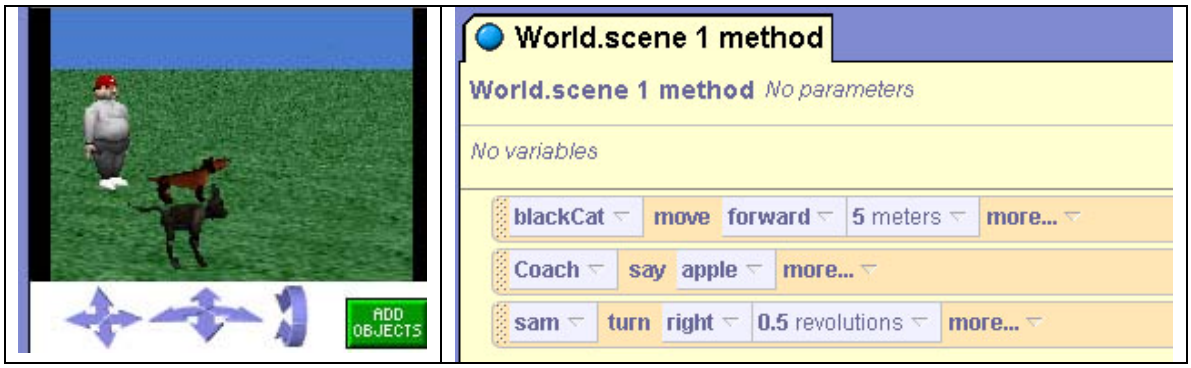
2.

3.

11.11 Programming Quiz

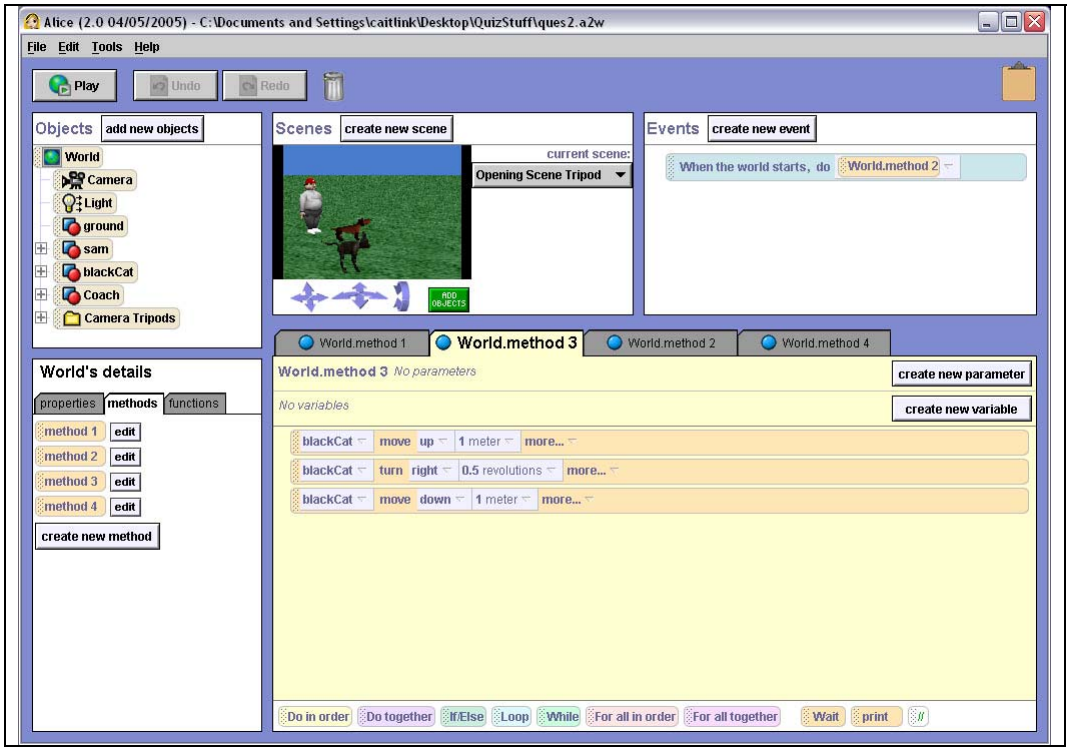
After the post-survey, participants completed a short programming quiz.

1. **If you were to play the following 3 lines of code in Alice, which of the following best describes what would happen?**



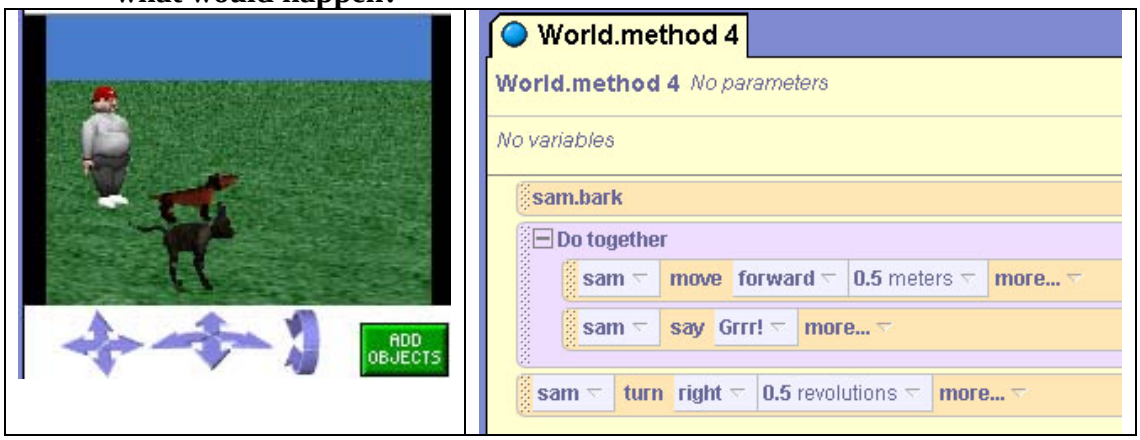
   a. First the black cat moves, then coach says apple, and finally sam turns.
   b. First the coach says apple, then the black cat moves, and finally sam turns.
   c. First sam turns, then the black cat moves, and finally the coach says apple.
   d. First the black cat moves, then sam turns, and finally the coach says apple.

2. **If you were to hit the play button for the Alice world pictured above, which animation would you expect Alice to play?**



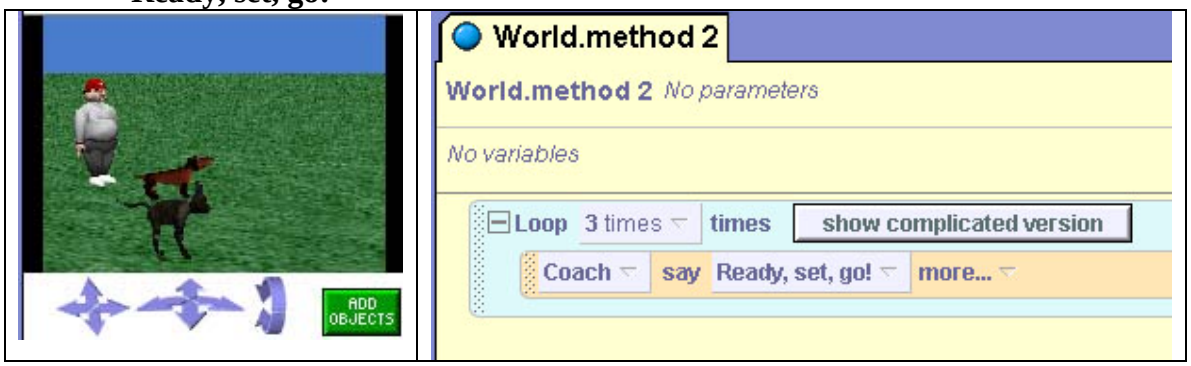
   a. Alice would play method 1.
   b. Alice would play method 2.
   c. Alice would play method 3.
   d. Alice would play method 4.

