Carnegie Mellon University

Robotics Institute

Thesis

Master's Degree

Title:

Visual Programming Pedagogies and Integrating Current Visual Programming Language Features

Date of Submission: August 19th, 2009

SUBMITTED BY: Erik Pasternak

SUPERVISOR: Professor Illah Nourbakhsh
Robotics Institute

COMMITTEE MEMBERS: Aaron Steinfeld
Robotics Institute

Leigh Ann Sudol
Computer Science

Contents

1 Introduction **4**

1.1 History of VPLs . . . 5

2 Literature Review **5**

2.1 Barriers to Entry . . . 6

2.2 Gulf of Execution . . . 7

2.3 Gulf of Evaluation . . . 8

2.4 Further Discussion . . . 9

3 JubJub **9**

3.1 Researcher Interviews . . . 10

3.2 System Design . . . 11

3.3 User Study . . . 17

3.3.1 Methods . . . 18

3.3.2 Results and Discussion . . . 20

3.4 Future Work . . . 22

4 Conclusion **22**

5 Acknowledgements **23**

A User Interview Questions **27**

B Concept Drawings **28**

Abstract

This Master's Thesis presents research on pedagogic visual programming languages for computer science education. In it we review previous work, discuss current concepts and tools, and propose further work with initial findings for a new visual Java environment called JubJub. JubJub was created as part of this research to support customizable sets of visual bocksl in order to easily interface with diverse sets of educational material. JubJub was inspired by many features of current pedagogic and visual languages, as well as interviews with experts in the field of CS education and visual programming languages. A prototype of JubJub was created and used in a pilot test, the results of which will direct future development of JubJub.

1 Introduction

The US Bureau of Labor predicts that by 2016 there will be over 1.5 million computer specialist jobs available in the US. The number of people with a bachelor's in Computer Science from a US university is only expected to account for 53% of the needed workforce [1]. Combined with the general decline in CS enrollment many colleges are facing [2], it is clear that improvements to CS curriculum are needed. Pedagogic tools, languages, and environments have been developed to make programming easier for novices and younger students to start learning and to find new ways of motivating and inspiring students. Languages like Scratch [3] and Turtle Art [4] are designed to appeal to children as young as 8, while IDEs such as BlueJ [5] and DrJava [6] provide increased support for novice programmers. One part of the solution is innovative visual programming languages (VPLs) designed to motivate students at all age groups and support the teaching of programming and CS concepts.

Visual programming languages are intended to provide metaphores for programmers. These metaphores often relate real world activities, such as snapping blocks together [7, 3, 4], creating a music patch [8], or building circuit diagrams [9], with existing programming paradigms. Through these metaphores the user becomes able to create an affect with minimal training and more easily grasp programming concepts. VPLs help to create a more natural programming environment that users find familiar. In Computer Science education, where the goal is to produce programming and software experts, VPLs have several goals beyond letting a user write programs:

1. Use metaphores that are understood by a broad audience.
2. Reduce the cognitive load on students learning their first language.
3. Create code that is easily read and understood.
4. Be usable by teachers.
5. Support current curriculum and make integration easy.
6. Support students transitioning into commercial programming languages.
7. Be fun.

All of these goals cannot be met for everyone at once, and so many different tools have been developed which focus on different groups and age ranges. Research on VPLs is still relatively new, and while there are some great tools already available, there is still a lot of work to be done. To help, this paper presents research on the current state of VPLs, how they can help with some of the problems that still persist in CS education, and presents work on a new VPL tool called JubJub to provide further discussion of the features necessary for pedagogic VPLs to be effective. JubJub is a visual environment being developed to allow the creation of custom sets of code blocks to easily support a diverse set of materials and curricula. The main goal of this research has been to identify the features and requirements important to pedagogic VPLs.

1.1 History of VPLs

W. Sutherland is credited with having created the first interactive VPL in 1966 [10]. Sutherland's system resembled a flow diagram with symbols for operations and interactions. VPLs continued to be developed during the 70's and 80's as a way to allow scientists and computer enthusiasts access to programming, but due to poor graphics capabilities at the time such languages had a difficult time establishing themselves. Prograph, a box and line style VPL, remains one of the few exceptions, with their first release in 1988 for the Mac. Prograph remained in active development until the mid 90's but has since given way to other languages. IDEs such as Visual Basic created a hybrid environment which combined visual and textual elements [10]. Interest in VPLs dramatically increased during the 90's and today numerous popular VPLs can be found[3, 9, 11, 8, 12, 13].

2 Literature Review

The research important to pedagogic VPLs is wide ranging with roots in many fields. This section will focus on a small portion of this research with two main goals in mind, (i) to discuss specific difficulties in CS education which motivate the development of VPLs and (ii) to discuss features of current VPLs and how they seek to overcome these difficulties. To this end, we will look at a subsection of the research concerning novice programmers and the difficulties they face, as well as research on the pedagogic tools that have been developed. When looking at tools we will mostly focus on VPLs and features of other systems which can readily integrate with a visual environment.

This is not intended to be a complete discussion of the field or of all the tools available, but instead to provide examples and support for important concepts and features of pedagogic VPLs.

Before a student faces any challenges learning to program they must choose to learn programming. For many students, this involves overcoming barriers to entry which can take many forms for many people. We will first look at some of these barriers and research on how they may be overcome. Once we examine these barriers, we will make use of Norman's Gulf of Execution, which is the distance between what a user intends to do and the actions a system presents, and Gulf of Evaluation, which is how well a system makes available and understandable its state to the user for evaluation [14], to scaffold our discussion of the difficulties students face.

