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

Accepted by .....
Prof. Albert R. Meyer
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

Here is where my abstract would go... IF I HAD ONE!!!

Thesis Supervisor: Prof. 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

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 Mathland 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. But compared to Logo, Scratch puts less emphasis on 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 delivering information and entertainment, rather than empowering children to create. 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 programming 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 projects with which I will evaluate the interaction set designs. Next, in Chapters 4, 5, and 6, I will describe three multi-touch interaction set designs and analyze them based on Scratch's design principles. Finally, in Chapter 7, 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 Multi-Touch Interfaces

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 (*command*, *function*, *control structure*, *trigger*, and *definition*) and is appropriately shaped to suggest how it can be used [22].

To begin, *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).

Meanwhile, 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.

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 Projects

It would be impossible to properly analyze the merits and faults of a variety of multi-touch interaction set designs without first considering a few of the most typical, potential use cases. In this chapter, I will present six project ideas that are representative of the kinds of projects that Tablet Scratchers of varying experience levels are likely to want to create. These six project ideas can be best thought of as three iterations of two basic projects: a Pong-style project and a finger-painting project. The three iterations are increasing in complexity and represent desirable projects for Scratchers at three different experience levels. I categorize the three iterations as *low floor cases*, *middle height cases*, and *high ceiling cases*. The low floor cases represent simple, basic projects that even novice Scratchers should be able to intuitively create with little or no prior experience. Meanwhile, the middle height cases represent slightly more complicated additions that ought to be implementable by a Scratcher with significant experience. Finally, the high ceiling cases represent augmentations that an expert Scratcher might aspire to make. Together, these projects will serve as benchmarks for which I evaluate the multi-touch interaction set designs that I will present and analyze in the following three chapters.

Introduction to Pong

Designed by Allan Alcorn in 1972, Pong is widely considered to be the first culturally significant video game. Although rather simple in its original incarnation,

many more complex variations have since been created. In fact, Pong-style games are quite popular within the Desktop Scratch community, as evidenced by the thousands of variants that Scratchers have created and shared. Pong is a particularly fitting project to demonstrate the floors and ceilings of the multi-touch interaction set designs, since Pong projects can begin with very simple touch interactions that later become complex through the addition of more users and new actions.

Introduction to Finger Painting

Since the early 1900s, children all over the world have been expressing themselves through the art of finger painting. For many, painting with one's fingers is a calming, liberating experience. Due to their capacity for touch interactivity, tablet computers naturally lend themselves to being a means for finger painting, albeit cleaner and less textured. Like Pong, finger painting is another project that will help with presenting the floors and ceilings of the multi-touch interaction set designs. Although implementing painting for just one finger at a time is rather simple, the complexity increases dramatically as support for more fingers is added.

3.1 Low Floor Cases

It is worth restating that it is of paramount importance that the multi-touch interaction set be designed with a low floor in mind. Novice Tablet Scratchers must be able to easily figure out how to use the multi-touch interaction set to create simple projects that are responsive to touch. If beginners are able to create simple touch interactions without much difficulty, their early success will encourage them to augment their touch interactions and continue tinkering with their projects. Otherwise, if the process of creating simple touch-interactive projects is painful, many beginners will become frustrated and will forgo improving their projects. Here I specify two examples of basic projects that new Scratchers should be able to create without much assistance or hardship.

3.1.1 Basic One-Player Pong

The simplest form of Pong consists of a single paddle and a ball, which moves around the *stage* (i.e., the background of a Scratch project) at a constant speed, rebounding whenever it hits either the paddle or a wall. The paddle has a fixed y-coordinate (and thus cannot be moved up or down), but can be moved left and right. To begin any Pong project, one must first implement this basic functionality.

This single-player form of Pong has for a long time been one of the example starter projects in Desktop Scratch. In the starter project, the x-coordinate of the paddle is simply mapped to that of the mouse cursor. Therefore, users control the paddle by moving the mouse left and right. Note that in this case, the mouse controls the movement of the paddle regardless of whether or not the mouse cursor is hovering over the paddle. Unfortunately, we cannot use this control scheme for a Tablet Scratch project, since tablets do not have mice.

For this case project, I will specify an analogous touch control scheme for tablets. In the Tablet Scratch equivalent of this starter project, the paddle's x-coordinate will begin at an arbitrary default state. When a user touches the project, the paddle's x-coordinate will follow the x-coordinate of the user's touch. After the touch is lifted, the paddle's x-coordinate can either remain where it is or return to the arbitrary default state.

The purpose of this case project is to test how easily novice Scratchers can create a single-finger touch interaction that relies on extracting coordinate information from a touch (in this case the x-coordinate) using the various multi-touch interaction sets. As a result, it will be assumed for this project that the user never presses two or more fingers on the tablet at the same time and I will leave it unspecified how the implementations of this project should handle simultaneous touches.

3.1.2 One-Finger Painting

In the same way that one-player Pong is the simplest form of interactive Pong, a one-finger painting project is the most basic form of a finger-painting application. At

its simplest, a one-finger painting application starts with a blank stage and paints an arbitrary color wherever the user touches the screen with one finger at a time. Again, it will be assumed for this project that the user will not touch the screen with more than one finger at a time, so I will leave it unspecified how implementations should handle cases when the assumption is broken.

In Desktop Scratch, an analogue of this project can be created easily by using a *pen*. As an homage to its predecessor, Logo, each sprite in Scratch carries a pen, which, when placed down, traces the movement of the sprite onto the stage. Consequently, a Scratcher would begin a "mouse painting" project in Desktop Scratch by first creating a "paint-brush" sprite that follows the mouse cursor wherever it goes. Next, the Scratcher would program the paint-brush sprite's pen to start in the up position, but go down only when the mouse clicks down (meaning the pen only traces the movement of the mouse when the mouse is clicked down).

For the Tablet Scratch version of the project, the Scratcher would want the paint-brush sprite to initially have its pen up. Whenever a finger is pressed down, the sprite must be programmed to first move to the touch location, then put its pen down, and finally follow the touch until it is lifted. At the moment the touch is lifted, the sprite must be programmed to lift up its pen. Accordingly, the purpose of this case project is to demonstrate how easily a novice Scratcher can code a sprite to follow a touch and react to when the touch begins and ends.