3. **If you were to play this method in Alice, which of the following best describes what would happen?**



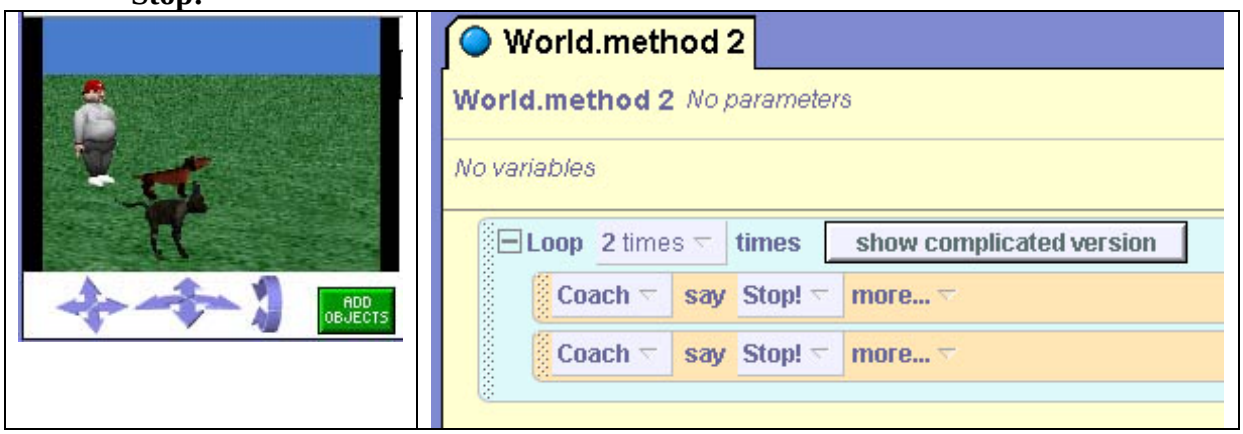
   a. Sam would bark, move forward, say "Grrr!", and turn right all at the same time.
   b. First sam would bark, then sam would move forward and say "Grrr!" at the same time, and finally sam would turn right ½.
   c. First sam would bark, then sam would move forward, then sam would say "Grrr!", and finally sam would turn right ½.
   d. First sam would bark, then sam would move forward, say "Grrr!" and turn right at the same time.

4. **If you were to play this method in Alice, how many times would the coach say "Ready, set, go!"**



   a. 1 time.
   b. 2 times.
   c. 3 times.
   d. 4 times.

5. **If you were to play this method in Alice, how many times would the coach say "Stop!"**

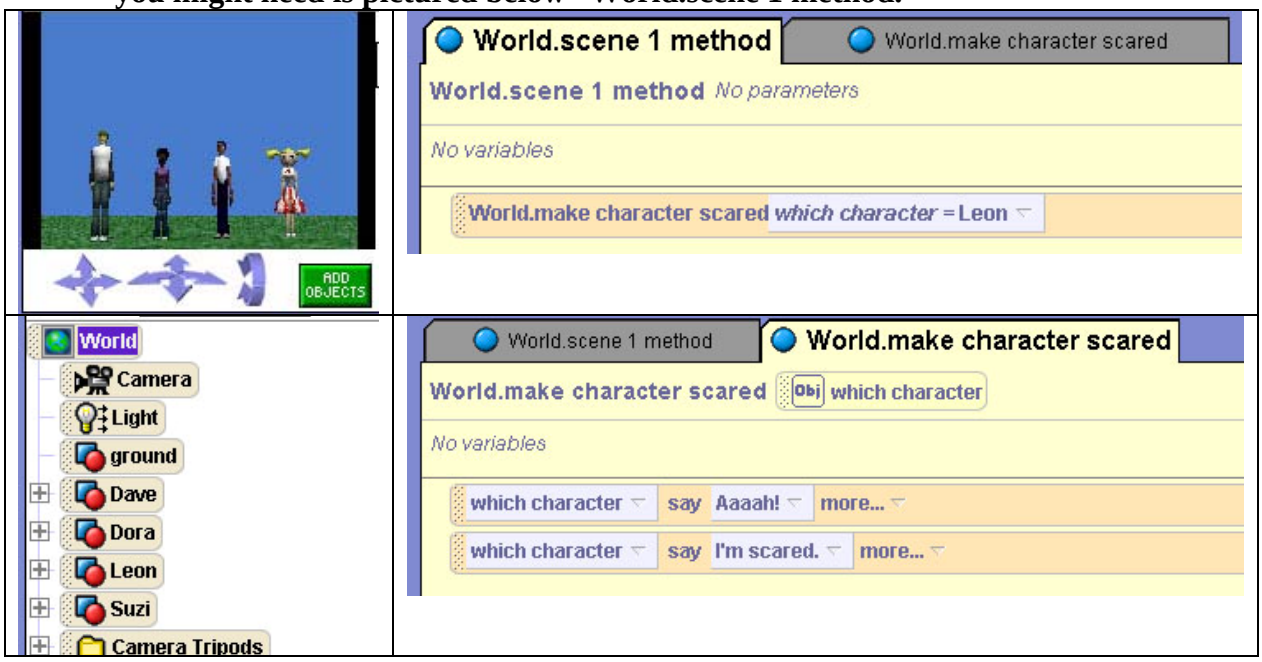


a. 1 time.
b. 2 times.
c. 3 times.
d. 4 times.

6. **If you were to play "World.scene 1 method" (below) in Alice, which words would Joey say?**

a. First, he would say "Sign," and then "Library."
b. First, he would say "Library," then he would say "Belt," and finally he would say "Sign."
c. First, he would say "Library," and then "Sign."
d. First he would say "Belt," then he would say "Sign", and finally he would say "Library."

7. **If you were to play "World.scene 1 method" (below) in Alice, which character would say "Aaaah!" and "I'm scared?" Some extra information you might need is pictured below "World.scene 1 method."**



a. None of the characters
b. Dave
c. Dora
d. Leon
e. Suzi

Appendix C: Generic and Storytelling Alice Reference Booklets

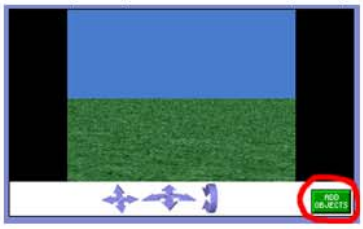
11.12 Generic Alice Reference Booklet

Writing Alice Programs

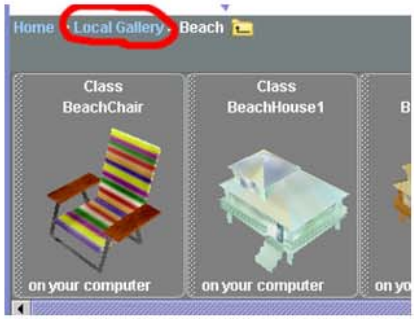
www.alice.org

Basic Alice Programming

Adding Objects and Navigating the Gallery



Getting to the Alice gallery:
To get to the Alice gallery, click on the add objects button.



Adding Objects:
Once you're in the Alice gallery, you can add objects to your Alice world by dragging cards into your scene.

Getting Back to the Main Gallery:
To get back to the main gallery (where you can see all the different kinds of objects) click on "Local Gallery."

Positioning Objects

To move objects around in your Alice worlds, you can click on them and drag them around the scene. This will move them around on the ground plane.

You can change the mouse so that it moves objects up and down, turns them to face another

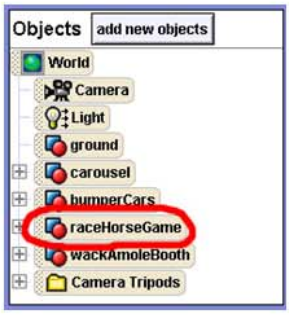
direction, copies them, or resizes them, you can use the controls on the right.

To change what the mouse does, click one of the buttons on the right. Then, you can click and drag objects in the scene like you do to move them around the scene.

To make the mouse move objects over the ground again, select the arrow button above. Then, you can click and drag objects in the scene again.

