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

1 Introduction **11**

1.1 Introduction to Scratch 11

1.1.1 Constructionist Learning and Scratch 12

1.1.2 Introducing Tablet Scratch 13

1.2 Motivation 13

1.3 Thesis Overview 14

2 Background **17**

2.1 Related Work 18

2.1.1 Floor Too High 18

2.1.2 Ceiling Too Low 21

2.2 Scratch Programming Language 23

2.2.1 Block Syntax 23

2.2.2 Sprite Object Model 24

2.2.3 Multi-threading 25

2.3 Goals for the Multi-Touch Interaction Set in Tablet Scratch 26

3 Case Studies **29**

3.1 Low Floor Cases 29

3.1.1 Basic 1 Player Pong 29

3.1.2 1 Finger Drawing without a Pen 29

3.2 Middle Height Cases 29

3.2.1 Basic 2 Player Pong 29

3.2.2 10 Finger Drawing with Pens 29
3.3 High Ceiling Cases 29
3.3.1 Advanced 2 Player Pong 29
3.3.2 10 Finger Drawing without Pens 29

4 Designs **31**
4.1 Limiting to Local Gesture Recognition 31
4.2 Function Paradigm 31
4.3 Watch Each Finger Paradigm 31
4.4 Watch Closest/Farthest Finger Paradigm 31
4.5 Watch Context Dependent ”Current” Finger Paradigm 31

5 Conclusion **33**
5.1 Future Work 33
5.2 Closing Remarks 33

List of Figures

Chapter 1

Introduction

1.1 Introduction to Scratch

Designed and developed by the Lifelong Kindergarten Group at the MIT Media Lab, Scratch is a graphical, drag-and-drop programming language and environment that allows users of all ages and educational backgrounds (but school-aged children in particular) to easily make and share their own stories, games, and animations[1, 2]. Scratch was first formally proposed in 2003 as a tool to help young children gain technological fluency, allowing them to use code to express themselves and manifest their imaginations [3]. Since being released to the public in 2007, over 3 million registered users have collectively created and shared well over 5 million projects [4] ranging from simple animated cards for Mother's Day to complex, interactive physical simulations. Activity in Scratch grew dramatically in the year following the public release of Scratch 2.0 in May 2013, which allowed Scratch users to develop their projects directly in the browser rather than having to download a separate Scratch development application.

Scratch users, or *Scratchers* as they are called within the community, make their programs by organizing and connecting various blocks, each representing a command, a value, or a piece of logic. After making their projects, Scratchers can easily share their creations on the Scratch website for other Scratchers to play with, comment on, and remix. There are also forums where Scratchers can share ideas, help one other, and even just hang out.

1.1.1 Constructionist Learning and Scratch

The design and motivation behind Scratch are deeply influenced by Seymour Papert's contributions to the theory of constructionist learning [5]. There are two main ideas behind the theory of constructionist learning, or "constructionism," as it is also known. The first idea is that it is best to think of learning as the process of the learner constructing ideas for themselves rather than simply as the transfer of knowledge from teacher to learner. The second idea is that learning is most effective when it centers around activities where the learners are creating products that are personally meaningful to them.

During the advent of personal computing, Papert saw the computer as a tool that could be used to easily allow children to create, explore, and learn through constructionism. In the late 1960's, Papert used the ideas of constructionism to create the educational programming language Logo, which he intended as a virtual math land in which children could concretely play with mathematical sentences and see the results. One of the most important ideas introduced by Logo was the metaphor of keeping *low floors* and *high ceilings.* Logo can be described as having a low floor in the sense that is easy for new Logo users to step into it, start exploring, and make simple projects. Similarly, Logo can be described as having a high ceiling because more experienced users can continue to use Logo to make more complex and sophisticated projects, that is to say "the sky is the limit."