At its core, this project is a simple exercise of recognizing and handling when a single finger is pressed down, when it moves, and when it is lifted. This capability is the building block to coding virtually all single finger touch interactions. Thus, an interaction set that allows novice Tablet Scratchers to easily figure out how to implement this case project would be quite powerful.

3.2 Middle Height Cases

As Tablet Scratchers gain experience, they will naturally want to augment their simpler projects to develop progressively more complex ones. In addition to being intrin-

sically rewarding, the process of building upon earlier projects is a fundamental part of constructionist learning. Therefore, the ideal multi-touch interaction set must be designed so that as Tablet Scratchers master the basics of simple touch interactions, they can continue playing and figure out how to make more complex ones. If, on the other hand, the multi-touch interaction set is designed such that touch interactions more complex than the low floor cases are terribly difficult to achieve, then Tablet Scratchers would be stuck developing simple touch interactions and thus robbed of a learning opportunity. Here I present two natural augmentations an experienced Tablet Scratcher might want (and should be able) to add to the two previously described low floor cases.

3.2.1 Basic Two-Player Pong

After a Tablet Scratcher successfully creates a simple one-player Pong project, the logical next step is to add a second player. For this case project, we again have a ball which moves around the stage with a constant speed, rebounding when it either hits a wall or a paddle. However, instead of one paddle with a fixed y-coordinate that moves only left and right, there are now, on either side of the stage, two paddles with fixed x-coordinates that move only up and down. Note that unlike in the one-player Pong project, this interaction cannot be developed in Desktop Scratch due to being limited to a single mouse.

Clearly the controls from the one-player Pong project will not work in this case, because there could only be one touch at a time and that touch would control both paddles. For this case, I will assume that the paddles only follow the y-coordinate of a touch if the location of the touch is within a defined control range of the paddle and that the control range is small enough that no touch can simultaneously control both paddles. To make things simpler, it will be assumed that there are never more than two simultaneous touches and there is at most one touch at a time within either paddle's control range.

The purpose of this case project is to compare the relative ease of using the various multi-touch interaction set designs to develop projects that require multiple

sprites to respond to the touches that are closest to them. Without this functionality, many (even simple) multi-finger interactions will be impossible to develop. Although the capability for sprites to respond to all touches on the screen is obviously more powerful, an extensive range of the most common multi-touch interactions can be developed when the sprites' ability to respond to touches is limited to the ones that are closest to them.

3.2.2 Two-Finger Painting

Similar to how two-player Pong is the natural next step after one-player Pong, two-finger painting is the natural next step after one-finger painting. In two-finger painting, the user is able to draw on the stage with two fingers simultaneously instead of being limited to just one finger at a time. Again for simplicity, it will be assumed that the user never has more than two fingers simultaneously pressed on the tablet. Also note that, again, this kind of interaction cannot be developed for a single mouse in Desktop Scratch.

This appended functionality adds a significant amount of complexity to the project. Now instead of having just one paint-brush sprite, there necessarily must be two paint-brush sprites to follow the up-to-two fingers pressed on the tablet. Moreover, when two fingers are being pressed at the same time, these two paint-brush sprites must be able to distribute themselves amongst the fingers, so that both fingers have a following paint-brush sprite.

In contrast to the two-player Pong game, it is not enough in this project for the paint-brush sprites to simply follow the closest touch. If the paint-brush sprites were to do so, they would be prone to end up following the same finger leaving the other finger without a paint-brush. There are many potential projects where it is necessary for sprites to distribute themselves among touches. Programming sprites to always follow separate fingers is a challenge, so the purpose of this project is to help distinguish which of the multi-touch interaction set designs make it easier.

3.3 High Ceiling Cases

While Scratch's mission prioritizes having a low floor and wide walls, its high ceiling has been a major factor in its success. There are certain complex tasks, such as creating 3D graphics and online multiplayer capabilities, that are particularly difficult to develop in Scratch, but are possible to implement. Every day a few expert Scratchers implement and share projects that continue to push the bounds of what is possible to create in Scratch. Not only do the expert Scratchers gain a valuable learning experience by coding these challenging projects, but by sharing them with the community they inspire and educate Scratch beginners. By playing with the high ceiling projects and looking at how they were executed, less experienced Scratchers are motivated to continue challenging themselves and exploring what is possible. Thus, while complex multi-touch interactions might not necessarily be easy to program in Tablet Scratch, they should at least be possible to develop. Here I present two projects which feature advanced augmentations to the middle height cases.

3.3.1 Advanced Two-Player Pong

While a basic two-player Pong game can be fun to play for a while, it's quite simple and eventually gets boring. The ball moves at a constant speed and players are limited to just moving their paddle up and down. In an attempt to make the game more interesting, an expert Scratcher might try adding a *bump* action. A bump action causes the paddle to jut forward towards the opponent for a half-second before returning to its previously fixed x-coordinate. If the paddle hits the ball when it is in the process of a bump, the ball's speed increases by a constant factor. The bump action adds a little more skill and excitement to the basic two-player Pong project, but the question that remains is how will users instigate a bump action.

For this case project, a player will bump their paddle each time a second touch is pressed within a paddle's control range while there is already a touch there. More simply put, albeit slightly less rigorously, a bump happens whenever a user already using one finger to move the paddle presses their second finger close to the paddle.

Note that if the second finger is first pressed outside of the paddle's control range and then slid into its control range, then the bump should not be initiated. The second touch must be pressed in the control range of the paddle to cause it to bump.

At the core of this interaction is the capacity for sprites to recognize and handle not only the touch closest to them, but also all subsequent nearby touches. Although this interaction feels natural for this particular project case, it is very specific and unconventional. The ability to design and implement complex, multi-touch gestures adds a powerful means of personalizing Tablet Scratch projects. Hence, the ideal multi-touch interaction set design will make it possible to develop complex multi-touch controls like the one exhibited in this project.

3.3.2 Ten Finger Painting

The next step for the finger-painting project after adding support for a second finger, is to add support for even more fingers. Since many Android devices can recognize up to ten simultaneous touches (not to mention that only a few people have more than ten fingers) this case project will be a finger painting application that supports up to ten simultaneous finger touches. Many expert Tablet Scratchers will naturally try to make projects that allow users to deploy as many fingers as possible. Thus, the purpose of this case project is to test how well the multi-touch interaction sets make it possible to handle a large number of simultaneous touches.