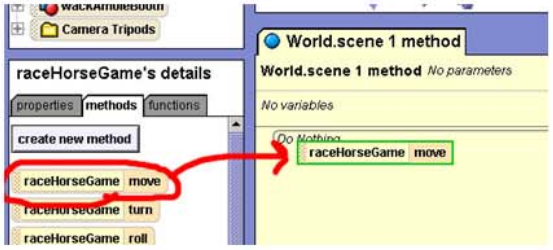
Basic Alice Programming

Once you've got a scene laid out in Alice, you'll probably want to animate it.



Every object in your Alice scene also has a tile in the object tree. **If you want to know what methods the raceHorseGame can do, click on the raceHorseGame tile in the object tree.**

Then, you'll see a list of the methods that the raceHorseGame knows how to do.



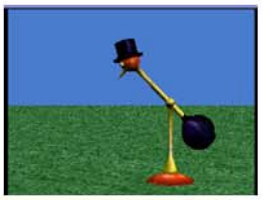
To add an animation, drag and drop the animation tiles into the method editor.

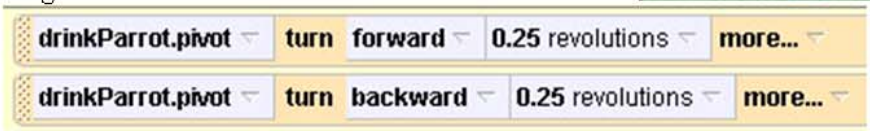
When you've added some animations, you hit the "Play" button to see what your Alice program looks like.

Using Loops to do the same thing again and again and again…

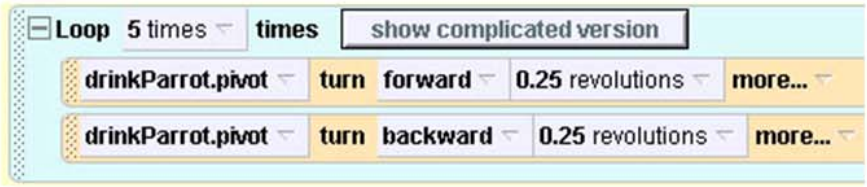
Sometimes you'll want to take an action or a few actions and do them several times in a row. To do this you can use something called a Loop.

Suppose we have this drinking parrot that we want to animate. If you've ever seen one of these in person, you'll know that they lean forward and then return to standing as though they are drinking from an imaginary glass of water over and over again.





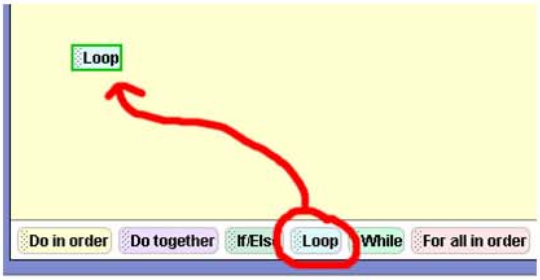
The two lines above tell the drinking parrot to turn forward (to take a sip) and then return to its normal position. So, how do we make the drinking parrot take lots and lots of sips?



To make the drinking parrot take lots of sips, we can put the two lines that make the parrot take one sip inside a loop (like the picture above). Then the parrot will take as many sips as the loop happens. In this case, the loop will happen 5 times (you can change how many times a loop happens by clicking on the little triangle next to the number of times the loop happens).

Add a Loop to your Alice program!

1. Drag the "Loop" tile from the bottom of your method editor and drop it into your method.
2. Choose how many times you want it to happen!
3. Then put the lines of code you want to repeat inside the loop box.



Using Do Togethers to make multiple things happen at the same time…

You may find that there are times when you need to have two things happening at the same time.

Suppose we want to animate this helicopter. Helicopters use their propellers to fly. So, we'd like to have the helicopter's propeller spin and the helicopter move up.

As a first try, we might end up with something like these lines:

But, if you play this program in Alice, you will find that the things don't do quite what you want. First, the propellers spin. Then, the helicopter moves up. We'd really like these to happen at the same time. To accomplish this, we can add something called a Do Together.

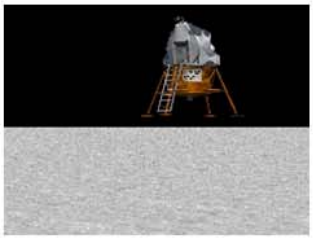
If we play this program in Alice, then the helicopter's propeller will spin and the helicopter will move up at the same time!

Add a Do Together to your Alice program!

1. Drag the "Do Together" tile from the bottom of your method editor and drop it into your method.
2. Then, put the lines of code you want to have happening at the same time inside the Do Together box.

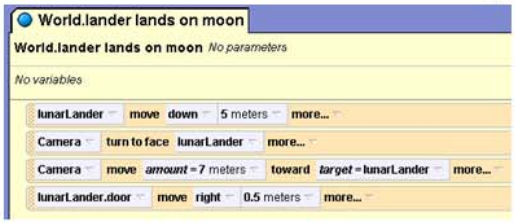
Using Methods to organize your Alice programs…

When you are creating longer programs in Alice, you may find that it's useful to be able to write your program in smaller sections.



Say you want to create an Alice program that shows an astronaut walking on the moon. This program might have several parts: 1) the lunar lander landing on the moon 2) the astronaut coming out of the lander and 3) the astronaut walking on the moon. We can each of these as separate methods so that we can work on them one a time.

Say we'd like to create the first method – the lunar lander landing on the moon. To do this, make sure you have "world" selected in the object tree. Then, click on "create new method" and type in "lander lands on moon."

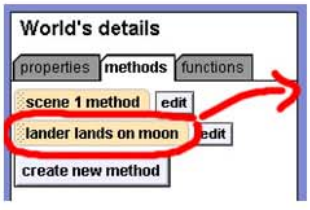


Then, you can define what it means for the lander to land on the moon in the editor. Here's one possibility for what that might look like.

Add a method to your Alice program!

You can use a method you've created in one of two ways: making it run when you hit the play button and calling it in another method.

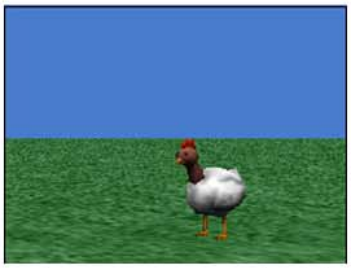
1. Click on the triangle next to the name of the method being played to change which method runs when you hit the play button.



2. To use "lander lands on moon" as part of another method, you can drag it into your method.

Teaching Alice Objects to Do New Methods

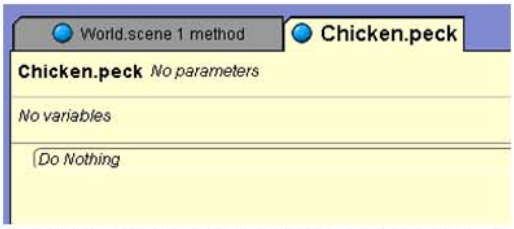
Sometimes it's useful to teach Alice objects to do new actions (or methods).



Perhaps you are creating an Alice world in which it would be useful to have the chicken peck.

First, click on the "Chicken" tile in the object tree so that you can see what methods the Chicken knows how to do.

Then, click on the "create new method" button and give you new method a name. Since we want to teach the chicken to peck, we'll call the method "peck."



Alice creates a new method editor for you where you can define what it means for the chicken to peck.

To get the chicken to peck you probably want to move just the chicken's neck, not its whole body. To see what parts the chicken has, click on the little plus signs next to the chicken tile.

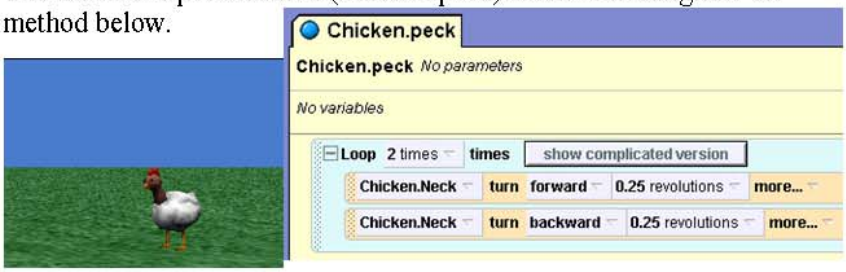
To find the methods that the chicken's neck can do, click on the chicken's "Neck" tile.