2.1 Barriers to Entry

Interest in computer science has been declining significantly since 2000 with a 39% decrease between the fall of 2000 and 2004 [2]. Women and minorities have had the steepest decline, with a 79% decrease in interest in CS among women between 2000 and 2008 [1]. This is on top of an already existing underrepresentation of these groups in CS [1]. Women and minorities face many extra barriers to entry, including external discouragement and a lack of access to the tools and communities necessary to learn [15, 16, 17, 18]. While dramatic changes are needed to remove these extra barriers, other steps can be taken to help. Maloney describes a program in which researchers introduced Scratch into urban youth centers in low income areas to great affect [19]. Within two years of supporting the Scratch IDE researchers found Scratch had become the most heavily used media creation tool at the centers, even over programs like MS Word. The key to Scratch's success was the explorative model of learning they used; Scratch was supported in the centers, but no classes or assignments were provided [19]. In interviews, youth who used Scratch did not associate scripting in Scratch with programming, yet they used a wide variety of programming concepts and structures [19]. The authors of the study believe not associating Scratch with programming may have helped youth see it as something fun and cool [19]. Maloney, in [19], also found Scratch was used by an even mix of boys and girls, showing a general appeal for the style of learning. This increase in women's interest has been shown with the use of storytelling and creating worlds with Alice as well [20].

There are also barriers to entry that affect all students. Negative stereotypes regarding computer experts and misconceptions about the skills needed in CS are common among high school students [18]. Many programs fail to excite students, and use lesson plans that are "disengaging and isolated from the problem solving and scientific reasoning that is at the core of computer science" [16]. VPLs empower students and teachers alike by providing access to powerful, real world applications such as graphics [13, 4], GUIs [12], gaming [11] and robotics [21, 7, 22]. These motivators have been shown to be effective with standard languages as well [23, 24], though VPLs allow much younger students to be reached.

2.2 Gulf of Execution

Once a student has decided to learn programming they are faced with many challenges. Students are asked to master a number of skills in parallel, including syntax and grammar of the language itself, problem solving and logic, and programming concepts such as object orientation and recursion [25]. Students must learn how to plan in terms of programming concepts, and also how to execute those intentions in a given environment and language. Faced with this challenge, it is no surprise [26] found very few students were able to write programs to accomplish simple tasks after one or two programming courses, with students from four universities achieving an average score of 22.9/110 on the study's assessment problems. This is where VPLs provide the greatest benefit; their use of metaphores, simplification of the language, and visible constraints helps guide students, changing the gulf of execution from a yawning chasm to a pothole and allowing them to focus on the paradigms important to programming.

Students face many specific problems with execution, especially with syntax, selection, and use. While most educators agree the goal in intro CS courses is to teach students programming concepts, not a specific language [25], studies have found students struggle with syntax far more than we think, with students spending up to 30 minutes on a single 'missing ;' error [27, 28, 29]. The five most common compiler errors – missing semicolon, unkown variable, illegal start of expression, unkown class, and bracket expected – have been shown to account for over half of all compiler errors among novice programmers [27]. [27, 28] also found a correlation between the students who struggled with syntax errors and their grades in the course. Others have suggested this leads to students focusing on writing code that compiles without understanding its function [26]. Many

VPLs eliminate the majority of syntax errors by creating visual cues and restrictions on how pieces can fit together. IDEs like Scratch and Alice only allow valid blocks to be placed [3, 13]. Scratch also uses shapes as visual cues to help students place code [3]. Students have difficulties with selecting and using appropriate commands as well; in a study on the barriers students face, half of the selection and use barriers encountered by students were insurmountable [30]. Pedagogic IDEs and VPLs both use reduced language sets to make finding an appropriate command easier for students, though this approach often fails to scale to more complex tasks [31]. VPLs also use visual hints to help students decide what commands will do and how they are used [9, 3].

2.3 Gulf of Evaluation

Students' also face problems with evaluating their results, due to conceptual misunderstandings as well as poor support for tracking the state of a system. [30] describes these as understanding barriers, when students must use system behavior (including compile and runtime errors) to determine what a system did and did not do, and information barriers, when students attempt to observe the internal behavior of a system to test their understanding. The majority of understanding and information barriers encountered by students were insurmountable (34/38 and 10/14 respectively) [30]. VPLs have explored several features to help students overcome these barriers. Data flow VPLs, like LabView, Robolab, and Pure Data, provide another way for students to visualize how information is passed [8, 12, 9]. Labview also provides visualization tools to animate program flow to make it easier for students to understand what values are being passed and where errors are occuring [9, 12]. Pure Data lets users run code while it is being edited, making it possible to observe the results of changes in real time [8]. During selection and use students must also evaluate the results and determine if their understanding of commands is correct. One benefit of many pedagogic languages and IDEs is that they provide what the creators of DrJava term a "read-eval-print-loop," or REPL [32]. REPL is a method by which students can execute small portions of code or individual commands and observe the output, making it much easier for them to discover differences between expected and actual behavior of commands [32]. DrJava supports REPL by allowing students to enter individual commands which the system evaluates and displays the results of [6]. Similarly, Scratch allows students to double click any block or group of blocks to run them [3] and BlueJ provides an interface for instantiating objects and calling methods from those objects [5]. It has been shown

that the REPL method encourages exploration and incremental development among students [32].

2.4 Further Discussion

It has been shown throughout the literature that VPLs can support students in various ways, especially by reducing cognitive barriers and empowering students to create relevant projects. Many features, such as animating program flow and providing access to the internal state of code, also help students evaluate their programs. However, in CS programs where the goal is to produce programmers and software experts it is also important to support students transferring into commercial languages. Scratch and Alice have both had success motivating these transitions [33, 22]. However, it is unclear that these are the best tools to support such a transition. [34] comments, "We can now more easily introduce beginners to programming; perhaps it is time to begin studying the intermediate programmer, someone who has been introduced to programming through a system designed for beginners and wants to apply that experience to learning a general language." The same observations led researchers at Hong Kong Polytechnic University to design a visual language which generates code that is displayed next to the visual code for the Arduino [35]. Other work has attempted to bridge the divide between VPLs and textual languages as well, including [36], which lets students program with sequential sets of blocks, in Logo, or by directly clicking and dragging the famous turtle around its window. Both of these systems remain experimental.