Notice that the problems for this project are essentially the same as those regarding the two-finger painting project, but scaled by a factor of five. While the similarities might make this case project appear redundant, certain multi-touch interaction set designs make it easy to handle one or two fingers but terribly difficult to handle even a few more fingers. Meanwhile, other multi-touch interaction set designs make it a little challenging to handle one or two fingers, but not any more challenging to handle ten. I will discuss these trade-offs in the following three chapter as I introduce and analyze a selection of multi-touch interaction set designs.

Chapter 4

Closest Finger Design

As mentioned previously, one reason why programming multi-touch interactions can be so difficult is that there can be up to ten simultaneous touches on the tablet at any given moment. Since one touch is a more manageable input than ten, multi-touch input handling in Tablet Scratch can be greatly simplified by restricting each sprite to only being able to listen to the touch closest to itself. This particular restriction is the main premise behind the *Closest Finger Design*, the first of the three multi-touch interaction set designs that I will be presenting.

In this chapter (as will be the pattern for the two subsequent chapters), I will first specify the interaction set. Then, I will evaluate the design on the three iterations of case projects that I described in the previous chapter. However, at the end of this chapter, I will also present a possible extension that provides additional functionality to the interaction set design, but at the cost of reducing simplicity and understandability.

4.1 Interaction Set Specification

The Closest Finger Design can best be thought of as a direct analogue to the Desktop Scratch mouse interaction set. In this design, sprites can only listen to one touch at a time, namely the closest touch at the moment of evaluation. As a result of this

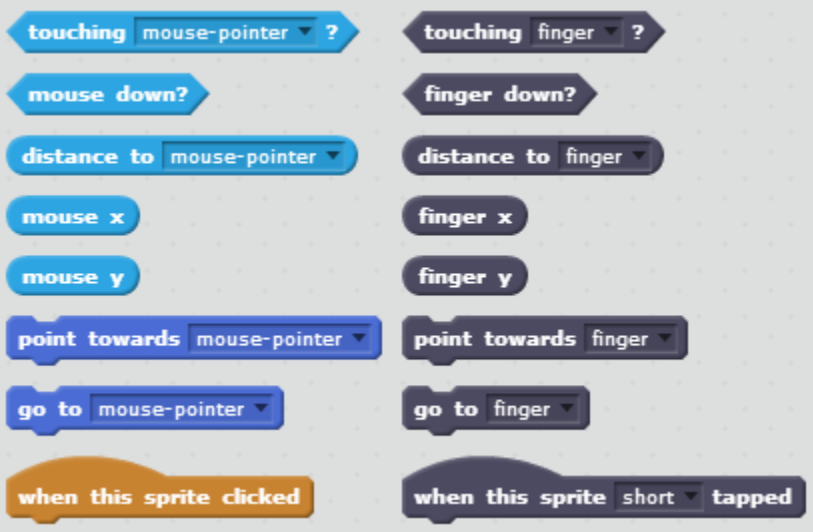

Figure 4-1: Desktop Scratch's mouse interaction set with analogous Closest Finger Design blocks.

one-touch restriction, the set of touch blocks in the Closest Finger Design closely resemble the set of mouse blocks that appear in Desktop Scratch.

In fact, the Closest Finger Design has a corresponding touch block for every mouse-related block in Desktop Scratch (Figure 4-1). While the mouse blocks in a Desktop Scratch script are influenced by the status of the lone mouse, the Closest Finger Design blocks in a sprite's scripts are, in general, similarly influenced by the status of the single touch that is currently closest to that particular sprite.

There are (as of now) eight blocks in Desktop Scratch that are related to mouse inputs: five function blocks, two command blocks, and a single trigger block. I will present all of these blocks while also introducing and specifying their Closest Finger Design block counterparts.

4.1.1 Function Blocks

The five mouse-related function blocks in Desktop Scratch are labeled "touching mouse-pointer?", "mouse down?", "distance to mouse-pointer," "mouse x," and "mouse y." Following their namesakes, the "touching mouse-pointer?" and "mouse down?" blocks both produce Boolean values (e.g., *true* or *false*) corresponding to the current

answer to their respective questions, while the "distance to mouse-pointer," "mouse x," and "mouse y" blocks produce integer values (e.g., *10*, *-33*, *147*, etc.) that similarly reflect their respective values.

The first two equivalent Closest Finger Design function blocks, which I have labeled "touching finger?" and "finger pressed?", are relatively straightforward.[1] As the name suggests, the "touching finger?" block takes on a value of *true* when a finger pressed on the tablet is touching the sprite using the block and a value of *false* otherwise. Similarly, the "finger pressed?" block takes on a value of *true* when at least one finger is pressed on the stage and a value of *false* when there are no touches.

The remaining three function blocks, labeled "distance to finger," "finger x," and "finger y," are a little more difficult to articulate. In general, the "distance to finger" block evaluates the distance from the sprite to the touch that is currently closest to the sprite (i.e., the touch that has the shortest two-dimensional distance to the center of the sprite), while the "finger x" and "finger y" blocks represent the values of the latest x and y coordinates of that touch. When no fingers are currently being pressed on the tablet, "finger x" and "finger y" will take the values corresponding to the location of the last touch to have been made during the execution of the project. If the project just recently began and touches have yet to be made, "finger x" and "finger y" must take on an arbitrary value. Since there is no "null value" in Scratch and error messages are generally avoided, I have chosen for both "finger x" and "finger y" to evaluate to 0 when there are no prior touches. The "distance to finger" block will always take the value of the distance between the current location of the sprite and the location with the coordinate values of "finger x" and "finger y."

4.1.2 Command Blocks

The two mouse-related command blocks in Desktop Scratch are the "point towards mouse-pointer" and "go to mouse-pointer" blocks; both of these, when executed,

[1]Note that these blocks reference "fingers," but, in reality, most tablets cannot actually tell the difference between a touch coming from a finger or a touch coming from any other body part. However, for the sake of concreteness (and to avoid the potential confusion of a "touch down" block), I have decided for blocks in the Closest Finger Design to refer to "fingers" instead of "touches."

cause sprites to act as their names imply. Their equivalent Closest Finger Design blocks are similarly named "point towards finger" and "go to finger." Both of these blocks behave in a similar manner to their mouse counterparts, but focus on the location of the closest touch as opposed to that of the mouse.

