Scratching with All Your Fingers: Exploring Multi-touch Programming in Scratch

by

Christopher Graves

S.B., Massachusetts Institute of Technology (2013)

Submitted to the Department of Electrical Engineering and Computer Science
in partial fulfillment of the requirements for the degree of

Master of Engineering in Electrical Engineering and Computer Science

at the

MASSACHUSETTS INSTITUTE OF TECHNOLOGY

June 2014

Author .....
Department of Electrical Engineering and Computer Science
May 23, 2014

Certified by .....
Mitchel Resnick
LEGO Papert Professor of Learning Research, MIT Media Lab
Thesis Supervisor

Accepted by .....
Prof. Dennis M. Freeman
Chairman, Masters of Engineering Thesis Committee

Scratching with All Your Fingers: Exploring Multi-touch Programming in Scratch

by

Christopher Graves

Submitted to the Department of Electrical Engineering and Computer Science
on May 23, 2014, in partial fulfillment of the
requirements for the degree of
Master of Engineering in Electrical Engineering and Computer Science

Abstract

Thesis Supervisor: Mitchel Resnick
Title: LEGO Papert Professor of Learning Research, MIT Media Lab

Acknowledgments

This is the acknowledgments section. You should replace this with your own acknowledgments.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

1.1 Introduction to Scratch

1.1.1 Scratch Programming Environment

1.1.2 Introducing Tablet Scratch

1.2 Motivation

1.3 Touch Interactivity in Scratch

1.4 Thesis Overview

Chapter 2

Background

The history of multi-touch tablets stretches back to the mid-1980's when the Input Research Group at the University of Toronto began developing the first multi-touch tablets [10, 12]. Since then, due in no small part to the popularity of iOS and Android devices, multi-touch tablets have become ubiquitous in modern life and are being used in all kinds of environments from grocery stores to classrooms.

Multi-touch tablets give users the ability to have as many pointers as they have fingers, which in turn enables users to interact with applications using natural, intuitive gestures. However, this ability adds a layer of complexity for multi-touch interface developers who must develop applications that constantly process and interpret simultaneous, asynchronous touch events. Due to the inherent intricacies of developing multi-touch applications, there have been several attempts to provide API's that make app development as easy as possible while still giving developers the power to make whatever interactions they can imagine.

In this chapter I will first discuss and categorize related work in the field of multi-touch interface development and then I will present the goals we have specifically for Tablet Scratch's multi-touch block set.

2.1 Related Work

There are several tablet programming environments currently available that allow developers to create projects with multi-touch interactivity. However, they all fall into two general categories which directly conflict with our goals. The first category consists of tablet programming environments which are not sufficiently accessible to novices, i.e., have a floor that is too high. The second category consists of tablet programming environments which do not provide enough functionality to allow experienced programmers to pursue more complicated ambitions, i.e., have a ceiling that is too low.

2.1.1 Floor Too High

The vast majority of widely used multi-touch interfaces are developed in tablet programming environments which allow for great functionality, but are insufficiently accessible to beginners for our purposes. These "professional" frameworks are the most extreme examples of "high floor" tablet programming environments and include iOS Developer Library[3], Android SDK[2], ActionScript 3.0[1], and JavaScript/HTML5[7]. Note that for all of these frameworks, developers are generally writing their code on their desktop computers despite their interactions being designed for tablets.

All of these "professional" frameworks involve textual programming languages (TPLs), as opposed to graphical programming languages (GPLs). Although TPLs allow experienced programmers to efficiently create whatever they can imagine, TPLs are generally unwelcoming to novices. Since developers must remember what tools are available or look them up in large highly technical specification documents, TPLs make it difficult for beginners to explore. Furthermore, TPLs force beginners to suffer through waves of demoralizing, frustrating syntax errors before being able to develop even the simplest of programs.

Thus, it may come as little surprise that these professional frameworks handle multi-touch inputs in a manner that allows developers to create as complex user interfaces as they can imagine, but can also be esoteric and difficult for novices.

Professional frameworks all handle multi-touch inputs in a way that mirrors how they handle mouse inputs, that is, using a programming approach known commonly as the observer pattern [9]. In these frameworks, every discrete touch action triggers an "event." An event is a datatype that carries with it some information, generally including an x-coordinate, a y-coordinate, a touch identification index, and an action type, along with several more complicated attributes. The three most basic action types are "touch down" (for when a finger first makes contact with the screen), "touch move" (for each time a finger already touching the screen moves to a new discrete location), and "touch up" (for when a finger ends contact with the screen).

As touch events are generated by user interactions, they are sent to developer-defined "event handlers" to be processed. Developers define an event handler by providing a callback function which takes in a touch event as an argument and processes it according to the information the events provide. While these frameworks provide developers with the ability to process touch inputs in any manner that they want, they are not very accessible to novices, even ignoring the fact that they are TPLs. To make use of the observer pattern, developers must first understand the concept of parameters and functions, and then slog through the oftentimes dense framework-specific documentation to understand what information touch events have, when events are triggered, and how events are triggered. To their credit, most of these professional frameworks also have predefined event handlers for common touch gestures, e.g., tap and swipe, for developers to use and customize, making certain interactions easier to develop.

