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

hey hey hey [?]

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 [1, 2]. 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 all kinds of environments from grocery stores to classrooms.

Multi-touch tablets give users the ability of 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 app developers who must develop applications that persistently 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 they can imagine.

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

2.1 Related Work

There are currently several tablet programming environments 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 easily 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 programmer to pursue more complicated ambitions, i.e. have a ceiling that is too low.

2.1.1 Floor Too High

There are several tablet programming environments available that allow for great functionality and complete control, but are insufficiently accessible to novices for our purposes. The most extreme of these are the ”professional” frameworks used for multi-touch application development, including iOS SDK, Android SDK, ActionScript 3.0, and JavaScript/HTML5. Note that for all of these frameworks, developers generally are coding 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. Although TPLs allow experienced programmers to efficiently create whatever they can imagine, they are generally unwelcoming to novices. TPLs are generally not explorable in the sense that developers must remember what tools available or look them up in large highly technical specification documents. Furthermore, TPLs force beginners to suffering through waves of syntax errors before being able to first develop even the simplest of programs.

Thus, it comes to 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 be esoteric and difficult for novices. ”Professional” frameworks all handle multi-touch inputs in a way that mirrors how they handle mouse events, that is through ”events,” ”event handlers,” and ”callback functions.” In these frameworks, every discrete touch action triggers an event 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 that is already touching the screen moves to a new discrete location), and "touch up" (for when a finger ends contact with the screen). However, most of these professional frameworks also have predefined event handlers for common touch gestures like tap and swipe for developers to use and customize, which makes common interactions easier to develop. As touch events are generated by user interactions, they are sent to developer defined event handlers to be processed. Developers define their event handlers by providing callback functions which take in touch events as an argument and process them 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 when ignoring the fact that they are TPLs. To make use of events, event handlers, and callback functions, developers must first understand the concept of functions and parameters and then must slog through the oftentimes dense framework-specific documentation to understand what information touch events have, when events are triggered, and how events are triggered.

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. Codea uses an augmentation of the scripting language Lua, so it is still a TPL and suffers from the same accessibility hindrances. Unlike the professional frameworks, Codea comes with a programming environment that allows developers to code directly on their tablets. Compared to the professional frameworks, Codea greatly simplifies the process of developing multi-touch interactive applications. Although, Codea keeps a general event and callback structure for handling touches, it is greatly simplified. Touch events in Codea are simple data-types and making the touch event handler is simple. All the developer must 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” which 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. 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 is neither explicit nor natural. The confusion begins with the event listeners and the callbacks. At first, it's not clear what calls the callback and exactly when the callback is called. It's natural, but incorrect, to think that the callback is called by the device exactly 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 runs as a separate entity to the rest of code. Thus, it would come to many novice developers' surprise that 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 false illusion of starting a new separate thread that can run simultaneously with the rest of the code. 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, 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 from many the pains of TPLs, including preventing most syntax errors. However, the friendly 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 callback structure, despite furthering the illusion that callback functions start their own separate thread deceiving novices to fall into the same previously mentioned pitfalls.

Event handling frameworks are inherently unintuitive and cause many headaches for even professional software engineers let alone novices. To quote a Pearson presentation:

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

Clearly frameworks that handle multi-touch using single threaded event handling frameworks will not do enough to lower the floor for novices to begin making multi-touch interactive applications.

2.1.2 Ceiling Too Low

On the other side of the spectrum are tablet programming environments which allow even the most novice of programmers to add at least some multi-touch interactivity to their projects. However, the utilities of these environments are too simplistic and weak that they inhibit 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 and ScratchJr. Both environments are drag-and-drop graphical programming languages that are highly inspired by Scratch, with the latter being developed in part by the creators of Scratch specifically as Scratch for a younger audience. 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 stage is tapped and running developer designated code. In a sense, multi-touch interactivity in these environments are 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 following most novice developers' intuitions. Thus, callbacks with infinite loops are not a problem and run simultaneously with the rest of the code.

While these environments make it extremely easy for novices to develop simple touch interactions, such as buttons, they make it essentially impossible to do anything that is more complicated, like a touch controlled pong game or a drawing application. In the interest of simplicity and accessibility, these environments do not let developers get direct access to the location of touches, nor do they allow the developers write scripts that can discover when a touch begins or ends. These limitations are fine for young children first exploring the world of creation on computers, but can be constraining for older, more experienced, and more ambitious developers. Oversimplified frameworks constrain developers not only in terms of complexity (i.e. lower the ceiling), but also in terms of project diversity (i.e. narrowing the walls). If the only touch gesture that is recognized in the framework is a tap, then all of the touch interactions are going to be taps.

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

As one can probably imagine after reading the previous sections, 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 specify priorities and necessities.

Following the ideologies behind the design of the original Scratch we are placing a higher priority on having low floors over 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 to not only 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 the 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 will help teach the Scratcher what must be changed. Thus, the multi-touch interactivity must be introduced in a way that makes it impossible to have a syntax error. Also, another key Scratch principle is to make the command set as minimalistic as possible. In order to preserve explorability and limit confusion for beginners, the multi-touch command set must be designed to limit redundancy wherever possible.

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

hey hey hey [?]

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

hey hey hey [?]

Chapter 5

Conclusion

5.1 Future Work

5.2 Closing Remarks

hey hey hey [**?**]

Appendix A

Appendix

Appendix

Bibliography

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

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