When there is at least one touch on the tablet, the location of interest for the blocks becomes that of the closest touch. Meanwhile, when no touches are currently present, the location of interest becomes the last location of the last touch. Finally, when no touches have been made during the execution of the project, the location of interest becomes the arbitrarily chosen coordinates of $\{0, 0\}$.

To put it simply, executing the "point towards finger" block causes the sprite to point towards the location at the present coordinate values of "finger x" and "finger y." Similarly, executing the "go to finger" block causes the sprite to go to that same location.

4.1.3 Trigger Block

Finally, the sole trigger block in Desktop Scratch's mouse interaction set is the "when this sprite is clicked" block. When triggered, this block initiates the execution of its connected stack each time the sprite is clicked by the mouse. Here, a click is defined as pressing down on the mouse left button and quickly releasing it. If the sprite is clicked a second time before the connected stack has finished its first execution, Scratch restarts the execution of the stack. Note that because of Scratch's thread-switching behavior, this can only happen when the stack contains loops or long "wait blocks."

The equivalent trigger block for the Closest Finger Design is the "when this sprite is tapped" block, which is triggered whenever a sprite is tapped. A sprite is said to be "tapped" whenever a finger is pressed down on the sprite and quickly lifted. The "when this sprite is tapped" block handles repeat triggerings in the same way as its mouse counterpart, i.e., by restarting the execution of the connected stack. Note that this trigger block can be triggered by any touch, not just the closest touch, although in practice the triggering touch will almost always be the closest touch.

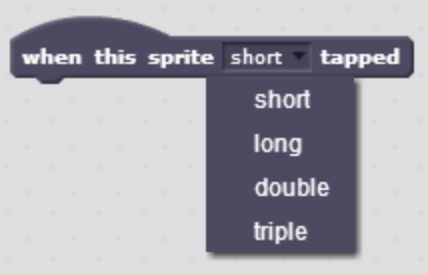

Figure 4-2: The "when this sprite is tapped" block expanded to demonstrate some of the simple gestures that could easily be supported.

Since a variety of single-touch gestures, including swiping, long taps, and double taps, are commonplace in tablet applications, it may be beneficial to augment the "when this sprite is tapped" block to support other common gestures (Figure 4-2). Additional support for common single-touch gestures lowers the floors to implementing those interactions, thus allowing beginning Scratchers to easily create a wide range of interactive projects. Nevertheless, the interaction set is not fundamentally changed by adding support for other common single-touch gestures, so I will not be discussing at length which specific gestures are best to include.

4.2 Evaluation

The simplicity of the Closest Finger Design makes the simple low floor case projects easy to implement, but makes some of the harder case projects literally impossible to implement. While it is obvious that the Closest Finger Design simplifies the handling of single-touch inputs, it is surprising that a few common multi-touch interactions are also easy to implement using the design.

4.2.1 Low Floor

Due in part to the assumption of the existence of at most one touch on the tablet at a time, the Closest Finger Design excels with the low floor case projects.

For example, the controls of the one-player Pong game can easily be implemented by constantly updating the x-coordinate of the paddle to the value of "finger x" (left

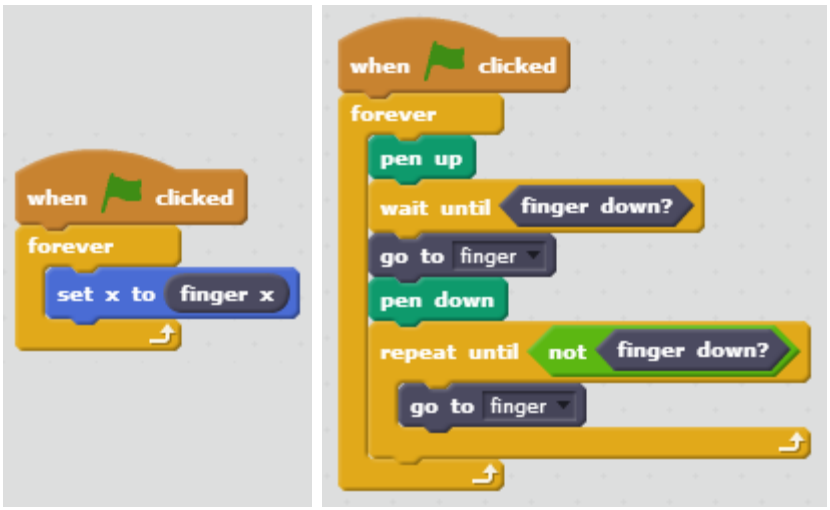

Figure 4-3: On the left is a sample of how the controls for one-player Pong could be implemented using the Closest Finger Design. On the right is a sample script using the Closest Finger Design to program the paintbrush sprite in one-finger painting.

side of Figure 4-3). At the beginning of each game, the paddle's x-coordinate will first be mapped to 0 (the arbitrary default value of "finger x" before any touches have been made). After the first touch has been made, the paddle will follow the x-coordinate of the touching finger and will remain at its last location whenever the finger is lifted.

Implementing the controls for the one-finger painting project is a little more complicated, but is still straightforward and definitely within the capabilities of a novice Scratcher (right side of Figure 4-3). Essentially, all the Scratcher must do is program the paint-brush to first wait for a touch to be made, then go to the touch's location and, with its pen down, continue following the touch until the finger is lifted. The Closest Finger Design, along with the additional framework blocks, makes programming this script almost as easy as just listing the instructions in plain English (or any other of the sixty-six languages Scratch supports).

Recall that both of the low floor case projects were notable because they were the only two presented that could be implemented in Desktop Scratch using mouse controls. As a result of the design's direct mapping with the Desktop Scratch mouse interaction set, many mouse-controlled Desktop Scratch projects can be directly translated into equivalent touch-controlled Tablet Scratch projects. This feature is of particular

Figure 4-4: Here is a sample script for controlling the paddle in a basic two-player Pong game using the Closest Finger Design.

interest, because it may later become desirable for projects to be transferable between the desktop and tablet versions of Scratch.

4.2.2 Middle Height

Although both middle height case projects feature two sprites being controlled with up to two simultaneous touches, one of the two projects demonstrates the Closest Finger Design's greatest strength, while the other highlights its worst flaw. The two-player Pong game is well suited for this design, because the two paddles must each be programmed to follow the touch that is within their respective control ranges. On the other hand, the two paintbrush sprites in the two-finger painting project must be programmed to distribute themselves among up to two touches, a task that is all but impossible to program using the Closest Finger Design.