Inspired by Logo, Scratch was also designed with the concept of keeping low floors and high ceilings. Unlike Logo, however, Scratch deemphasizes high ceilings in favor of placing higher priority on *wide walls* [6]. While Logo users were almost entirely limited to making drawings and patterns, Scratchers can make all kinds of projects ranging from games to physical simulations to interactive stories. Also, Scratchers can upload their own graphics and music to even further personalize their projects. Children with a wide range of backgrounds and interests use Scratch to work on projects that are aligned with their individual passions.

1.1.2 Introducing Tablet Scratch

Since 2010, there has been a massive proliferation of tablet computers. Tablets are currently being used for purposes ranging from ordering sandwiches to guiding museum goers to everything in between. Recently there have been a number of large-scale initiatives to use tablets for the purpose of educating children. While these initiatives are admirable, for the most part they are centered on using tablets to provide students with access to books, videos, and simple exercises. More simply put, these initiatives generally focus on using these tablets for consumption rather than creation. This focus reflects the fact that most people today mainly use tablets for consuming news, videos, and emails.

However, the makers of Scratch believe that children (and people in general) learn most effectively through an iterative process of designing, creating, experimenting, and exploring. As a result of this belief, in combination with these aforementioned initiatives, the Scratch team decided to provide children with a tablet application that they can use to create personalized stories, games, and animations directly on the tablet. To do so, the Scratch team decided to create Tablet Scratch, a tablet-specific version of Scratch to come out in late 2014. With Tablet Scratch, the Scratch team hoped to bring the same experience of the desktop version of Scratch to the tablet, preserving the low floors, wide walls, and high ceilings.

1.2 Motivation

One of the main complications of bringing the Desktop Scratch experience to the tablet is that people do not interact with their tablets in the same way that they interact with their desktop computers. The most salient difference in these interactions is that people use a mouse and keyboard to manipulate desktop programs, while they use touch controls to interact with tablet applications.

As a result of lacking a mouse and keyboard, tablet Scratch users will use multi-touch controls not only to create their projects, but as a main means of interaction with their projects as well. However, while the current Desktop Scratch block library has many blocks to help add mouse and keyboard interactivity, it lacks any blocks to allow multi-touch. Consequently, a multi-touch interaction set must be designed for Tablet Scratch in order to allow Tablet Scratchers to create multi-touch interactive projects.

While it is tempting to think that the mouse blocks in Scratch can be directly translated into multi-touch blocks in Tablet Scratch, there are two main differences that make multi-touch interactions necessarily more complex than mouse inputs. The first difference is that on desktop computers people interact with only one mouse, while people with tablets often interact with multiple fingers. The second is that the mouse cursor on a desktop is always present (meaning the mouse's state can be persistently tracked), while touches are only present on a tablet when the fingers are pressed down (meaning that individual touch states are ephemeral).

The currently standard systems of coding multi-touch interactivity are far too unintuitive and complex for the purposes of Scratch. Although there have been several attempts to simplify implementing multi-touch interactivity, none so far have been successful in simultaneously keeping the floor low enough and ceiling high enough for Scratch's standards. Therefore, it is necessary, and also the premise of this thesis, to design and evaluate a variety of archetypal approaches to creating the multi-touch interaction set for the forthcoming tablet version of Scratch.

1.3 Thesis Overview

In this thesis I will describe several potential designs for Tablet Scratch's multi-touch interaction set and analyze their respective merits and faults. In Chapter 2, I will analyze previous work in the field of programing multi-touch interactions, provide background on the Scratch programming language, and specify the goals for Tablet Scratch's multi-touch interaction set. In Chapter 3, I will describe an assortment of case study projects with which I will evaluate the interaction set designs. Next, in Chapter 4, I will describe several multi-touch interaction set designs and analyze them based on Scratch's design principles. Finally, in Chapter 5, I will present conclusions and outline future work possibilities.