3 JubJub

The first goal in the design of JubJub is to create an architecture for a customizable and expandable iconic programming interface. The reason for making this the main goal is to support integration with current curricula, software, and educational materials. The second goal in designing JubJub is to identify and integrate a comprehensive set of features useful to pedagogic VPLs with the specific aim of supporting junior high and high school students who are interested in programming. The current field of VPLs supports many unique and powerful features that are of great benefit to novice programmers, and while some features are mutually exclusive, there are many lessons to be learned from each system. Throughout the design description for JubJub, we will include what we feel are the most significant contributions and features of current tools, from which JubJub has

borrowed heavily. JubJub is not a definitive set of the best features for pedagogic VPLs and many concessions had to be made due to an accelerated development schedule, however, we feel it is a solid base for achieving our goals. Initial requirement and feature identification was guided by a series of interviews with researchers and educators in the field of pedagogic and visual languages. Further development looked to research and informal feedback from students and educators for help. Graphics and images for JubJub were created by Cheng Xu, an Interaction Design major at CMU, while I designed and coded the system. The prototype system was demonstrated to high school CS teachers and changes guided by their feedback were adopted. The modified system was then used for an initial pilot with students to identify any critical problems and inform future development.

3.1 Researcher Interviews

Nine researchers and educators were interviewed during the initial design of JubJub, including members of the Scratch, Alice, RoboLab, and BlueJ teams. The interviews used an open format in an informal setting and focused on the interviewees experience with students, the problems students face, and their opinions and advice about visual languages. All of the people interviewed provided excellent insight and advice, which will be summarized for discussion in this section.

Many of the problems discussed in the literature were reiterated anecdotally during the interviews. The cognitive load associated with standard environments and languages was seen as a large problem in teaching new students. One educator explained it as, “Java is an 18-wheeler,” while another described the ’gotchas’ of languages as the hardest part, citing the difference between ’.equals’ and ’==’ in Java as an example. However, when talking about syntax, all but one of the educators who did not work with a VPL glossed over it, saying it was minor but necessary and that they expected students to learn the syntax from examples on their own. In contrast, the visual language researchers listed syntax as one of the problems they were avoiding, with the common belief that “conceptual models of programming are the most important to learn and syntax and typing really just get in the way.” Understanding the state of a system was another challenge described in interviews. Difficulties with debugging, artifacts like overflow, and scope of variables were all issues students were described as having.

The responses about visual languages were more varied, with many educators who used Java IDEs commenting that VPLs obscured the details, making it too easy for students to build something without understanding how it works. Professors who used text based IDEs were also uncertain if features like error highlighting during code writing were good for students, with some pointing out that they'd like students to complete a thought first and then correct the grammar and syntax. Even among VPL researchers, there was debate whether a language should rely more on text or icons. All of the VPL researchers did feel that they did not have 'the' solution, or that there was a single solution, and were encouraging of the successes of other languages. Several common themes regarding programming environments were described during interviews. Environments which encourage secondary notation were beneficial to students, with features like auto-indentation and brace matching being the most often mentioned. Having students create code that does 'cool things' was also important, and most professors provided extensive frameworks to make projects like Tetris possible. A sense of collaboration and sharing were also emphasized by researchers, especially among the Scratch team who encourage students to build off of each others' work. One other point of emphasis was that students would grasp a new system faster than teachers will, and that this should be kept in mind when designing a VPL.

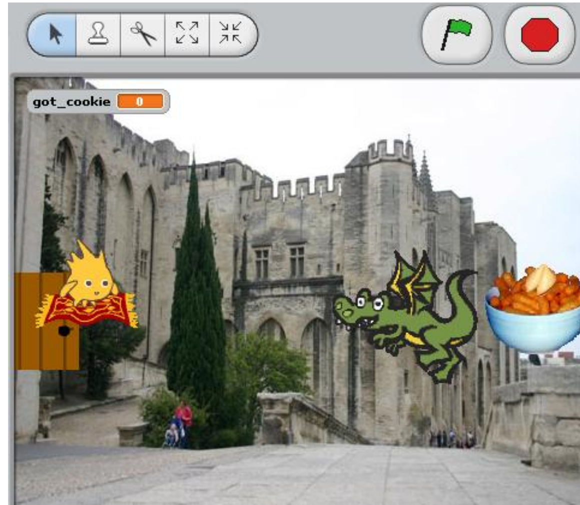
3.2 System Design

Figure 1: Example code from Scratch

To guide the design of JubJub, the research and interviews discussed earlier were used as a scaffold to explore other systems. Several different approaches and pedagogies were found and each system's pros and cons were considered. Throughout this review of other languages, it became increasingly apparent there is a lack of material to bridge a student's first programming experience with commercial environments and languages. Three systems were found to be representative of the most successful approaches in VPLs, these are Scratch, Alice, and RoboLab. These systems have been used by multiple universities and programs and have each demonstrated success in certain areas. Scratch provides the best example of a system that reaches out to new audiences and supports a first foray into programming concepts, but part of their success comes from the restricted domain

they use. Scratch's simplicity and a carefully chosen set of blocks supports their core concepts, but

Figure 2: Scratch's stage with a background and several sprites



restricts what users can achieve and necessitates switching to another environment for more complex programs. A snippet of example code from Scratch can be seen in Figure 1. Code in Scratch affects sprites in a stage area, seen in Figure 2, which students can use to tell stories or make simple games. Alice provides a larger set of concepts and behaviors and even allows some discussion of