The controls for the basic two-player Pong game are extremely similar to that of the one-player Pong game when using the Closest Finger Design. In order to implement the interaction, each paddle can easily be programmed to constantly check to see if the value of "distance to touch" is within the control range and, if that is the case, move the paddle vertically to the coordinate value "finger y" (Figure 4-4).

Even though each sprite can only listen to the touch nearest to itself, many multi-touch interactions are still possible to implement. For example, it would also be simple to implement a multi-touch keyboard that allows multi-note chords to be played, just by programming each key to play its note when touched. The important factor

in these implementable multi-touch interactions is that each sprite is only concerned with its closest touch, which makes the design's touch restriction advantageous rather than detrimental.

Despite the ease of implementing the basic two-player Pong game, it is prohibitively difficult to code the two-finger painting project using the Closest Finger Design. The main obstacle for this implementation is that each paintbrush sprite can only see the touch closest to itself, even though one of the paintbrush sprites might need to follow the touch furthest from itself.

One might think first to try to program the paintbrush to follow its closest touch only if the other paintbrush is not following it, but there are two significant problems with this approach. First, touches in the Closest Finger Design have no labels, so the only way to see if two sprites are listening to the same touch would be to compare their respective "finger x" and "finger y" values, which is an inelegant solution. Secondly, and more importantly, each paintbrush can only see the touch nearest to itself, so even if one paintbrush sprite realizes that its closest touch is already being followed, it would not know where to go to find the other touch (if it even exists) and would thus have to travel around the stage looking for an "unfollowed" touch. While there may in principle be an implementation of the two-finger painting project using the Closest Finger Design, that implementation would have to be so nonintuitive that it would not be worth considering for the purposes of this evaluation.

4.2.3 High Ceiling

Unfortunately, the simplicity of the Closest Finger Design makes the high ceiling projects essentially impossible to implement. Many complex multi-touch gestures simply cannot be implemented practically when each sprite can only listen to one touch. However, it is worth restating that many of the most popular tablet applications do not involve complex multi-touch gestures, and those that do can often be redesigned to be functional with simpler controls.

Since the advanced two-player Pong game requires that each paddle simultaneously follow one touch while listening for subsequent touches that might initiate bump,

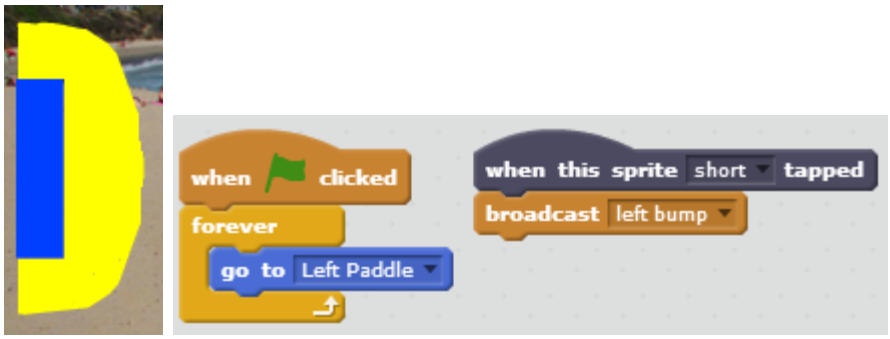

Figure 4-5: On the left is a sample bump detector, colored yellow instead of transparent for visibility. On the right is the bumper detector script which programs the sprite to always follow the left paddle and to initiate a bump when tapped.

it is strictly impossible to implement it exactly as specified using the Closest Finger Design. That being said, it would not be too difficult to make a functionally similar game with slightly more restricted controls using this interaction set.

The implementation becomes significantly easier if the controls of the advanced two-player Pong game are changed so that each paddle is moved by dragging and bumps are initiated by taps that are close to, but not directly on the paddle (Figure 4-5). If this were the case, the paddle would simply be programmed to follow the closest touch only if the touch is within the bounds of the sprite (which can be checked using the "touching finger?" block). To listen for bump-initiating taps, a transparent crescent-shaped bump detector sprite would then be created for each paddle. Both bump detectors would be programmed to follow their corresponding paddle. When a bump detector gets tapped, it would simply tell its corresponding paddle to initiate a bump.

Unfortunately, the ten-finger painting project does not have a functionally equivalent alternative that is implementable with the Closest Finger Design. Since each paintbrush sprite can only see one touch and they have no practical way to distinguish which touches are not being followed, there does not exist a reasonable implementation for this project.

4.3 Possible Extension

Many of the limitations of the Closest Finger Design are the direct result of sprites only being able to listen to one touch at a time. However, it is this restriction that makes simple single-touch interactions easy to program. Hence, the challenge in extending the Closest Finger Design to allow sprites to listen to multiple fingers lies in preserving the ease of programming simple touch controls.

4.3.1 Specification

The possible extension I will present is to add to the Closest Finger Design a new special trigger block named "when finger pressed." Note that the trigger for this block is different from that of the "when this sprite is tapped" block. First, the "when finger pressed" block is triggered by all touches, even ones whose locations are not on the sprite. Also, the "when finger pressed" block is triggered whenever a finger first makes contact with the tablet, as opposed to only being triggered by a quick tap (defined as a quick touch and release). Furthermore, the "when finger pressed" block has two special properties which differentiate it from all other trigger blocks.

The first special property is that the *touch of interest* in the execution of a stack of blocks under a "when finger pressed" block becomes the touch that triggered the execution of that trigger block, instead of just the closest touch. Before the addition of this extension, all of the five function blocks and two command blocks in the Closest Finger Design have always been solely focused on the sprite's presently closest touch, i.e., the touch of interest was always the current closest touch. In nearly all case with the extension, this is still holds true. However, when these blocks are in a stack under a "when finger pressed" block, they correspond to the triggering touch as opposed to always corresponding to just the closest touch. Furthermore, throughout the execution of the connected stack, the Closest Finger Design blocks will always refer to the triggering touch. If the triggering touch ends (i.e., is lifted) before the execution of the stack ends, the location of interest for all of the location-related Closest Finger Design blocks becomes the last location of the triggering touch.