When thinking about how to animate parts of an object, most people want to use the "move" animation. Sadly, move causes object's parts to fall off of them. So, mostly you'll want to use the turn and roll animations. You can experiment with them to see how they work.

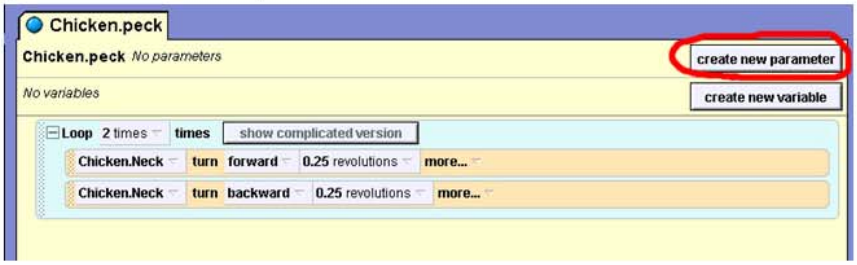
You can use a character method in your Alice world the same way that you use a world method, so take a look at the "Add a Method to your Alice program!" section on the previous page.

Using parameters to make object methods more flexible…

Let's suppose that we've taught the Chicken object in Alice how to peck. The Chicken's peck method (Chicken.peck) looks something like the method below.



Right now, peck will always have the Chicken peck 2 times. What if we wanted to be able to tell the Chicken how many times we wanted him to peck when we called the method Chicken.peck? We can do this if we add a parameter to Chicken.peck.



Click on the "create new parameter" button and type "howManyTimes" into the Name box. Our parameter "howManyTimes" is a number, so make sure that "Number" is selected in the Type box.

Then, just drag the "howManyTimes" tile and drop it on top of "2 times."

Now, when you call the method "Chicken.peck," you can tell Alice how many times you want the chicken to peck.

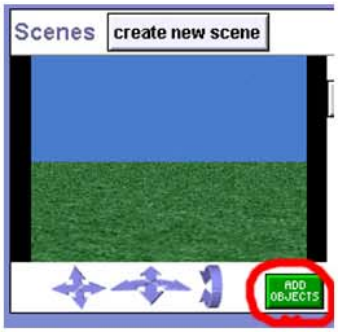
11.13 Storytelling Alice Reference Booklet

Creating Animated Movies in Alice

www.alice.org

Alice Basics

Adding Objects and Navigating the Gallery



Getting to the Alice gallery:
To add characters and scenery to your Alice world, click on the add objects button.

Adding Objects:
Once you're in the Alice gallery, you can add characters and scenery to your Alice world by dragging cards into your scene.

Getting Back to the Main Gallery:
To get back to the main gallery click on "Local Gallery." From the local gallery, click on the "Characters" and "Scenes" folder to see the characters and scenes you can use in your stories.

Positioning Objects

To move characters or objects around in your Alice worlds, you can click on them and drag them around the scene. This will move them around on the ground plane.

You can change the mouse so that it moves objects up and down, turns them to face another direction, copies them, or resizes them, you can use the controls on the right.

To change what the mouse does, click one of the buttons on the right. Then, you can click and drag objects in the scene like you do to move them around the scene.

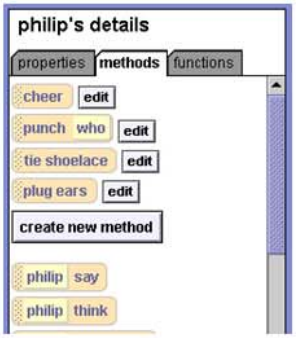
To make the mouse move objects over the ground again, select the arrow button above. Then, you can click and drag objects in the scene again.

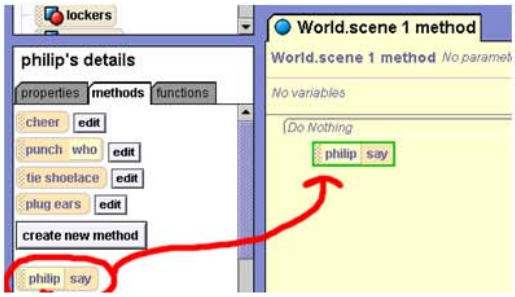
Basic Alice Programming

Once you've got a scene laid out in Alice, you'll probably want to animate it.

Every character and object in your Alice scene also has a tile in the object tree. If you want to know what methods Philip can do, click on the "Philip" tile in the object tree.

Then, you'll see a list of the methods that Philip knows how to do.





To add an animation for Philip, drag and drop the animation tiles into the method editor.

When you've added some animations, you can hit the "Play" button to see what your Alice program looks like.

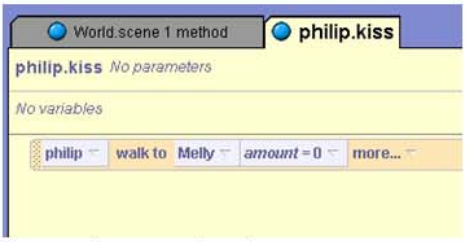
Teaching Philip to Kiss Part 1: First Steps

Suppose that you are telling a story about a boy named Philip and a girl named Melly falling in love. In one scene, Philip needs to kiss Melly.

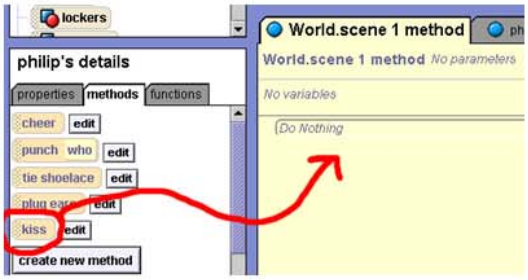
But, when you go and look at what Philip knows how to do, you discover that Philip doesn't know how to kiss. In Alice, you can teach your characters to do new things by creating new methods for them. To create a new method for Philip, select Philip in the object tree and then click on his "create new method" button, give your animation a new name (like "kiss") and press the OK button.

Alice will open a new method editor for Philip.kiss. In this new editor, you can teach Philip how to kiss.

The first step in kissing someone is probably walking up to that person. Since, this story is about Philip and Melly, we'll have Philip walk up to Melly. Since they're supposed to kiss, use the "more" menu to set the amount (distance between them) to 0.



Obviously, we'll need to add some more details to philip's kiss. But, go ahead and add Philip.Kiss to your story, so that you can see how to use it.

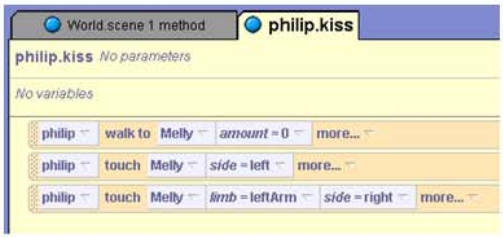


If you look at the methods that Philip knows how to do, you'll find that there's a new tile called "kiss." Drag this into your main story (not Philip.kiss) and hit the play button. Philip should walk to Melly.

Teaching Philip to Kiss Part 2: Embracing Melly

The next step in getting Philip to kiss Melly is to have him put his arms around her. We can use the touch method to get Philip to touch the left and right sides of Melly.

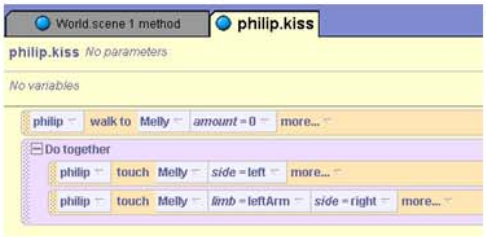
You'll need to use the more menu to set which *side* of Melly Philip will touch and which *limb* (right arm or left arm) he will use.