Figure 3: Screenshot of the Alice code area

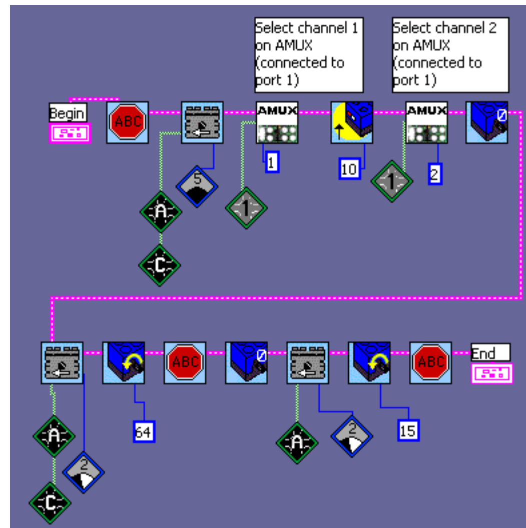
objects, but at the cost of many of the visual hints that makes Scratch so simple for novices to use without direction (Figure 3). And Alice still uses a restricted domain of programming, focusing on manipulating 3D scenes, called microworlds, for storytelling and games. An example microworld

can be found in Figure 4. Expanding Alice is possible but difficult, something which the creators

Figure 4: An Alice microworld

acknowledge and are working to support more explicitly in Alice 3.0. RoboLab takes visual languages a step further, using a purely iconic language, where functions are represented by an image and use as little text as possible (Figure 5). RoboLab comes in several flavors, including ones for Lego Mindstorms and as an extension to LabView, the commercial product RoboLab is based on. RoboLab supports a large amount of expandability and customization of blocks and allows ad-

Figure 5: Example code from RoboLab for Lego Mindstorms(c)



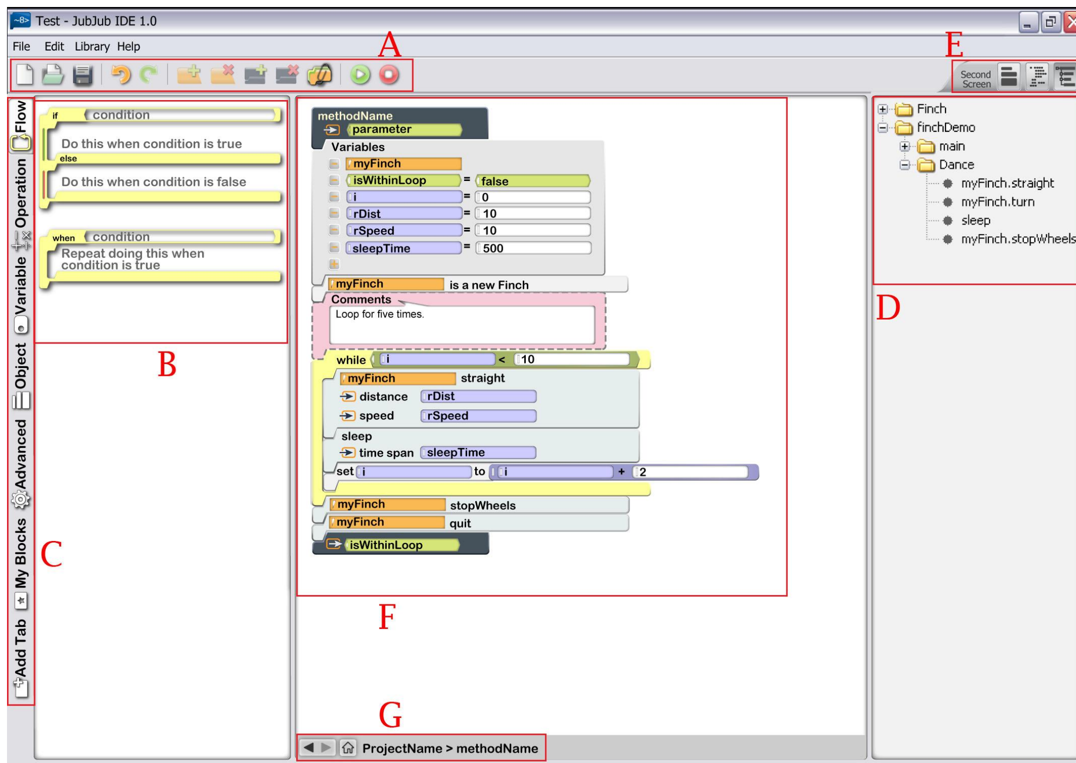
vanced users to write code to create new blocks as well. Because RoboLab is based off of LabView it is easy for students to transition to the commercial language LabView. However, RoboLab has several problems as a pedagogic language (addressed more thoroughly in [37]). It tends towards large sets of blocks, making selection a difficult process for novices. Students also face problems of organizing their code and readability, making complex programs much more difficult to conceptualize and debug. And while LabView is very useful in controls and data processing, especially among engineers without formal programming experience, it does not have the ubiquity of languages like Java and C. Finally, RoboLab is primarily a commercial product, coming as a bundle with Lego Mindstorms or as an add-on to LabView, making access for students more costly. JubJub seeks to combine many of the features that have been shown to be successful in other languages and act as a bridge to commercial IDEs. In order to have the greatest impact, JubJub was targeted for 6th through 12th graders and is intended as a transitional language.

The observations from other systems and interviews led to a set of primary goals which the design of JubJub focuses on:

1. Explicitly support the creation of language subsets.
2. Make building examples and lessons which focus on single concepts easy.
3. Support integration with varied materials, especially robotics and existing libraries of code.
4. Build a framework that can be scaled to more complex projects.
5. Make it easy to share developed material.
6. Help students transition to a popular commercial programming language.
7. Maintain the level of simplicity and usability that makes VPLs a powerful tool.