This property allows sprites to listen to all touches as they are made, but only within the stack of blocks under a "when finger pressed" block. Furthermore, the sprite loses access to the touch that triggered the "when finger pressed" block as soon as the connected stack has finished being executed (that is, unless the touch is or becomes the sprite's closest touch). Thus, for some interactions, it will be beneficial to assemble the stack to continue running until the touch is lifted. However, this kind of script raises the question: what happens when a second touch is made while the stack under the "when finger pressed" block is still executing from the first triggering? The answer to this question is the "when finger pressed" block's second special property.

Instead of ignoring the second trigger or restarting the execution of its stack (as the other trigger blocks do), the "when finger pressed" trigger block essentially "clones" its stack and executes it in another a thread. In each of these identical threads the touch of interest becomes the touch that triggered the making of the thread. Thus when three simultaneous touches are made, the stack connected to the "when finger pressed" block will execute three threads in parallel, each listening to a separate touch.

4.3.2 Use Cases

When augmented with the "when finger pressed" block, the Closest Finger Design becomes powerful enough to implement the advanced two-player Pong game relatively easily. The paddles can use the "when finger pressed" block to first find a touch to follow and then, when following the touch, listen for a bump-initiating touch. The basic idea of such a script is presented in Figure 4-6.

The addition of the "when finger pressed" block still does not make it possible to implement the multi-finger painting projects elegantly, but it does make it technically possible and significantly more practical to implement. Although all of the paintbrush sprites can listen to all of the touches with this augmentation, there still is no way to see which touches are already being followed, other than the rudimentary method of checking if any of the paint brush sprites are already at the location of the touch in question. Thus, the multi-finger painting projects can be implemented by having

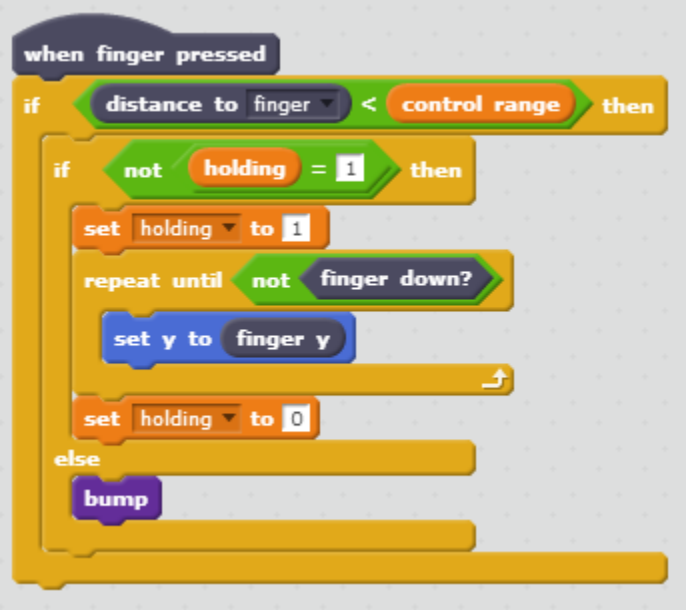

Figure 4-6: Sample paddle script for advanced two-player Pong using the "when finger pressed" trigger block.

each paintbrush use the "when finger pressed" block to listen to all of the touches being made. If the paintbrush is not already following a touch, the paintbrush will follow the first incoming touch it encounters to the completion of the touch if the touch is not already being followed. In Figure 4-7, I present the basic idea of this script.

4.3.3 Potential Issues

Although it is cause for concern that the "when finger pressed" block's special influence on the touch blocks within its stack can be confusing to novice Scratchers, the main issues with the extension are a result of the block's thread-splitting feature.

One of Scratch's main design philosophies maintains that the execution of code should be visible. Since Scratch blocks are highlighted as they are executed, Scratchers can generally see in real time how their scripts are being executed. However, with the thread splitting, it is not clear how best to exhibit its execution when it is split into multiple threads. One idea would be to actually make copies of the stack as

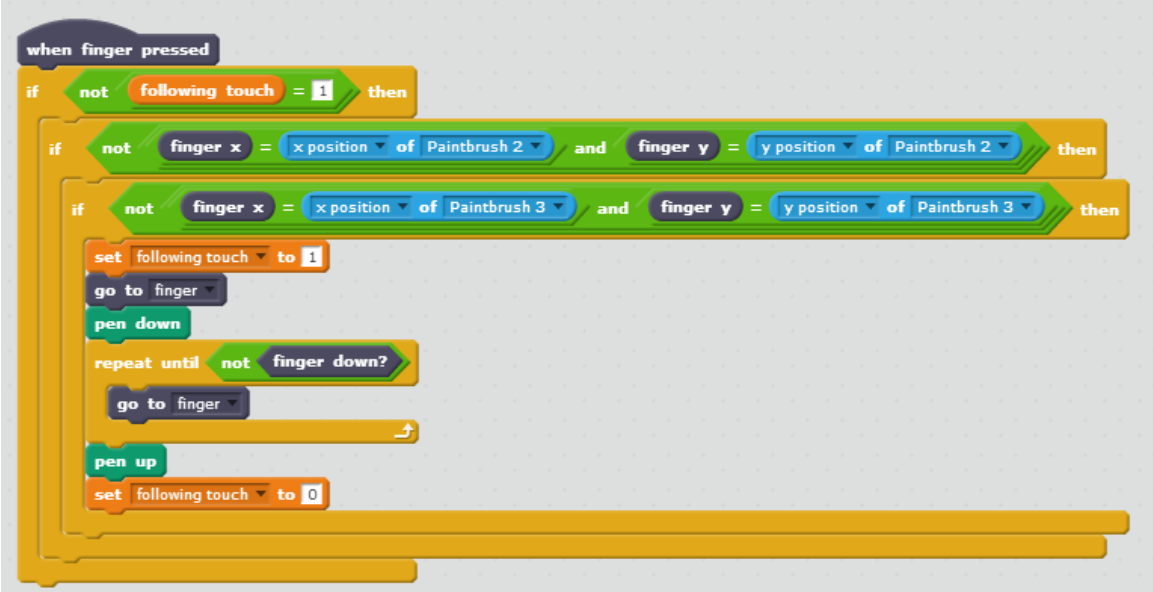

Figure 4-7: Sample paintbrush script for three-finger painting using the "when finger pressed" trigger block. Note that more fingers can be supported by adding more paintbrush sprites and additional "if checks."

the threads are being created, but the tablet screen is so small that it could quickly become unwieldy when handling multiple simultaneous touches.

