**STARTING AFTER SCRATCH**

TEACHING TEXT-BASED CODING AFTER GRAPHICAL LEARNING TOOLS

A Master’s Thesis or Doctoral Dissertation Presented to the Faculty of

the Department of Computer Science

Villanova University

In Partial Fulfillment

of the Requirements for the Degree of

Master of Science in Computer Science

by Christopher Hallberg

April, 2013 (of completed thesis)

Under the Direction of

Vijay Gehlot, PhD

# Table of Contents

The Table of Contents is a consecutive listing of the names and page numbers of the

chapters and other major divisions as they appear in the body of the thesis.

2

An abstract (limited to 150 words for Master’s Thesis and 350 words for Doctoral

Dissertation) must be prepared on a single page.

3

The reference material contains a bibliography or a list of references arranged

alphabetically and any other pertinent sections, such as appendices.

# List of Tables

# List of Figures

# Abstract

*An abstract (limited to 150 words for Master’s Thesis and 350 words for Doctoral*

*Dissertation) must be prepared on a single page.*

Visual programming languages are useful and effective tools when it comes to teaching and learning how to program. However, these languages create capable programmers isolated by the limited power and professional acceptance of these learning tools. Due to their graphical, often syntax-free nature they make the next step of learning a textual language more difficult. Rather than do away with graphical language, it would be far better to provide these capable programmers with a bridge or a proverbial Rosetta Stone to guide them into a textual language.

In the following pages, I hope to present my attempts at creating such a bridge by connecting the popular visual language Scratch with the rising textual language Javascript while maintaining all the non-functional traits of the highly social Scratch environment.

# 1. Introduction

## Problem Statement

“It has already been amply shown that ``graphicality'' in itself is not sufficient to alleviate novice programmer difficulties (see e.g. [Good & Brna, 1996]): we need to provide novices with further scaffolding if we are to support them as they learn to program.” The need for transfer after almost negates the usefulness of the graphical language in the first place - Oberlander referencing Good, J. and Brna, P. (1996). Novice difficulties with recursion: Do graphical representations hold the solution? In Proceedings of the European Conference on AI in Education, Lisbon, Portugal, September 30 -- October 2, 1996.

## 1.2 Purpose of Study

Design an intermediate language or program that helps users move between a visual programming language to a textual programming language. Determine compatible and non-functionally similar base and target languages. Explore the feasibility of creating a program bridging these two languages.

## 1.3 Importance of Study

President Obama on Science and Engineering in a Google+ Hangout - <http://www.youtube.com/watch?v=Ka-C7yBu_dE> – “math and science is part of your overall educational experience, we don’t want you to be intimidated by it.”

President Obama on Computer Programming in High School in a Google+ Hangout - <http://www.youtube.com/watch?v=PClfyIbIr5Q> –

1.4 Scope of Study

## 1.5 Rationale of Study

### 1.5.1 Choice of Scratch as the Base Language

For my study, I wanted to build on a strong basis. I wanted to meet several criteria when choosing a visual programming language to build on. I wanted a language that has an effective pervasiveness in pre-college school students, especially in the middle schools. Most preferably, one that already had a large grasp on this age group. The language needed to be powerful enough to bring its users to a level where they are ready to advance to less restrained programming environments. Finally, the language I chose as the base must effectively and quickly teach the programming mind-set that allows programmers to parse ideas into language semantics.