Then, we might end up with a Philip.kiss that looks like this. Try playing this to see what it looks like.

It looks a little awkward because Philip raises his hands one at a time. We can use something called a "Do together" to make the two touch animations happen at the same time.

To add a "Do together" to your Philip.kiss, drag the "Do together" tile from the bottom of the method editor and drop it at the end of Philip.kiss.

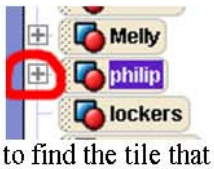


Do togethers do all the animations inside them at the same time. Since we want the two touch animations to happen at the same time, drag those inside the "Do together." Now, Philip.kiss should look like this.

If you play Philip.kiss, Philip will walk over to Melly and put his arms around her.

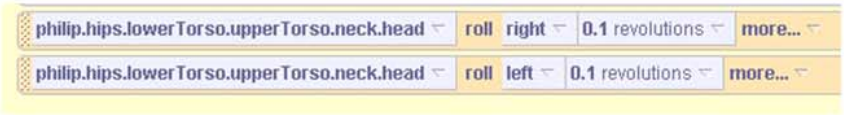
You can use "Do togethers" anytime you have multiple animations that you want to have happen at the same time

Teaching Philip to Kiss Part 3: The Kiss!



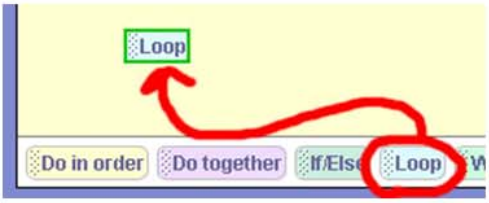
Now that we've got Philip standing in front of Melly with his arms around her, let's get him to tilt his head back and forth a bunch of times to kiss her. The first thing we'll need to do is figure out how to get him to tilt his head. To do this, we'll need to find the tile that represents Philip's head. You can see Philip's body parts by clicking on the "+" sign next to his tile. You may have to open up some of Philips' body parts to find his head.

When you find Philip's head, click on it to see what methods it can do. When you want to move a body part around, you'll want to use either turn or roll. In this case we want roll which will make Philip tilt his head left and right.

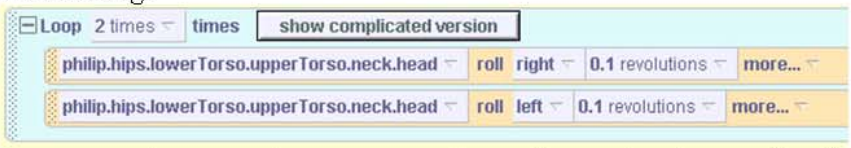


If you drag in Philip's "head turn" tile, you can add these two animations to the bottom of Philip.kiss.

If you play this, you'll see that Philip now kisses Melly by tilting his head right and left once. But, we'd really like him to do tilt his head right and left multiple times. To do this, we can add something called a Loop. Loops do all the animations inside them multiple times.



Add a loop to the end of Philip.kiss and move Philip's two head rolls inside of the loop. The end of Philip. Kiss should look something like this. You can change how many times the loop happens by changing "2 times" to something.

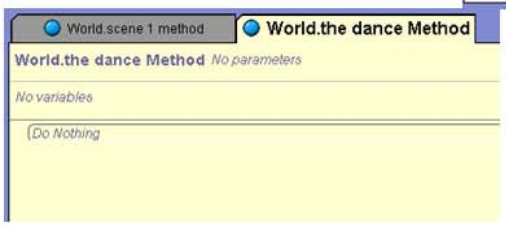


You can use loops any time you want Alice to do an animation or a list of animations multiple times in a row.

Creating New Scenes

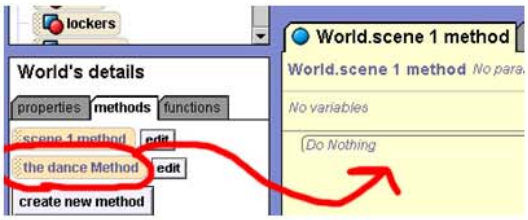
Often, you'll find that you need multiple locations in your story. For example, one part of your story might take place in a classroom and another part might take place in a garden. In Alice, we call these locations scenes. To create a new scene, click on the "create new scene" button. Give your new scene a name and choose what kind of ground you want. Then press "OK."

Alice will create a new blank scene for you to but your characters and scenery into. You can use the "current scene" drop down to choose which scene you want to look at.



Since you'll probably want to watch your scenes one at a time, Alice will also make a new method and open an editor where you should put the action for your new scene.

When you press the Play button, Alice looks at the events area to figure out which animations it should play. If you've created a new scene called "the dance," you'll want to change what method Alice plays when the world starts to be "the dance Method" by clicking on the triangle on the right side.

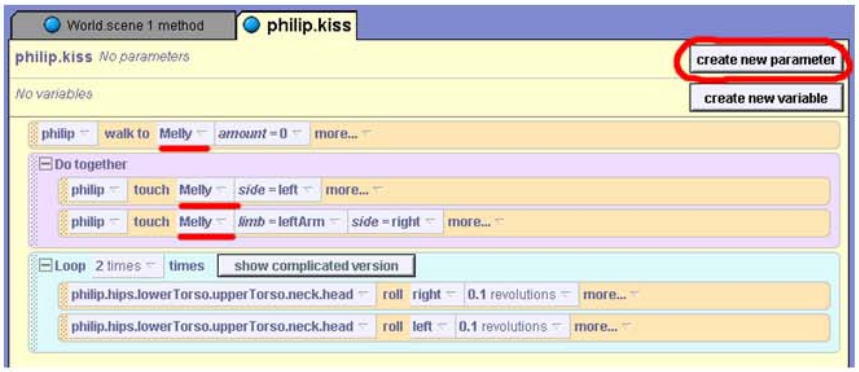


If you click on "World" in the object tree, you will find that there's a new tile for "the dance Method" that you can drag in to your story (although you'll need to put some action into it first!)

Teaching Philip to Kiss Part 4: Kissing Anyone!

If you walked up to a friend named Philip at school and said "Philip kiss." He would probably say "Kiss who?" Right now when we say "Philip kiss," Philip will always go kiss Melly. But, wouldn't it be cool if he could kiss anyone? Then we'd be able to say "Philip kiss Melly." But we could also say "Philip kiss Teacher" or "Philip kiss Ogre."

Let's look at our current Philip.kiss. We need to replace the places where it says Melly (they're underlined) with something like "the person you're going to kiss." In Alice, you can do this by creating something called a parameter. A parameter allows you to say, "Philip this is how you kiss



someone and I'll tell you who that's going to be later." To create a parameter, click on the "create new parameter" button, call the parameter "kissee" and make sure it's an "object" not a number.

When you've created your parameter, you should get a little tile at the top of your method. Drag this tile and drop it on all the Mellys to replace them.

Now, Philip.kiss should look like this.

In your story, where you called Philip.kiss you now need to choose who Philip should kiss.

References

(1993). The Incredible Machine, Sierra Games.

(1995). My Make Believe Castle, Logo Computer Systems, Inc.

(1995). Thinkin' Things Collection 3: Half Time, Edmark Corporation.

(1995). Widget Workshop, Maxis.

(1998). Lego Mindstorms Robotics Invention System, LEGO Systems, Inc.

(1999). Alice 99, Carnegie Mellon University.

(2001). Mindrover, Cognitoy.

(2002). Magic Forest, Logotron.

(2003). Alice 2, Carnegie Mellon University.

AAUW (1992). How Schools Shortchange Girls: A Study of Major Findings on Girls and Education. New York, NY, Marlowe & Company.

AAUW (1996). Girls in the Middle: Working to Succeed in School. Washington, DC, American Association of University Women Educational Foundation.

AAUW (1998). Gender Gaps: Where Schools Still Fail Our Children. Washington, DC, American Association of University Women Educational Foundation.