The first four goals required extensive architectural considerations. The architecture described here has been implemented and was used for the prototype, though not all features could be fully implemented by the prototype interface. In order to provide the level of customization desired, a modular approach was taken. Methods, functions, and objects were conceptually treated as black boxes with inputs and outputs and properties that defined how they were used and what they did, while abstracting the code underneath. The framework developed is general enough to support

Figure 6: Annotated concept for the standard view of JubJub



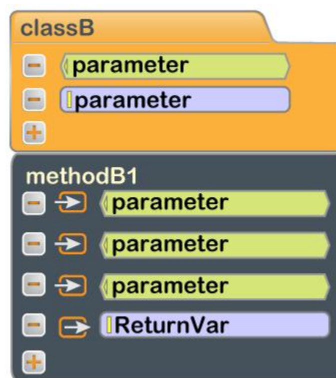
object use, common flow control, including loops and decision making, and supports scaling for more complex structures like switch statements and multiple returns, which makes the architecture expandable to languages other than Java and C. Attributes to track visibility and relation between blocks were added to support identifying scope and tracking the location of errors. To support recombining of blocks into new functionality all blocks were used through references. This mirrors the structure of Java and C languages by allowing a single block of code to be called with different parameters from many places throughout the code. Blocks were designed to maintain a list of these references so that changes to blocks could immediately update all references and check for errors that may have been introduced. A straightforward API was developed to make setting up the properties of these blocks simple. Finally, the structure that stores these blocks was built around XML concepts of heirarchy, making it easy to store project data in a single file that can be shared. This structure continues up to the project and library level allowing block sets for lesson plans to be shared in the same way that students would share their own code.

The last three items are primarily features of the interface, which was given equal consideration in JubJub's design. The following describes the core features and behavior of the desired interface, though only a simplified version was implemented for user testing. Most of the features described appear in Figure 6. Features described will provide a reference to this figure and the letter annotation they reference. After many iterations of concepts, a visual metaphore very similar to Scratch was created (Figure 6: F). Blocks were shaped to visually snap together and both color and shape were used to indicate how blocks should be put together. A palette on the left divides the blocks into high level categories using tabs (Figure 6: C), which are then subdivided into specific categories with any block specific help displayed in the palette (Figure 6: B). The interface supports the creation of new tabs and subgroups which the user can specify and rearrange to fit their own thought process. The palette also uses dynamic updating to provide access to variables, objects and methods as they are created and shading and messages to indicate when variables are in scope. This allows students to see when things are visible in their code and build relations between methods, classes, and variables. On the right is a call list (Figure 6: D), which provides quick navigation to methods and classes and helps students to visualize the structure of their code. Buttons along the top provide access to creating new projects, classes, and blocks, as well as a straightforward run/stop to make it easy for students to test their code often (Figure 6: A). The center work frame allows students to manipulate code by a drag-and-drop interface and uses highlighting to indicate placement (Figure 6: F). Many errors are prevented by only allowing legal blocks to be placed. Some blocks do allow users to type in values directly, such as expression blocks, and other errors may be created by changing the properties of a method or class, so the interface would identify errors on-write and use highlighting to indicate any errors both in the work space and the call tree. While the shapes of blocks are similar to Scratch, more details are included, such as variable types and parameter names. These details are generally represented by both a word and a symbol, so integers use the word 'int' and a rounded, verticle bar as a sybmol (Figure 7). Booleans, numbers, and objects are also differentiated by color and shape, making it clear when each is appropriate as a parameter. The work area also supports temporal navigation, with forward and back buttons and a visitation stack displayed at the bottom so students can quickly jump between sections of code (Figure 6: G). Finally, a button

Figure 7: An integer block in JubJub. A verticle bar at the left is the symbol for int

allows the work area to be divided into two sections in order to display a second method or the generated source code (Figure 6: E). This allows students to compare sections of their code to each other and to a textual language, like Java. Students' code can be written out to either an XML file or used to generate language specific text files that students can use as a starting point for more advanced editing. A similar view would be used to display classes for more advanced projects, listing the attributes and methods within a class (Figure 8). More advanced attributes, like Java's 'private' keyword would be accessible through a right-click menu. Finally, blocks would contain

Figure 8: A JubJub class icon. Attributes and methods are listed under the class name.



information to allow them to be run and evaluated individually, as in Scratch or DrJava. The full set of concept drawings can be found in Appendix B. The design of an interface for sharing code and block libraries has not yet been given extensive consideration, though Scratch's web interface has been very successful and RoboLab provides a robust interface for documenting code users have written by putting runnable snippets into a document with text and embedded media. Future work on code sharing would hope to build on these two systems. Many details of JubJub were inspired by current VPLs. From Scratch, JubJub uses many of the natural language and shape metaphores that are so successful with younger students. Alice 2.0 and the beta for Alice 3.0 introduced several new features as well that were used in JubJub, such as using small symbols to represent variable types on variable blocks and color coding code output to match the visual language.

3.3 User Study

To test our system we conducted a pilot study using six students and a prototype system. The prototype used only a small subset of the designed features. We chose the features to focus on four of our original seven goals:

1. Support integration with varied materials, especially robotics and existing libraries of code.
2. Build a framework that can be scaled to more complex projects.
3. Help students transition to a popular commercial programming language.
4. Maintain the level of simplicity and usability that makes VPLs a powerful tool.

The prototype system, shown in Figure 9, only allows editing of a single method and does not support adding or removing variables. The textual code on the right side of the prototype updated in real time, with the output generated by the code users were writing. Code was assembled by dragging and dropping blocks into the center area. A blue line would indicate where code blocks would be placed on release and a black box outlined where variables and expressions would be inserted. The prototype contains only two tabs with a small subset of blocks chosen to support basic Java functionality, including while and if/else constructs, and a set of blocks for interaction with the Finch, a usb tethered robot programmed in Java and developed for CS education by the CREATE Lab at Carnegie Mellon.