Several visual programming languages meet these criteria. I’ve ruled out a few common visual language types for various reasons. Dataflow languages are languages that have their logic structured into a flowchart style layout. They are among the earliest attempts at visual programming languages [Johnston, 2004]. VVVV is an example of one such language. It describes itself as a “hybrid graphical/textual programming environment” [http://vvvv.org]. Code written in VVVV is laid out in nodes instead of separate files or scopes. While visual in this regard, the actual logic is written in textual Java, putting it outside of the realm of visual in my mind. In either case, the flow chart interface is not an interface conducive with teaching outside of a strict computer science environment, where logic graphs and flows are most common. In fact, I’ve drawn logic graphs identical to those found in Johnston’s “Advances in Dataﬂow Programming Languages” [Johnston] in order to learn about programming language semantics exclusively while studying here at Villanova, as they are commonly used to represent the logic of basic computers called Turing Machines. A study by J. Good has analyzed graphical, dataflow languages and has concluded that they are less than beneficial. She discovered that novices had “great difficulty in understanding data-flow representations”. [The Match-Mismatch Conjecture and Learning to Use Data-Flow Visual Programming Languages, Oberlander referencing Good, Judith (1996). The `right' tool for the task: An investigation of external representations, program abstractions and task requirements. In Gray, W.D. and Boehm-Davis, D.A., (eds.), Empirical Studies of Programmers: Sixth Workshop, pages 77--98. Ablex Publishing Corporation.]

The second type of visual programming eliminated from candidacy will be referred to as macro languages. Macro languages are languages that allow a user to assign properties to objects. These properties can be as simple as “background” to make them react to the position of the level and player or as complex as “physics” to make the object simulate gravity and collisions, normally quite a challenge to create well in code. The purest example of this is the game studio Construct 2 by Scirra [https://www.scirra.com/construct2]. Without any direct logic manipulation you can create a full game with advanced physics and graphics. It is a powerful creation tool, but not a tool for teaching coding or a programming mindset. GameMaker [http://www.yoyogames.com/gamemaker/studio] is a strong second example, but without textual coding using the environment’s unique scripting language the scope of this environment is quite limited and, again, not educationally engaging.

That narrows down the larger categories of visual languages to block-based environments, of which we have two strong candidates.

Alice is a visual programming language that allows users to create interactive 3D systems. This is accomplished by arranging nesting code chunks (resembling draggable rectangles) and pre-programmed macros (“Let mouse move any object” [Alice Web site, “A Demonstration of Alice”]) into a desired order. Originally conceived in 1995, Alice was designed for rapid prototyping for virtual reality systems [Pausch, 1995]. It was later revised by Matthew Conway into a learning system targeted to 19-year old students, your usual college freshman. It was designed to teach object-orientation and basic programming constructs. Alice has even integrated different syntaxes into its interface, allowing it to be friendlier to new users or look more like Java, in an attempt to “bridge” the gap between itself and Java [Dann, 2011].

Scratch is a visual programming environment developed at the MIT Media Lab. Color blocks can be snapped together to influence the behavior inside of a “stage”, which updates live with every code update. First developed in 2003, “the original design of Scratch was motivated by the needs and interests of young people (ages 8 to 16)” [Maloney, 2010]. Scratch’s interface naturally teaches objects, basic variable types through shapes (strings, numbers, Booleans), and parallel operations. Since its release, it has grown into a highly public programming environment[Scratch Web site] and was ranked 29th in 2012 on the TIOBE list, the highest ranked visual programing language [TIOBE, 2012 <http://www.tiobe.com/index.php/content/paperinfo/tpci/index.html>].

NEEDS MORE

### 1.5.2 Choosing the Target Language

Having selected Scratch as the base language for my tool, I focused on selecting a language that had many of the features of Scratch. Not features such as syntax or semantics, more contextual features. Scratch was designed to be **social**, so I needed a language that was **easy to share**. Programmers and users of my bridge not only needed to be able to share the finished product easily, but also need to share the code behind the project just as easily. This made me lean towards a run-time language. Unlike more traditional compiled languages, programs written in a run-time language are parsed as they are run. This means that you have one product that is code and program, as opposed to two separate products as you would have with code and its compiled binary. Another important part of being social is being relevant. While this was a minor point, I wanted to be sure my language was choice was useful in this day and age.