As a response to the complexity and inaccessibility of the professional frameworks, several tablet programming environments have emerged which amiably attempt to make application development more accessible for beginner programmers without limiting what they can create as they gain expertise.

One of the most successful of these tablet programming environments is Codea[4]. Codea uses an augmentation of the scripting language Lua, so it is still a TPL and suffers from the same accessibility hindrances. Unlike professional frameworks, Codea comes with a programming environment that allows developers to code directly on

their tablets. Compared to the professional frameworks, Codea significantly facilitates the process of developing multi-touch interactive applications. Although Codea keeps a general observer pattern structure for handling touches, it is greatly simplified. Touch events in Codea are relatively straightforward datatypes. Codea predefines a touch event handler, so all the developer has to do is designate their desired callback function as "touched" and it will automatically be called for every touch event. Furthermore, Codea adds a global touch event named "CurrentTouch," that provides touch values for an arbitrarily defined primary touch. The values stored in CurrentTouch can easily be accessed anywhere in the code, making it easy to access needed values when necessary without having to deal with event handlers. While Codea definitely makes it easier for developers to make multi-touch interactive applications, it still requires that they learn the Lua syntax and the general linear flow of Codea applications.

Like the professional frameworks Codea tries to simplify, Codea unfortunately still suffers from a computational flow that is neither explicit nor natural. The confusion surrounding the observer pattern begins with the event handlers and the callbacks. To a novice, it is not clear what calls the callback and exactly when the callback is called. It's natural, but technically incorrect, to think that the callback is called by the device immediately after the triggering event happens. Since no line in the developer's written code calls the callback, it is also natural, but incorrect, to think that the callback function runs as a separate entity from the rest of code, i.e., it is non-blocking. Thus (to many novice developers' surprise), if one runs an infinite loop in one of the callbacks, no other code in the applications will ever run after that callback is triggered. The reason for this confusion is that callback functions give the illusion of starting a new separate thread that can run simultaneously with the rest of the code, while the truth is that these callbacks are blocking functions. The problem with this illusion is not merely that it is an illusion, but that it leads novices into constructing a mental model that conflicts directly with the reality of the programming environment.

Another notable attempt to make tablet application development easier is MIT's

AppInventor [15], which allows developers to make interactive Android applications using a simplified drag-and-drop graphical programming language (GPL) rather than the professional Java based Android SDK. By being a GPL, AppInventor spares users many of the pains of TPLs, including preventing most syntax errors. However, the friendly, welcoming appearance of this environment can give beginners a false sense of security when implementing multi-touch interactivity. AppInventor still utilizes the almost entirely single-threaded observer pattern, despite furthering the illusion that callback functions are non-blocking, luring novices to fall into the same previously mentioned pitfalls.

Observer pattern frameworks are inherently unintuitive and cause many headaches even for professional software engineers, let alone novices. To quote an Adobe Software Technology Lab presentation [13]:

- 1/3 of the code in Adobe's desktop applications is devoted to event handling logic;
- 1/2 of the bugs reported during a product cycle exist in this code.

As a result, there is a small campaign among certain professional programmers to try to move away from the observer pattern in favor of what is known as reactive programming[11, 14]. One notable example of such a framework that handles touch inputs is the relatively new textual, functional reactive programming language Elm[5]. In Elm, mouse events and touch events are replaced by "signals" such as Mouse.position and Touch.touches, which always have a value which other parts of the code can persistently depend on. In theory, functional programming languages such as Elm are more intuitive due to their consistency and completeness. However, in practice, many people (professional programmers included) find it extraordinarily difficult to code without discrete states or mutable data.

2.1.2 Ceiling Too Low

At the other end of the spectrum are tablet programming environments that allow even the most inexperienced programmers to add at least some multi-touch interac-

tivity to their projects. However, the utilities of these environments are so simplistic and weak that they prevent experienced programmers from going beyond the most basic of applications. Many of these programming environments are aimed towards children younger than our targeted demographic and, as a result, try to simplify application development as much as possible.

Two current examples of such tablet programming environments include Hopscotch[6] and ScratchJr[8]. Both environments are drag-and-drop graphical programming languages that are highly inspired by Scratch, with the latter being developed in part by some of the core creators of Scratch. Furthermore, both environments allow users to develop directly on their tablets. For simplicity, multi-touch interactivity in these environments is limited to allowing objects to recognize when they or the background are tapped and running developer-designated code as a response. In a sense, multi-touch interactivity in these environments is implemented in a simplified event listener/handler framework. However, these frameworks are distinguished from the floor-too-high frameworks, because their callback functions are run as separate threads, i.e., are non-blocking. Thus, callbacks with infinite loops are not a problem and run simultaneously with the rest of the code.