3.3.1 Methods

The six users were located through social connections and recruitment fliers distributed on the Carnegie Mellon campus. Our users included three men and three women. Two of the men were non-native English speakers. They came from three majors: ECE, Design, and Math. All of our users had taken two or fewer college level classes. All of them had used more than one programming language, though only two of them programmed as part of their regular coursework. None of our users had used a VPL before. We met with users individually for about an hour and our interactions were audio recorded for review later. The first 20 minutes consisted of an interview in which we asked users about their programming background, experiences learning a

Figure 9: Prototype built to test the design of JubJub

first programming language, and IDEs they had used. The interviews used a set of questions as a guide, which can be found in Appendix A. After the interview, users were shown the Finch and the prototype system. They were given a quick overview of the features of the Finch and a roughly two minute demo of how to use JubJub. Users were presented with an example program that can be seen in Figure 9. Before being shown the output of the program, users were asked to look through the code and describe what they thought the code would do. After they had given their explanation the code was run on the Finch and anything about the starting program they had trouble with was further explained. Blocks not in the example code were given no explanation for use. The users were then asked to add code to the default method to make the Finch dance, using at least two unique behaviors. If a user asked a question that had been covered in the intro it was answered, for other questions they were told it would be answered at the end of the exercise. Screen capture software was used to record their interaction with the prototype and used to perform timing and identify features of their use. After the exercise was completed, users were asked a series of follow-up questions about their interaction and they were asked to compare it to other systems they had used.

3.3.2 Results and Discussion

During the interview portion, the users had a mostly neutral opinion of the IDEs thay had used. Several of them stressed the importance of a simple interface and two of our users who had used Processing mentioned liking the run/stop buttons in the Processing interface. Several users also mentioned features like code completion and brace highlighting positively. However, the users still felt that the IDEs did not give them enough or the right kind of information, mentioning cryptic error messages and little or no help debugging faulty code. When asked if the environment helped or hindered learning the language all of our users said it had little affect. The two users who program regularly both said they had been exposed through a parent, while the rest of our users had first learned in college. This further supports research on the benefits of teaching concepts to a younger audience.

During the first portion of the exercise we introduced users to the JubJub interface and asked them to read the visual code and describe what the program would do. All of the users correctly identified that the Finch would drive straight and turn and all but one user said the correct number

of times the behavior would loop, with the one user miscounting and being off by two. A more in depth study comparing the readability of JubJub with other languages is needed to provide definitive evidence, but these early results are very positive. This observation was supported by students during follow-up questions who said it took them a minute to understand the layout, but once they did it was very easy to read. While modifying the code two users made mistakes which took more than a minute to fix and one user introduced a compile-time error, though no errors were unrecoverable. The types of errors encountered were mostly conceptual and it was observed that code flow visualization would have been of benefit for most of the encountered errors. All six users completed the task, with the fastest completing the exercise in 2m40s and the longest in 24m. Three of the users took between 6 and 12 minutes, and the two non-native English speakers took 18 and 24 minutes, which implies a language barrier affected use. The most promising result is that three users asked to continue experimenting after they were told they had completed the task and two of the three were women. This suggests that the system was enjoyable to our users and can motivate learning. Further, five users experimented with a block that had not been explained, which demonstrates the system's explorability. During follow-up questions, students supported these observations verbally. On comparison of the follow-up responses, a clear difference in opinion could be found between our less experienced and more experienced users. The less experienced users made primarily favorable comments about the prototype, saying it was more intuitive and easier for them to use than text languages. The more experienced users also felt it was easy to use but that it was restrictive and that they could type the commands in more quickly. This distinction is similar to observations made by [35]. A transitional language should therefore support editing code within blocks and expressions directly, both to support users learning the textual language and to allow more advanced users to edit code quickly while still receiving the benefits of a visual interface for organizing a project. Finally, most users compared the interface to devices they see in their daily lives, such as an iPhone.

While not statistically significant, these results are encouraging of further work. The ease with which the users learned the interface suggests it could be introduced to schools with minimal support. The positive response the system received, despite being a prototype, shows that students are attracted to the interface and would be likely to explore concepts through JubJub. Furthermore, with minimal support from the interface users were able to match up the Java with the visual code,

demonstrating the close conceptual mapping desired. Since all our users were familiar with at least one textual language this demonstrates a link from textual programming to the visual, but it is our opinion that concepts would be transfered in the other direction as well. Users also associated the interface with interactions in their daily lives, showing it felt relevant to them. The architecture also fully supported the creation of blocks for interfacing with the Finch, and the full set of blocks took only about an hour to assemble using the API.

3.4 Future Work

The next step for JubJub will be to provide further documentation and establish the system as an open source project. As a more complete interface is developed extensive attention will be given to the design of visualization and debugging tools, especially for program flow visualizations and making the value of variables during run-time clearly visible. A first step towards debugging and advanced editing tools will be to integrate JubJub with a current text IDE, such as BlueJ or Eclipse. An interface for creating new sets of blocks and sharing them online will be critical to the success of JubJub and will be pursued in parallel to work on the main interface. In developing the sharing interface other systems will be used for inpsiration, especially RoboLab's RoboBook tutorial system and the Scratch community website. As JubJub is developed further user studies are required to ensure its goals are being met. These studies should be performed in junior and senior high schools as part of a CS class and compared against other systems.

4 Conclusion

Current VPLs have been shown to be very successful in motivating students to explore programming, but they lack a clear transition to commercial environments and languages. They are often restricted to a single domain, making them difficult to adapt to other areas of programming and educators' lesson plans. JubJub demonstrates a set of features that can be used to fill this need and provides a level of flexibility necessary for wide-spread distribution. Initial results for JubJub are very promising and with further development can become a step towards more general pedagogic VPLs. However, there remains a great deal of work before a full system is implemented and tested. JubJub is by no means the best solution and it will continue to be influenced by the successes of

other systems and will hopefully inspire new features in the work of others as well.