Scratch is **easy to approach**. This led me to look for a language that is loosely typed. A language that is loosely typed handles variable types dynamically, changing them when necessary to avoid clashing in common operations. Most loosely typed languages have one variable type for all variables. This is a step back from the functionality of Scratch, which has different shapes for string and numbers (figure x), but I felt it was a necessary choice to make the transition as smooth as possible.

Scratch is very **visual** and **immediately rewarding**. Your actions in the code are immediately reflected in the program. When you open a new project, there is no setup necessary before you start. This gives very positive feedback to the user and highly encourages learning through play and experimentation. Many languages, like C, keep their code in various modules that need to be loaded in. In any language there would be some work on my part to create a template and support functions that add specific functionality in Scratch to the target language, but I should be building on a language that is mostly ready to from the first keystrokes.

Finally, I needed a language that I knew well enough to create a template and system for. I needed to understand the language at a level that would allow me to guide users gradually. As a former Scratch student and user, I know the language thoroughly. I would need to know the semantic effect each action is taking below the surface in order to match it accurately and feasibly to a final language.

Many teachers I’ve had and worked with like to move students to Processing, another coding environment from MIT []. Processing is an extremely useful IDE for Java, that has a library of functions and templates that make coding in Java much more appealing to artists. It sets up most of the initial work for you by defining a window and global classes, which can be a chore for experienced programmers and a confusing guessing game for beginners.

Unfortunately, I think it’s a steep jump to Processing. The environment uses a very specific control flow to simply the experience for its users. This very strict flow, however, is completely different from the usual flow found in Scratch programs. In particular, the lack of easy concurrency is going to leave many Scratchers lost and seeking back to their roots. Open source is an excellent strong point for sharing, but even with an export feature, getting the finished product out there is less than ideal.

### 1.5.3 Choice of Javascript as the Target Language

I chose Javascript as the far point on the bridge because it matched my criteria perfectly. When it comes to getting started with the language, Javascript is unmatched. Javascript is a browser-based run-time language. This mean that Javascript not require a compiler, its ideal environment for execution is a web browser. Web browsers are not only one of the most pervasive pieces of software in the world [find citation], they are becoming ever more powerful. Any computer with a text editor and browser can be used as a tool to create programs in Javascript. The code for Javascript is written into an HTML file, which is the standard document of the Internet. This means that code can be placed online to be run without download (by being visited in a browser as a page) or can be put on any machine with a web browser and run without installing any other software. Javascript also rules social coding sites, being the most used language on GitHub [https://github.com/languages] and in the top 10% on Sourceforge [http://lang-index.sourceforge.net], two of the most widely used social coding sites.

The syntax of Javascript also made it very attractive. Variables are “duck-typed”, which means they are all declared the same and their type is determined automatically as they are used, even changing variable type as the program’s operations need. This is not the only way that Javascript is forgiving with its syntax; it also inserts semi-colons where you may have forgotten them and often provides easy to understand and line-specific errors. Since server-side Javascript is a relatively new technology [cite nodejs here], a majority of Javascript is available in open-source format, creating one of the largest collections of code samples imaginable: the Internet itself. In fact, 92.6% of pages use Javascript in some form [http://w3techs.com/technologies/details/cp-javascript/all/all] providing trillions of documents of working code to dive through. [Google index size, http://www.google.com/insidesearch/howsearchworks/thestory]

1.6 Definition of Terms

# 2. Literature Review

## 2.1 Importance of K-12 Computer Science Education

# 3. Methodology

## 3.1 Approach

## 3.2 Data Gathering Method

## 3.3 Database of Study

## 3.4 Validity of Data

## 3.5 Originality & Limitations

## 3.6 Summary

# 4. Presentation of Findings

4.1 Sub-heading

4.2 Sub-heading

# 5. Conclusions

# 6. Future Work

# 7. Bibliography

The reference material contains a bibliography or a list of references arranged

alphabetically and any other pertinent sections, such as appendices. Chicago Manual of Style