AAUW (2000). Tech-Savvy: Educating Girls in the New Computer Age. Washington, DC, American Association of University Women Educational Foundation.

Apple Designing Coachmarks.

Atkinson, B. (1987). Hypercard, Apple Computer.

Aubel, A., R. Boulic, et al. (2000). "Real-time display of virtual humans: levels of details and imposters." Circuits and Systems for Video Technology **10**(2): 207-217.

Baecker, R. (2002). Showing Instead of Telling. International Conference on the Design of Communication, ACM Press.

Bartlett, J. (1992). Transparent Controls for Interactive Graphics. Palo Alto, CA, Digital Equipment Corporation.

Becker, B. (2004). Robots: Learning to Program with Java. Waterloo, Self-published.

Begel, A. (1996). LogoBlocks: A Graphical Programming Language for Interacting with the World. Electrical Engineering and Computer Science Department. Boston, MA, MIT.

Begel, A. (1997). Bongo: A Kids' Programming Environment for Creating Video Games on the Web. Electrical Engineering and Computer Science Department. Cambridge, MA, MIT.

Bell, B. and C. Lewis (1993). ChemTrains: A Language for Creating Behaving Pictures. IEEE Symposium on Visual Languages.

Bergin, J., M. Stehlik, et al. (1996). Karel++: A Gentle Introduction to the Art of Object-Oriented Programming. New York, NY, John Wiley & Sons, Inc.

Bergin, J., M. Stehlik, et al. (2001). Karel J. Robot: A Gentle Introduction to the Art of Object-Oriented Programming.

Bier, E., M. Stone, et al. (1994). A taxonomy of see-through tools. Conference on Human Factors in Computing Systems, ACM Press.

Bier, E., M. Stone, et al. (1993). Toolglass and magic lenses: the see-through interface. Conference on Computer Graphics and Interactive Techniques, ACM Press.

Blackwell, A. and R. Hague (2001). AutoHAN: An Architecture for Programming the Home. IEEE Symposia on Human-Centric Computing Languages and Environments, Stresa, Italy.

Blanchard, C., S. Burgess, et al. (1990). Reality Built for Two: A Virtual Reality Tool. Symposium on Interactive 3D Graphics, Snowbird, Utah.

Booher, H. R. (1975). "Relative comprehensibility of pictorial information and printed words in proceduralized instructions." Human Factors **17**(3): 266-277.

Bruckman, A. (1997). MOOSE Crossing: Construction, Community, and Learning in a Networked Virtual World for Kids. MIT Media Lab. Boston, MA.

Brunner, C., D. Bennett, et al. (1998). Girl Games and Technological Desire. From Barbie to Mortal Kombat. J. Cassell and H. Jenkins. Cambridge, MA, MIT Press**:** 72-88.

Brusilovsky, P. (1991). Turingal - the language for teaching the principles of programming. Third European Logo Conference, Parma, Italy.

Brusilovsky, P., E. Calabrese, et al. (1997). "Mini-languages: A Way to Learn Programming Principles." Education and Information Technologies **2**(1): 65-83.

Budge, B. (1983). Pinball Construction Set, Exidy Software.

Burnett, M., J. Atwood, et al. (2001). "Forms/3: A First-Order Visual Language to Explore the Boundaries of the Spreadsheet Paradigm." Journal of Functional Programming **11**(2): 155-206.

Carroll, J. and J. Carrithers (1984). "Training Wheels in a User Interface." Communications of the ACM **27**(8): 800-806.

Castell, S. d. and M. Bryson (1998). Retooling Play: Dystopia, Dysphoria, and Difference. From Barbie to Mortal Kombat: Gender and Computer Games. J. Cassell and H. Jenkins.

Catlin, D. (1989). Roamer, Valiant Technologies.

Catrambone, R. and J. Carroll (1987). Learning a Word Processing System with Training Wheels and Guided Exploration. Conference on Human Factors in Computing Systems and Graphical Interfaces.

CAWMSET (2000). Land of Plenty: Diversity as America's Competitive Edge in Science, Engineering and Technology. Arlington, VA, Commission on the Advancement of Women and Minorities in Science, Engineering, and Technology.

Chai, J.-x., J. Xiao, et al. (2003). Vision-based control of 3D facial animation. SIGGRAPH/Eurographics Symposium on Computer Animation, San Diego, CA, ACM Press.

Cheng, A. (1998). A Graphical Programming Interface for a Children's Constructionist Learning Environment. Electrical Engineering and Computer Science Department. Boston, MA, Massachusettts Institute of Technology.

Cockburn, A. and A. Bryant (1997). "Leogo: An Equal Opportunity User Interface for Programming." Journal of Visual Languages and Computing **8**(5-6): 601-619.

Cockburn, A. and A. Bryant (1998). Cleogo: Collaborative and Multi-Metaphor Programming for Kids. 3rd Asia Pacific Conference on Computer Human Interaction, Japan.

College_Board (2004). 2004 A.P. Exam National Summary Report.

College_Board (2005). 2005 A.P. Exam National Summary Report.

Collins, W. A. and S. A. Kuczaj (1991). Developmental Psychology: Childhood and Adolescence. New York, NY, Macmillan.

Conway, M. (1997). Alice: Easy-to-Learn 3D Scripting for Novices. School of Engineering and Applied Science. Charlottesville, VA, University of Virginia.

Conway, M., S. Audia, et al. (2000). Alice: lessons learned from building a 3D system for novices. Conference on Human Factors in Computing Systems, The Hague, The Netherlands, ACM Press.

Cooper, S., W. Dann, et al. (2003). Teaching Objects-first in Introductory Computer Science. SIGCSE, ACM Press.

Cypher, A. (1993). Watch What I Do: Programming by Demonstration. Cambridge, Massachusetts, MIT Press.

DeBonte, A. (1998). Pet Park: A Virtual Learning World for Kids. Electrical Engineering and Computer Science. Boston, MA, MIT.

DiGiano, C. (1996). Self-Disclosing Design Tools: An Incremental Approach Toward End-User Programming. Computer Science Department. Boulder, CO, University of Colorado at Boulder.

Dijkstra, E. W. (1969). Structured Programming. Software Engineering Technologies, Rome, Italy.

diSessa, A. and H. Abelson (1986). "Boxer: A Reconstructable Computational Medium." Communications of the ACM **29**(9): 859-868.

Dobbyn, S., J. Hamill, et al. (2005). Geoposters: a real-time geometry / imposter crowd rendering system. Symposium on Interactive 3D Graphics and Games, Washington, DC, ACM Press.

Dossey, J. A., I. V. S. Mullis, et al. (1988). The mathematics report card: Are we measuring up? Princeton, NJ, Educational Testing Service.

Eisenstadt, M. (1983). "A User-Friendly Software Environment for the Novice Programmer." Communications of the ACM **26**(12): 1058-1063.

Ekman, P., W. Friesen, et al. (2002). Facial Action Coding System, A Human Face.

ETC What is the ETC?

Fennema, E. and J. Sherman (1977). "Sex Related Differences in Math Achievement, Spatial Visualization and Affective Factors." American Educational Research Journal **14**: 51-71.

Fenton, J. and K. Beck (1989). Playground: an object-oriented simulation system with agent rules for children of all ages. Proceedings on Object-oriented Programming Systems, Languages and Applications, New Orleans, LA.

Finzer, W. and L. Gould (1984). Programming by Rehearsal. Palo Alto, CA, Xerox Palo Alto Research Center.

Fishkin, K. and M. Stone (1995). Enhanced Dynamic Queries via Movable Filters. Conference on Human Factors in Computing Systems, ACM Press.

Flanagan, M., H. Nissenbaum, et al. (2005). New Design Methods for Activist Gaming. Digital Games Research Association Conference.

Frankel, L. (2002). Research Facts and Findings: Identity Formation in Adolescence.

Frei, P., V. Su, et al. (2000). Curlybot: Designing a New Class of Computational Toys. Conference on Human Factors in Computing Systems, Los Angeles, CA.