5 Acknowledgements

Many people gave their time and skills to make this project successful. Cheng Xu is an interaction design major at CMU and the creator of all the graphics used in JubJub. Without her help JubJub would not have been a success. My adviser, Illah Nourbakhsh, and committee members, Aaron Steinfeld and Leigh Ann Sudol, provided excellent support and advice throughout my work on JubJub. The researchers who found time to meet with me were very encouraging and provided the motivation for developing JubJub as well as insight towards many of the features included in the final design. I am also grateful to our users who found several bugs and made excellent suggestions. Finally, my housemates were very encouraging and gave up several of their evenings to review and provide on my presentation and this document.

The source code for JubJub and updates on current work can be found at http://code.google.com/p/jubjub/

References

[1] "National center for women and information technology," web, 2006.

[2] Jay Vegso, "Interest in cs as a major drops among incoming freshmen," *Computing Research News*, vol. 17, no. 3, 2005.

[3] "Scratch," web.

[4] Sugar Labs, "Turtle art," web.

[5] "Bluej," 2009.

[6] "Drjava," web, 2009.

[7] Lego, "Mindstorms nxt software," 2009.

[8] "Pure data," web, 2009.

[9] "Robolab @ ceeo," .

[10] Jeffrey Vernon Nickerson, *Visual programming*, Ph.D. thesis, New York University, New York, NY, USA, 1995.

[11] "Kodu," .

[12] National Instruments, "Labview," .

[13] "Alice," 2009.

[14] D.A. Norman, *The Design of Everyday Things*, Doubleday, 1988.

[15] Allan Fisher and Jane Margolis, "Unlocking the clubhouse: women in computing," in *SIGCSE '03: Proceedings of the 34th SIGCSE technical symposium on Computer science education*, New York, NY, USA, 2003, p. .23, ACM.

[16] Jane Margolis, *Stuck in the Shallow End: Education, Race, and Computing*, The MIT Press, 2008.

[17] Jonathan Kozol, *Savage Inequalities: Children in America's Schools*, Harper Perennial, 1992.

[18] Uma G. Gupta and Lynne E. Houtz, "High school students' perceptions of information technology skills and careers," *Journal of Industrial Technology*, vol. 16, 2000.

[19] John H. Maloney, Kylie Peppler, Yasmin Kafai, Mitchel Resnick, and Natalie Rusk, "Programming by choice: urban youth learning programming with scratch," in *SIGCSE '08: Proceedings of the 39th SIGCSE technical symposium on Computer science education*, New York, NY, USA, 2008, pp. 367–371, ACM.

[20] Caitlin Kelleher and Randy Pausch, "Using storytelling to motivate programming," *Commun. ACM*, vol. 50, no. 7, pp. 58–64, 2007.

[21] "Mindstorms robolab," .

[22] Briana Lowe Wellman, James Davis, and Monica Anderson, "Alice and robotics in introductory cs courses," in *TAPIA '09: The Fifth Richard Tapia Celebration of Diversity in Computing Conference*, New York, NY, USA, 2009, pp. 98–102, ACM.

[23] Joseph Bergin, Mark Stehlik, Jim Roberts, and Richard Pattis, "Karel j robot," March 2008.

[24] T. Lauwers, I. Nourbakhsh, and E. Hamner, “Csbots: A case study in introducing educational technology to a classroom setting,” Tech. Rep., October 2008.

[25] Tony Jenkins, “On the difficulty of learning to program,” in *Annual Conference of the LTSN Center for Information and Computer Sciences*, 2002, number 3, pp. 53–58.

[26] Michael McCracken, Vicki Almstrum, Danny Diaz, Mark Guzdial, Dianne Hagan, Yifat Ben-David Kolikant, Cary Laxer, Lynda Thomas, Ian Utting, and Tadeusz Wilusz, “A multi-national, multi-institutional study of assessment of programming skills of first-year cs students,” *SIGCSE Bull.*, vol. 33, no. 4, pp. 125–180, 2001.

[27] M.C. Jadud, “A first look at novice compilation behaviour using bluej,” *Computer Science Education*, vol. 15, no. 1, pp. 25–40, 2005.

[28] E.S. Tabanao, M.M.T. Rodrigo, and M.C. Jadud, “Identifying at-risk novice java programmers through the analysis of online protocols,” *Philippine Computing Science Congress*, 2008.

[29] James B. Fenwick, Jr., Cindy Norris, Frank E. Barry, Josh Rountree, Cole J. Spicer, and Scott D. Cheek, “Another look at the behaviors of novice programmers,” in *SIGCSE ’09: Proceedings of the 40th ACM technical symposium on Computer science education*, New York, NY, USA, 2009, pp. 296–300, ACM.

[30] A.J. Ko, B.A. Myers, and H.H. Aung, “Six learning barriers in end-user programming systems,” in *2004 IEEE Symposium on Visual Languages and Human Centric Computing*, 2004, pp. 199–206.

[31] Dieter Vogts, André Calitz, and Jean Greyling, “Comparison of the effects of professional and pedagogical program development environments on novice programmers,” in *SAICSIT ’08: Proceedings of the 2008 annual research conference of the South African Institute of Computer Scientists and Information Technologists on IT research in developing countries*, New York, NY, USA, 2008, pp. 286–095, ACM.

[32] Eric Allen, Robert Cartwright, and Brian Stoler, “Drjava: a lightweight pedagogic environment for java,” in *SIGCSE ’02: Proceedings of the 33rd SIGCSE technical symposium on Computer science education*, New York, NY, USA, 2002, pp. 137–141, ACM.

[33] David J. Malan and Henry H. Leitner, "Scratch for budding computer scientists," in *SIGCSE '07: Proceedings of the 38th SIGCSE technical symposium on Computer science education*, New York, NY, USA, 2007, pp. 223–227, ACM.

[34] Caitlin Kelleher and Randy Pausch, "Lowering the barriers to programming: A taxonomy of programming environments and languages for novice programmers," *ACM Comput. Surv.*, vol. 37, no. 2, pp. 83–137, 2005.