More important, however, is the problem caused when a novice Scratcher innocently puts an infinite loop in the stack under a "when finger pressed" block. In this case, a new, never-ending thread would be created at the initiation of every single touch. Since there is no bound on the total number of touches that can be made throughout the execution of a project, the number of threads that could be created is limitless. This potential for rapid thread propagation makes the project vulnerable to crashing from running out of computer resources without any indication to the novice Scratcher as to what is the cause. One possible solution would be to limit the number of threads to ten (since there can only be at most ten simultaneous touches), but the limit would be difficult to express visually and might restrict intentional use of the thread-splitting feature.

Chapter 5

Relative Indexing Design

While the Closest Finger Design does greatly simplify the task of handling up to ten simultaneous touches, it does so by avoidance. By restricting each sprite to only being able to listen to one touch at a time, the Closest Finger Design eludes having to handle ten touches within the same script. Even though this approach is reasonable, there are other potential strategies for simplification without such a restriction.

For example, one natural way to think about handling the up to ten simultaneous touches is to think of them as ten persistent touch-objects, among which all touch inputs are distributed. Since there can be at most ten simultaneous touch inputs, ten touch-objects are enough to handle all touch inputs made throughout the execution of any project. Each of these touch objects will be used to store the input values of the touch it is currently associated with, i.e. the x and y coordinates of the touch and whether the touch is being pressed on the tablet. The big question with such a design becomes: What is the best way to distribute the touch inputs amongst the ten touch-objects?

In this second multi-touch interaction set design, which I have named the *Relative Indexing Design*, the persistent touch-objects are indexed 1 through 10 and represent the status of the touch inputs with matching indices. The touch inputs are indexed relative to the other concurrent touches in order of initiation from 1 to the number of current touches. Since touches are (as the design's name implies) indexed relatively, a touch's associated index (and, thus, associated persistent touch-object) can be subject

Figure 5-1: The "number of fingers" block.

to change. For example, a touch that was the third-oldest current touch (meaning it has an associated index of 3) can become the oldest current touch (meaning an associated index of 1) if the two older current touches are lifted. While this manner of indexing is intuitive, it has a few quirks that can lead to many unforeseen bugs in the development of seemingly simple projects.

In this chapter, I will first specify the Relative Indexing Design and then follow up by evaluating the design on the three iterations of case projects that I described in Chapter 3.

5.1 Interaction Set Specification

The Relative Indexing Design consists of nine touch blocks, only one of which is wholly original to this thesis: the "number of fingers" block (Figure 5-1). As the name implies, the "number of fingers" block produces a value ranging from *0* to *10*, representing the number of fingers on the tablet at the time of evaluation. Since most tablets can only recognize up to ten simultaneous touches, the "number of fingers" block will return the value of *10* whenever there are 10 or more touches.

The remaining eight touch blocks in the Relative Indexing Design are similar in appearance to the extended Closest Finger Design interaction set minus the "when this sprite is tapped" block. The only difference is that each of the Relative Indexing Design blocks has an added numerical parameter (see left side of Figure 5-2). This parameter is used to select the index (ranging from 1 to 10) of the desired touch of interest.[1] The parameter is a drop-down menu with values ranging from 1 to 10, but it is also a slot that can be filled with a function block (see right side of Figure 5-2).

[1]Note that in these blocks I again use the term "finger" instead of touch. Although this terminology could lead some novices into thinking that the touch indices correspond to particular fingers (e.g., finger 5 always refers to the user's left thumb), I believe it is still beneficial to use the concrete term "finger," rather than the abstract term "touch."

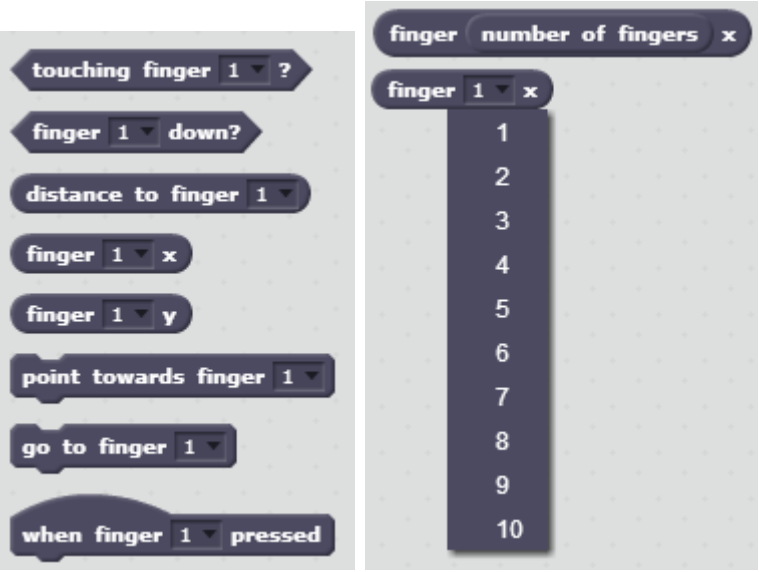

Figure 5-2: On the left are the remaining eight blocks in the Relative Indexing Design interaction set. On the right there are two examples of the "finger (*index*) x" block: one with the drop-down menu opened and the other with an inserted function block.

If the inserted function block returns a value other than an integer between 1 and 10, the value gets defaulted to 1, so as to avoid throwing errors.

Of these eight remaining touch blocks, only two, the "finger (*index*) down?" and "when finger (*index*) pressed" blocks, are not affected by the location of the touch inputs, so I will quickly explain them first. The "finger (*index*) down?" block is a simple Boolean function block that returns a value of *true* if the touch with the index of (*index*) is currently down and a value of *false* otherwise. By the nature of relative indexing, this is the equivalent of determining whether there are at least (*index*) touches currently pressed on the tablet. As a result, the "finger (*index*) down?" block is redundant, since this value can be determined using the "number of fingers" block along with some logic blocks. Even so, the "finger (*index*) down?" block is a worthwhile shortcut, because it can become tedious to determine the value repeatedly.

The "when finger (*index*) pressed" block is a simple trigger block that is closely related to the "finger (*index*) down?" function block. The "when finger (*index*) pressed" block is triggered at the moment the finger with the index of (*index*) is pressed. In other words, the "when finger (*index*) pressed" block executes its connected stack at

the moment the number of touches on the tablet increases from (*index-1*) to (*index*). Unlike its extended Closest Finger Design counterpart, which creates new threads for every triggering, the "when finger (*index*) pressed" block restarts the execution of its connected stack when it is triggered for a second time before finishing the first execution.