Furger, R. (1998). Does Jane Compute?: Preserving Our Daughters' Place in the Cyber Revolution. New York, Warner Books, Inc.

GA (2003). Children, Families and the Internet 2003, Grunwald Associates.

Gilligan, D. (1998). An Exploration of Programming by Demonstration in the Domain of Novice Programming. Computer Science. Wellington, Victoria, Victoria University**:** 176.

Gindling, J., A. Ioannidou, et al. (1995). LEGOsheets: A Rule-Based Programming, Simulation and Manipulation Environment for the LEGO Programmable Brick. IEEE Symposium on Visual Languages, Darmstadt, Germany.

Glinert, E. and S. Tanimoto (1984). "Pict: An Interactive Graphical Programming Environment`." Computer **17**(11): 7-25.

Goldman, K. (2003). A Demonstration of JPie: An Environment for Live Software Construction in Java. Conference on Object-Oriented Programming Systems, Languages, and Applications, ACM Press.

Goldman, K. (2004). "An Interactive Environment for Beginning Java Programmers." Science of Computer Programming **53**(1): 3-24.

Gomoll, K. Some Techniques for Observing Users, Advanced Technology Group, Apple Computer.

Goodall, S. (1991). Online Help in the Real World. International Conference on Design of Communication, ACM Press.

Goodall, S. (1992). Online Help: A Part of Documentation. International Conference on Design of Communication, ACM Press.

Goodman, D. (1987). The Complete Hypercard Handbook. Birmingham, AL, Bamtam Computer Books.

Guzdial, M. (1995). "Software-Realized Scaffolding to Facilitate Programming for Science Learning." Interactive Learning Environments **4**(1): 1-44.

Hancock, C. (2001). Children's Understanding of Process in the Construction of Robot Behaviors. Symposium on Varieties of Programming Experiences, Seattle, WA.

Harrison, S. (1995). A Comparison of Still, Animated, or Nonillustrated On-Line Help with Written or Spoken Instructions in a Graphical User Interface. Conference on Human Factors in Computing Systems.

Hartmann, W., J. Nievergelt, et al. (2001). Kara: finite state machines, and the case for programming as part of general education. IEEE Symposia on Human Centric Computing Languages and Environments, Stresa, Italy.

Hays, J. and M. Burnett (2001). Guided Tour of Forms/3. **2004**.

Hecker, D. (2005). "Occupational employment projections to 2014." Monthly Labor Review(November 2005).

Hintze, J. and M. Masuch (2004). Designing a 3D Authoring Tool for Children. Second International Conference on Creating, Connecting and Collaborating through Computing, Kyoto, Japan.

Holt, R. and J. Cordy (1988). "The Turing Programming Language." Communications of the ACM **31**(12): 1410-1423.

Holt, R., D. Wortman, et al. (1977). "SP/k: A System for Teaching Computer Programming." Communications of the ACM **20**(5): 301-309.

Hudson, S., R. Rodenstein, et al. (1997). Debugging Lenses: A New Class of Transparent Tools for User Interface Debugging. Symposium on User Interface Software and Technology, ACM Press.

Igarashi, T., S. Matsuoka, et al. (1999). Teddy: a sketching interface for 3D freeform design. International Conference on Computer Graphics and Interactive Techniques, ACM Press.

Ingalls, D., S. Wallace, et al. (1988). Fabrik: A Visual Programming Environment. Object Oriented Programming Systems, Languages, and Applications, San Diego, CA.

Jackson, J., J. Krajcik, et al. (1998). The Design of Guided Learner-Adaptable Scaffolding in Interactive Learning Environments. Conference on Human Factors in Computing Systems.

Joshi, P., W. Tien, et al. (2003). Learning Controls for Blend Shape Based Realistic Facial Animation. SIGGRAPH/Eurographics Symposium on Computer Animation, San Diego, CA, ACM Press.

Kafai, Y. (1995). Minds in Play. Hillsdale, NJ, Lawrence Erlbaum Associates.

Kahn, K. (1996). "Drawings on napkins, video-game animation, and other ways to program computers." Communications of the ACM **43**(3): 104-106.

Kato, H. and A. Ide (1995). Using a Game for Social Setting in a Learning Environment: AlgoArena -- A Tool for Learning Software Design. Computer Supported Collaborative Learning, Bloomington, Indiana.

Kay, A. Etoys and Simstories in Squeak.

Kay, A. (1993). "The Early History of Smalltalk." ACM SIGPLAN Notices **28**(3): 69-96.

Kelleher, C. and R. Pausch (2005). "Lowering the Barriers to Programming: a survey of programming environments and languages for novice programmers." ACM Computing Surveys **37**(2): 83-137.

Kimura, T., J. Choi, et al. (1990). Show and Tell: A Visual Programming Language. Visual Programming Environments: Paradigms and Systems. E. P. Glinert, IEEE Computer Science Press**:** 397-404.

Kloper, E. and A. Begel (2006). StarLogo TNG.

Klopfer, E. and S. Yoon (2005). "Developing Games and Simulations for Today and Tomorrow's Tech Savvy Youth." Tech Trends **49**(3): 33-41.

Knabe, K. (1995). Apple Guide: A Case Study in User-Aided Design of Online Help. Conference on Human Factors in Computing Systems, ACM Press.

Kolling, M., B. Quig, et al. (2003). "The BlueJ system and its pedagogy." Journal of Computer Science Education, Special Issue of Learning and Teaching Object Technology **12**(4): 249-268.

Kolling, M. and J. Rosenberg (1996). Blue - A language for teaching object-oriented programming. Proceedings of the twenty-seventh SIGCSE technical symposium on Computer Science Education, Philadelphia, PA.

Kolling, M. and J. Rosenberg (1996). An Object-Oriented Program Development Environment for the First Programming Course. Proceedings of the twenty-seventh SIGCSE technical symposium on Computer Science Education, Philadelphia, PA.

Kramer, A. (1994). Translucent Patches. Symposium on User Interface Software and Technology, ACM Press.

Kurtz, T. (1981). BASIC. History of Programming Languages. R. Wexelblat. New York, Academic Press**:** 515-537.

Laurel, B. (2001). Utopian Entrepreneur. Cambridge, MA, MIT Press.

Liebermann, H. (1993). Mondrian: A Teachable Graphical Editor. Watch What I Do: Programming by Demonstration. A. Cypher. Cambridge, MA, MIT Press.

Lionet, F. and Y. Lamoureux (1994). Klik and Play, Maxis.

Looser, J., M. Billinghurst, et al. (2004). Through the Looking Glass: The Use of Lenses as an Interface Tool for Augmented Reality Interfaces. Conference on Computer Graphics and Interactive Techniques, ACM Press.

Maloney, J., L. Burd, et al. (2005). Scratch: A Sneak Preview. International Conference on Creating, Connecting, and Collaborating through Computing., Kyoto, Japan.

Marcia, J. (1980). Identity in Adolescence. Handbook of Adolescent Psychology. J. Adelson. New York, NY, Wiley - Interscience.

Margolis, J. and A. Fisher (2002). Unlocking the Clubhouse: Women in Computing. Cambridge, MA, MIT Press.

Martin, F., G. L. Colobong, et al. (1999). Tangible Programming with Trains.

McIver, L. (1999). Grail: A Zeroth Programming Language. Conference in Computers in Education.

McIver, L. (2001). Syntactic and Semantic Issues in Introductory Programming Education. Computer Science and Software Engineering. Melbourne, Australia, Monash University.

McNerney, T. (2000). Tangible Programming Bricks: An Approach to Making Programming Accessible to Everyone. MIT Media Lab. Cambridge, MA.

Merrill, D. C. and B. J. Reiser (1993). Scaffolding the acquisition of complex skills with reasoning-congruent learning environments. Workshop in Graphical Representations, Reasoning, and Communication from the World Conference on Artificial Intelligence in Education, University of Edinburgh.

Microsoft Windows Family.

Miller, P., J. Pane, et al. (1994). "Evolution of Novice Programming Environments: The Structure Editors of Carnegie Mellon University." Interactive Learning Environments **4**(2): 140-158.