[35] Joey C.Y. Cheung, Grace Ngai, Stephen C.F. Chan, and Winnie W.Y. Lau, "Filling the gap in programming instruction: a text-enhanced graphical programming environment for junior high students," in *SIGCSE '09: Proceedings of the 40th ACM technical symposium on Computer science education*, New York, NY, USA, 2009, pp. 276–280, ACM.

[36] Andy Cockburn and Andrew Bryant, "Leogo: An equal opportunity user interface for programming," *Journal of Visual Languages and Computing*, pp. 601–619, 1997.

[37] T.R.G. Green and M. Petre, "Usability analysis of visual programming environments: A 'cognitive dimensions' framework," *Journal of Visual Languages and Computing*, vol. 7, no. 2, pp. 131–174, 1996.

A User Interview Questions

Background Questions:

What was the first language you learned?

What IDE did you use?

What was the hardest thing to learn when starting a new language? Why?

Which concepts did you have the most trouble learning? Why?

Which concepts were the easiest to learn? Why?

How did the IDE you used help or hinder your learning?

What do you think are the most important programming concepts you've learned?

What are the main language concepts you had to learn (variables, methods, etc.)

After showing JubJub:

Ask to describe what the demo code does.

How easy is it to read and understand the iconic code?

After exercise with JubJub:

What aspects of JubJub were easy to understand?

What are you still unclear about or did you have trouble understanding?

How does JJ compare to other programming environments youve used? What features are missing that youve used before? Are there any features youd like other environments to have?

Did you find having the Java displayed on the right helpful? In what way?

How comfortable did you feel using JubJub? How intuitive was the interface?

Were there any aspects of the interface you didnt like?

If there were an application for creating custom sets of blocks how likely would you be to use it?

B Concept Drawings

Figure 10: A snapshot concept drawing for the JubJub interface, showing a basic method editing activity and access to color coded Java for comparison.

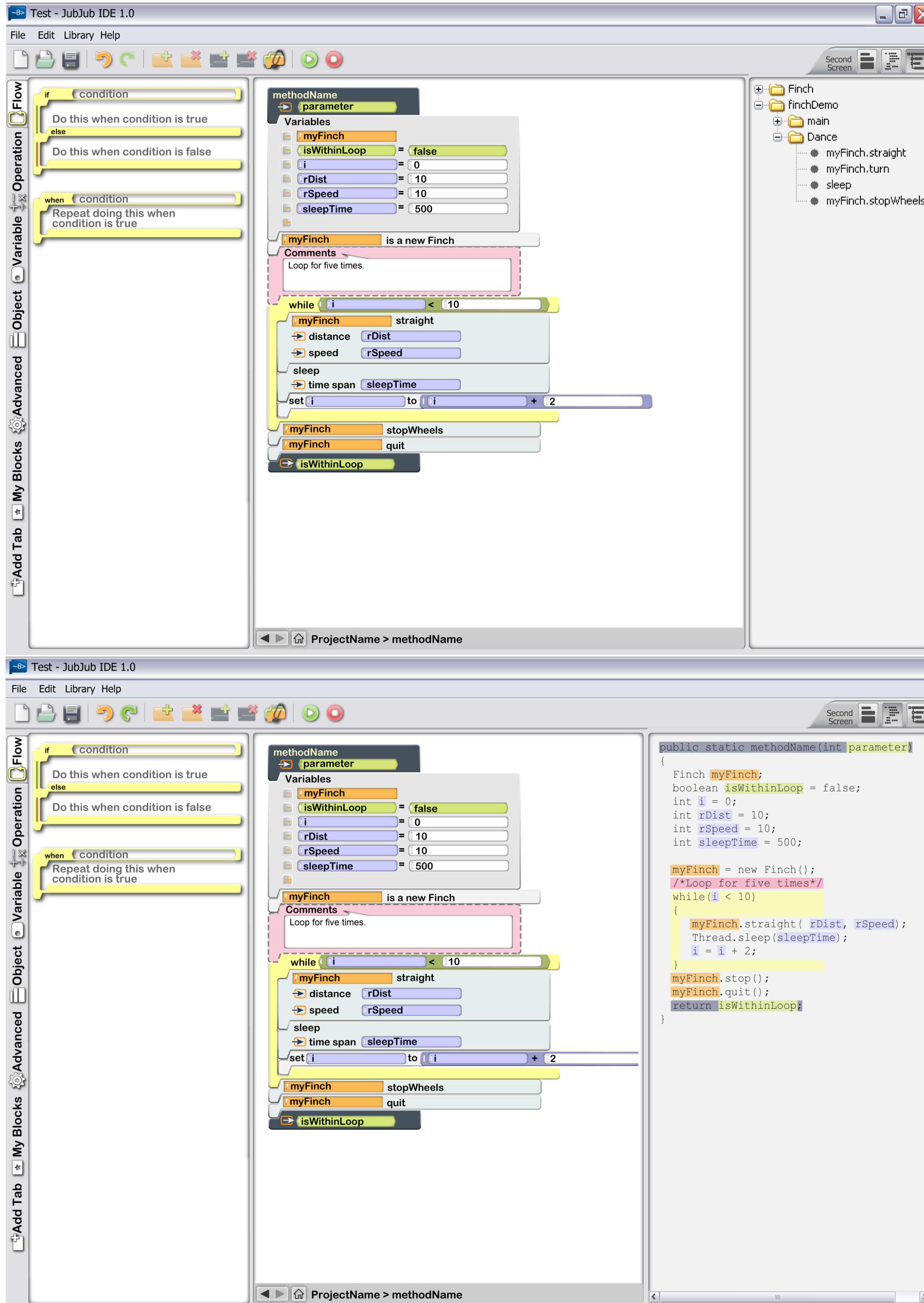


Figure 11: A snapshot concept drawing for the JubJub interface, showing the classes view and a split screen mode for editing.