While these environments make it extraordinarily easy for novices to develop simple touch interactions, such as buttons, they make it essentially impossible to do anything that is much more complicated, like a touch-controlled pong game or a drawing application. In the interest of simplicity and accessibility, these environments deny developers direct access to the exact locations of touches and preclude developers from writing scripts that can discover when touches begin and end. These limitations are fine for young children first exploring the world of digital creation, but can be constraining for older, more experienced, and more ambitious developers. Oversimplified frameworks constrain developers not only in terms of complexity (i.e., they lower the ceiling), but also in terms of project diversity (i.e., they narrow the walls). If the only touch gesture that is recognized in the framework is a tap, then all touch interactions are going to be taps.

2.2 Goals for the Multi-Touch Command Set in Tablet Scratch

As the reader might imagine, our main goal for multi-touch interactivity is to simultaneously have as low floors, as wide walls, and as high ceilings as possible. Unfortunately, low floors and high ceilings are often achieved at the expense of one other, making it necessary to further specify priorities.

Following the values behind the design of the original Scratch, we are placing a higher priority on having low floors than on having high ceilings. Any design that does not allow beginners to painlessly create the simplest touch interactions will be considered an abject failure, no matter how high the ceiling is. That being said, any design will be considered undesirable if it simplifies multi-touch interactivity to the point that many complex interactions are more than just difficult, but actually impossible to implement. On a similar note, an important aspect of the Scratch philosophy is not only to embrace the diversity of projects, but to actively encourage it. Thus, the blocks we develop for multi-touch interaction must lend themselves to being used in a wide variety of ways.

In addition to following the high-level ideals of Scratch, it is important that multi-touch interactivity be designed with some, if not all, of the lower-level principles as well. Perhaps the most important of these principles is that there are no error messages. A Scratcher's code might not always run exactly the way they want it to, but it will always run. Furthermore, the disparity between how the code runs and the Scratcher's desired outcome helps teach the Scratcher what must be changed. Hence, the multi-touch block set must be designed in a way that makes it impossible to have a syntax error. Another key Scratch principle is to make the block set as minimal as possible. In order to preserve explorability and limit confusion amongst beginners, the multi-touch block set must be designed to limit redundancy wherever possible. Finally, a part of Scratch philosophy prioritizes simplicity over 100% correctness. If a block set is simple and intuitive, but only works in 95% of use cases, it is still preferable to an overly complicated block set that works in all use cases.

As mentioned earlier, many of the principles that are being considered for the design of the multi-touch block set are directly at odds with one another, meaning that even the best designs will exhibit certain trade-offs. Despite these necessary concessions, there are multiple strong, potential designs for the multi-touch block set, all highlighting different principles over others.

Chapter 3

Case Studies

3.1 Low Floor Cases

3.1.1 Basic 1 Player Pong

3.1.2 1 Finger Drawing without a Pen

3.2 Middle Height Cases

3.2.1 Basic 2 Player Pong

3.2.2 10 Finger Drawing with Pens

3.3 High Ceiling Cases

3.3.1 Advanced 2 Player Pong

3.3.2 10 Finger Drawing without Pens

Chapter 4

Designs

4.1 Limiting to Local Gesture Recognition

4.2 Callback Paradigm

4.3 Watch Each Finger Paradigm

4.4 Watch Closest/Farthest Finger Paradigm

4.5 Watch Context Dependent "Current" Finger Paradigm

Chapter 5

Conclusion

5.1 Future Work

5.2 Closing Remarks

Bibliography

[1] ActionScript 3.0. http://help.adobe.com/en_US/FlashPlatform/reference/actionscript/3/index

[2] Android SDK. https://developer.android.com/sdk/index.html?hl=sk.

[3] Apple iOS Developer Library. https://developer.apple.com/library/ios/.

[4] Codea. http://twolivesleft.com/Codea/.

[5] Elm. http://elm-lang.org/.

[6] Hopscotch. https://www.gethopscotch.com/.

[7] JavaScript. http://www.w3.org/standards/webdesign/script.

[8] ScratchJr. http://scratchjr.org/.

[9] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. *Design Patterns: Elements of Reusable Object-oriented Software*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1995.

[10] SK Lee, William Buxton, and K. C. Smith. A multi-touch three dimensional touch-sensitive tablet. *SIGCHI Bull.*, 16(4):21–25, April 1985.

[11] Ingo Maier and Martin Odersky. Deprecating the Observer Pattern with Scala.react. Technical report, 2012.

[12] N. Mehta. A flexible machine interface. Master's thesis, University of Toronto, 1982.

[13] S. Parent. A possible future of software development. Adobe Software Technology Lab, 2006. http://stlab.adobe.com/wiki/images/0/0c/Possible_future.pdf.

[14] Atze van der Ploeg. Monadic Functional Reactive Programming. *SIGPLAN Not.*, 48(12):117–128, September 2013.

[15] David Wolber, Hal Abelson, Ellen Spertus, and Liz Looney. *App Inventor for Android: Create Your Own Android Apps*. O'Reilly, 2011.