Minsky, M. (1986). The Society of Mind. New York, NY, Simon and Schuster.

Montemayor, J., A. Druin, et al. (2002). Physical Programming: designing tools for children to create physical interactive environments. Conference on Human Factors in Computing Systems, Minneapolis, MN.

Moskal, B., D. Lurie, et al. (2004). Evaluating the Effectiveness of a New Instructional Approach. Technical Symposium on Computer Science Education, ACM Press.

Motil, J. and D. Epstein (1998). JJ: A Language Designed for Beginners (Less is More).

Mulholland, P. and S. Watt (1998). Hank: A Friendly Cognitive Modelling Language for Psychology Students. IEEE Symposium on Visual Languages, Nova Scotia.

NCLB (2002). No Child Left Behind.

Nelson, M. (2001). Robocode, IBM Advanced Technologies.

Newburger, E. C. (2000). Home Computers and Internet Use in the United States: August 2000, U.S. Census Bureau.

Nielson, J. (1993). Usability Engineering. Boston, MA, Academic Press.

Norman, D. (1986). Cognitive Engineering. User Centered System Design: New Perspectives on Human-Computer Interaction. D. Norman and S. Draper. Hillsdale, NJ, Lawrence Erlbaum Associates.

Overmars, M. Drape: Drawing Programming Environment.

Palmiter, S., J. Elkerton, et al. (1991). "Animated demonstrations vs. written instructions for learning procedural tasks: A preliminary investigation." International Journal of Man-Machine Studies **34**: 687-701.

Palmiter, S. and L. Elkerton (1991). An Evaluation of Animated Demonstrations for Learning Computer-based Tasks. Conference on Human Factors in Computing Systems, ACM Press.

Pane, J. (2002). A Programming System for Children that is Designed for Usability. Computer Science. Pittsburgh, PA, Carnegie Mellon University.

Pane, J., B. Myers, et al. (2002). Using HCI Techniques to Design a More Usable Programming System. HCC, IEEE Press.

Papert, S. (1980). Mindstorms: Children, Computers, and Powerful Ideas. New York, Basic Books.

Patterson, D. (2005). "Stop Whining About Outsourcing!" ACM Queue **3**(9).

Pattis, R. (1981). Karel the Robot: A Gentle Introduction to the Art of Programming with Pascal. New York, Wiley and Sons.

Pausch, R. Building Virtual Worlds.

Perlman, R. (1976). Using Computer Technology to Provide a Creative Learning Environment for Preschool Children. Electrical Engineering and Computer Science. Boston, MA, MIT.

Quintana, C., J. Eng, et al. (1999). Symphony: a case study in extending learner-centered design through process space analysis. Conference on Human Factors in Computing Systems, Pittsburgh, PA, ACM Press.

Repenning, A. (1993). Agentsheets: a tool for building domain-oriented visual programming. Conference on Human Factors in Computing Systems.

Repenning, A. and J. Ambach (1996). Tactile Programming: A Unified Manipulation Paradigm Supporting Program Comprehension, Composition, and Sharing. IEEE Symposium on Visual Languages, Boulder, CO.

Reps, T. and T. Teitelbaum (1989). The Synthesizer Generator: A System for Constructing Language-Based Editors. New York, NY, Springer-Verlang.

Resnick, M. (1996). StarLogo: An Environment for Decentralized Modeling and Decentralized Thinking. Human Factors in Computing Systems, Vancouver, BC.

Reynolds, C. (1987). "Flocks, Herds, and Schools: A Distributed Behavioral Model." Computer Graphics **21**(4): 25-34.

Robinett, W. (1979). Atari 2600 Basic Cartridge, Atari Company.

Robinett, W. and L. Grimm (1982). Rocky's Boots/Robot Odyssey, The Learning Company.

Roper (1999). The America Online/Roper Starch Youth Cyberstudy.

Sammet, J. (1981). The Early History of Cobol. History of Programming Languages. R. Wexelblat. New York, Academic Press**:** 199-241.

Scaffidi, C., M. Shaw, et al. (2005). Estimating the Numbers of End Users and End User Programmers. IEEE Symposium on Visual Languages and Human-Centric Computing, IEEE.

Schneiderman, B. (1983). "Direct manipulation: A step beyond programming languages." IEEE Computer **16**(8): 57-69.

Selker, T., R. Barber, et al. (1996). Effective, Selective Presentation of Help Material in a Graphical Environment: Experience with COACH/2, a graphical adaptive help system., IBM.

Sellman, R. (1992). Gravitas: An Object-Oriented Discovery Learning Environment for Newtonian Gravitation. Proceedings of East-West Conference on Human-Computer Interaction.

Shashaani, L. (1994). "Gender-Differences in Computer Experience and its Influence on Computer Attitudes." Journal of Educational Computing Research **11**(4): 347-367.

Sherwood, B. and J. Sherwood (1988). The cT Language. Champaigne, IL, Stipes Publishing Company.

Skills, P. f. s. C. (2006). Results that Matter: 21st century skills and high school reform.

Smith, D. (1993). Pygmalion. Watch What I Do: Programming by Demonstration. A. Cypher. Cambridge, MA, MIT Press.

Smith, D., A. Cypher, et al. (1994). "KidSim Programming Agents without a Programming Language." Communications of the ACM **37**(7): 54-67.

Smith, D., A. Cypher, et al. (2000). "Programming by example: novice programming comes of age." Communications of the ACM **43**(3): 75-81.

Smith, J. L. (2006). Living Dolls. Telegraph, Telegraph Media Group.

Smith, R. (1987). Experiences with the Alternate Reality Kit: An Example of the Tension Between Literalism and Magic. Human Factors in Computing Systems.

Stone, L. J. and J. Church (1984). Childhood and Adolescence: A Psychology of the Growing Person, 5th Edition. New York, Random House.

Sukaviriya, P., E. Isaacs, et al. (1992). Multimedia Help: A Prototype and an Experiment. Extended Abstracts of Conference on Human Factors in Computing Systems.

Sun (2006). Trail: Creating a GUI with JFC/Swing.

Suzuki, H. and H. Kato (1995). Interaction-Level Support for Collaborative Learning: AlgoBlock -- An Open Programming Language. Computer Supported Collaborative Learning, Bloomington, IN.

Tanimoto, S. and M. Runyan (1986). Play: An Iconic Programming System for Children. Visual Languages. S. K. Chang, T. Ichikawa and P. A. Ligomenides, Plenum Publishing Corporation**:** 191-205.

Teitelbaum, T. and T. Reps (1981). "The Cornell Program Synthesizer: a syntax-directed programming environment." Communications of the ACM **24**(9): 563-573.

Tomek, I. (1983). The First Book of Josef: an introduction to computer programming. Englewood Cliffs, New Jersey, Prentice Hall.

Travers, M. (1994). Recursive Interfaces for Reactive Objects. Human Factors in Computing Systems, Boston, MA.

Tucker, A., F. Deek, et al. (2002). A Model Curriculum for K-12 Computer Science: Report of the ACM K-12 Education Task Force Computer Science Curriculum Committee, ACM.

Vegso, J. (2005). "Interest in CS as a Major Drops Among Incoming Freshmen." Computing Research News **17**(3).

Viega, J., M. Conway, et al. (1996). 3D Magic Lenses. Symposium on User Interface Software and Technology, ACM Press.

Wallace, R., E. Soloway, et al. (1998). ARTEMIS: learner-centered design of an information seeking environment for K-12 education. Conference on Human Factors in Computing Systems, Los Angeles, California.

Wikipedia Tex Avery.

Wing, J. (2006). "Computational Thinking." Communications of the ACM **49**(3): 33-35.

Wirth, N. (1993). "Recollections about the Development of Pascal." ACM SIGPLAN Notices **28**(3): 333-342.

Wyeth, P. and H. C. Purchase (2000). Programming without a computer: a new interface for children under eight. First Australasian User Interface Conference, Canberra, Australia.