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 [7, 8]. 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. Second, I will provide background on the Scratch programming language. Finally, I will present the goals we have specifically for Tablet Scratch's multi-touch interaction 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[9], Android SDK[10], ActionScript 3.0[11], and JavaScript/HTML5[12]. 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 [13]. 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[14]. 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 [16]:

- 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[17, 18]. One notable example of such a framework that handles touch inputs is the relatively new textual, functional reactive programming language Elm[19]. 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 interactivity 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[20] and ScratchJr[21]. 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 Scratch Programming Language

In order to achieve low floors, wide walls, and high ceilings, the Scratch programming language was carefully designed to minimize complexity while preserving versatility and power. Three of the most important aspects of the Scratch programming language's design are its minimalist block syntax, its simple sprite object model, and its intuitive approach to multi-threading [22].

2.2.1 Block Syntax

Scratch's simplicity begins with its elegant minimalistic block syntax. All Scratch scripts are created solely by connecting blocks like puzzle pieces. Each block belongs to one of only five different block types (*function*, *command*, *control structure*, *trigger*, and *definition*) and is appropriately shaped to suggest how it can be used [22].

We can think of *function* blocks as being values. These values can be straightforward like "mouse down?" or more complex, like the the sum of two other function blocks. Unlike other blocks, function blocks have no notches or bumps and thus cannot be stacked on top of each other. Function blocks are either rounded or hexagonal to signify their type. Rounded function blocks can be either strings or numbers, depending on context, while hexagonal blocks are Booleans.

Many blocks, of all types, have customizable inputs or *parameter slots.* Parameter slots come in different shapes specifying the types they accept. For example, rounded parameter slots with white backgrounds can be filled in either by the user typing in the values or by inserting a rounded function block. On the other hand, colored hexagonal parameter slots can only take in hexagonal function blocks. Since many function blocks can take in parameters that they themselves can fill, it is possible to recursively nest them as many times as desired. These shape specifications make it clear to the user how the blocks can be arranged and prevent users from connecting blocks in a way that is meaningless.

Meanwhile, *command* blocks can be thought of simply as action instructions that can vary from"move 10 steps" to "set video transparency to 50%". They generally have a notch on top and a corresponding bump on the bottom so they can be stacked on top of each other to form a set of instructions, appropriately called a *stack*. Once a particular stack is initiated, the command blocks in the stack are actuated in order from top to bottom ()as users would naturally assume).

Next, a *control structure* block can be thought of as a command block that takes in stacks of other command blocks and executes them according to some logic. For example, there are "if *boolean function block* then" command blocks that when called only execute the nested command stack if the given *boolean function block* is true. Control structure blocks are "C" and "E" shaped with appropriate notches and bumps that make it clear how to nest stacks within them.

Trigger blocks can be thought of as links between events and the stacks of command blocks that are to be executed when the events happen. The shape of these blocks can be described as block hats since they only have a bump on the bottom and no notch on top. This shape lets users know that stacks can be connected under the trigger blocks, and that the stacks will be executed once their connected trigger block's associated event occurs.

Lastly, *definition* blocks can be thought of as a special kind of trigger block whose triggering event is a user-defined command block. Since these user-defined command blocks can have parameter slots, definition blocks come with input function blocks which can only be used within the definition block's stack. These input function blocks take whatever value was passed into the user-defined command block that triggered the execution.

2.2.2 Sprite Object Model

In Scratch, each sprite maintains its own set of variables and its own set of scripts. Although one sprite is able to access the variable values of another sprite, no sprite can directly alter another sprite's variables. Similarly, although any sprite can broadcast messages for other sprites to act upon, no sprite can directly execute another sprite's scripts. This level of encapsulation makes any particular sprite's behavior significantly easier to understand, since all of the necessary information is contained within its scripts [22].

While the inability of sprites to "share code" with one another can lead to otherwise unnecessary code copying, it does not preclude any functionality. As a result, the sprite object model successfully lowers the floor without strictly lowering the ceiling (even though it might make certain complex tasks a little more difficult).

2.2.3 Multi-threading

Multi-threading, i.e., concurrent command processing, is often a more natural way for people to think about real-world behaviors than one at a time command processing. For example, consider when someone wants to tell a person how to walk down the street while chewing gum at the same time. Most people would first tell that person how to walk and how to chew gum individually and then tell the person to do both at the same time, which is to process two commands simultaneously. On the other hand, if people could only process one command at a time, it would be necessary to tell the person to merge the two activities together in a much more complicated way, like "take right step, bite down, take left step, open mouth."

Despite all this, multi-threading in most programming languages is generally considered an advanced, perplexing feature, reserved for only the most experienced programmers. However, multi-threading is an essential aspect of Scratch and even the newest of programmers quickly pick it up [22]. In Scratch, each executed stack in a Scratch's scripts will automatically run concurrently. As a result, a Scratcher could code the behavior in the previously mentioned example by creating two stacks, one handling walking and another handling chewing gum and then have them both execute on the same event.

While race conditions have been the bane of many programmers who code multiple-threaded applications, they are generally automatically avoided in Scratch. In Scratch, thread switches only take place at wait commands and at the end of loops [22], rather than between any two commands (as is the case for most other programming languages). As a result, all individual command stacks are executed in their entirety, thus following most Scratchers' intuitions. Scratch is still prone to certain race conditions, particularly as Scratchers make more and more complex projects where the timing of code execution becomes crucial. Luckily, Scratch is set up in a way where advanced Scratchers can easily create their own synchronization locks in order to gain near-complete control of thread switching.

Scratch's handling of concurrency is truly emblematic of the low floors, high ceiling philosophy. Scratchers who are new to programming generally do not know enough to even worry about race conditions; hence Scratch's handling of concurrency automatically prevents race condition from troubling them in nearly all simple cases. However, as novice Scratchers become more and more experienced and ambitious, they are more likely to run into race conditions. Fortunately, by the time they run into race conditions they will be experienced enough to figure out how to address them.

2.3 Goals for the Multi-Touch Interaction 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 interaction set must be designed in a way that makes it impossible to have a syntax error. Another key Scratch principle is to make the interaction set as minimal as possible. In order to preserve explorability and limit confusion amongst beginners, the multi-touch interaction set must be designed to limit redundancy wherever possible. Finally, a part of Scratch philosophy prioritizes simplicity over 100% correctness. If an interaction set is simple and intuitive, but only works in 95% of use cases, it is still preferable to an overly complicated interaction 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 interaction 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 interaction 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 Function 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] Mitchel Resnick, John Maloney, Andrés Monroy-Hernández, Natalie Rusk, Evelyn Eastmond, Karen Brennan, Amon Millner, Eric Rosenbaum, Jay Silver, Brian Silverman, and Yasmin Kafai. Scratch: Programming for All. *Commun. ACM*, 52(11):60–67, November 2009.

[2] Andrés Monroy-Hernández. Scratchr: Sharing user-generated programmable media. In *Proceedings of the 6th International Conference on Interaction Design and Children*, IDC '07, pages 167–168, New York, NY, USA, 2007. ACM.

[3] Mitchel Resnick, Yasmin Kafai, John Maloney, Natalie Rusk, Leo Burd, and Brian Silverman. A networked, media-rich programming environment to enhance technological fluency at after-school centers in economically-disadvantaged communities. Technical report, 2003.

[4] Scratch Statistics. http://scratch.mit.edu/statistics/.

[5] S.A. Papert. *Mindstorms: Children, Computers and Powerful Ideas.* Basic Books, 1993.

[6] Mitchel Resnick and Brian Silverman. Some reflections on designing construction kits for kids. In *Proceedings of the 2005 Conference on Interaction Design and Children*, IDC '05, pages 117–122, New York, NY, USA, 2005. ACM.

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

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

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

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

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

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

[13] 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.

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

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

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

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

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

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

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

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

[22] John Maloney, Mitchel Resnick, Natalie Rusk, Brian Silverman, and Evelyn Eastmond. The scratch programming language and environment. *Trans. Comput. Educ.*, 10(4):16:1–16:15, November 2010.