The final six blocks, "touching finger (*index*)?", "distance to finger (*index*)," "point towards finger (*index*)," "go to finger (*index*)," "finger (*index*) x," and "finger (*index*) y," behave the same as their Closest Finger Design counterparts except that their location of interest is the current location of the touch with the index of (*index*) instead of the sprite's closest touch. When (*index*) is less than or equal to the current number of touches (i.e., the "finger (*index*) down?" block evaluates to *true*), the location touch with the index of (*index*) is simply the location of the (*index*)th-longest-lasting touch currently on the tablet. Otherwise, when (*index*) is greater than the current number of touches (i.e., the "finger (*index*) down?" block evaluates to *false*), the location of the touch with the index of (*index*) remains at its last location. Again, prior to all touches, the fingers of all indices have the arbitrary, default location of $\{0, 0\}$.

5.2 Evaluation

Due to the functionality of the Relative Indexing Design being fairly straightforward, many of the simpler multi-touch interactions can be implemented intuitively. However, as projects get more complicated, the limitations of the Relative Indexing Design become evident.

5.2.1 Low Floor

For projects that assume a maximum of one touch on the screen at a time, the Relative Indexing Design has the capability to behave identically to the Closest Finger Design. When there is only one touch on the screen, that touch is every sprite's closest touch. Similarly, the relative index of the single touch will, by definition, always be

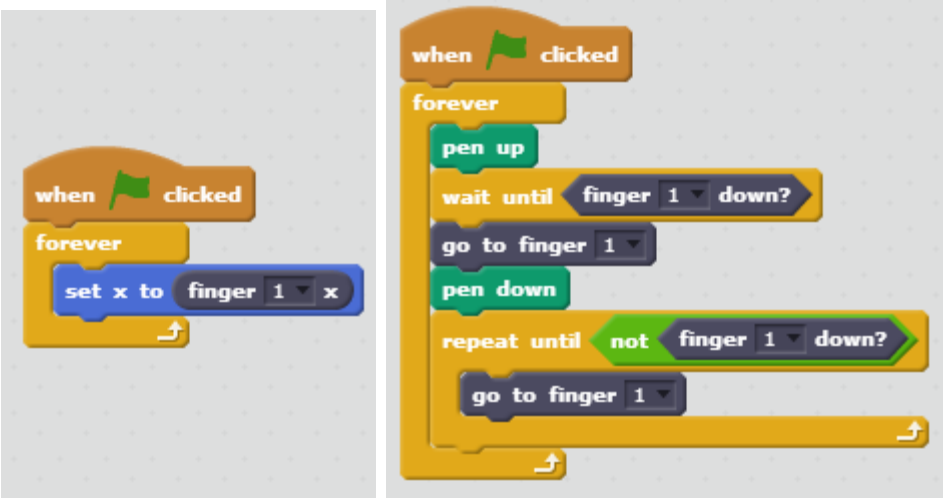

Figure 5-3: On the left is the Relative Indexing Design script for the one-player Pong game paddle controls. On the right is the script for one-finger painting.

1. Thus, both the one-player Pong game and the one-finger painting project can be implemented using the Relative Indexing Design in essentially the same way as they were implemented in the Closest Finger Design in the previous chapter (Figure 5-3).

Although these low floor case projects can be implemented using the Relative Indexing Design as easily as with the Closest Finger Design, the latter does a slightly better job of lowering the floors. The parameter in the Relative Indexing Design blocks does not provide any added functionality for single-touch projects, so it can only contribute to potential confusion.

5.2.2 Middle Height

The middle height case projects are not particularly well suited for the Relative Indexing Design. To begin with, the Relative Indexing Design does not filter the touch inputs for the closest touch (unlike the Closest Finger Design). As a result, implementing the basic two-player Pong game is slightly more laborious, even though it is still a relatively straightforward process (Figure 5-4). Since we have the assumption that there will be at most two simultaneous touches, both paddles only have to listen to the touches with indices 1 or 2. For each paddle, it does not matter whether a touch was the first or the second touch made. Thus, the paddle's script must forever

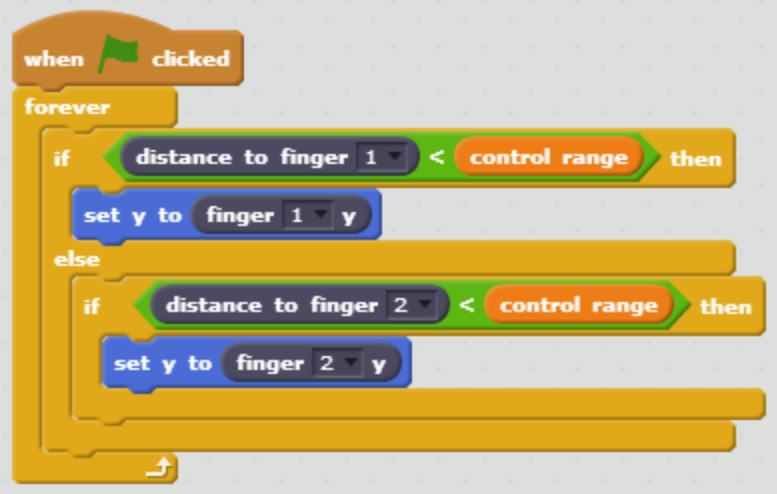

Figure 5-4: Here is a sample script for controlling the paddle in a basic two-player Pong game using the Relative Indexing Design.

repeat the process of checking both touches and follow the y-coordinate of the touch that is within the paddle's control range (if there is such a touch).

This implementation is fairly easy to follow when presented, but has a few subtleties that can be tricky for Scratchers to figure out. For example, if a Scratcher is not careful, it is possible to fall victim to the bug of having the paddle stuck "following" an already lifted touch. While these bugs can be tricky for beginners, they are relatively concrete as far as bugs go, so I am confident that Scratchers (especially with the help of the supportive Scratch community) will be able to overcome them and learn from them.

Similarly, implementing the two-finger painting project using the Relative Indexing Design also has a few subtle pitfalls. However, unlike the case for the basic two-player Pong game, the bugs that arise when implementing the two-finger painting project are very difficult to overcome. Naive intuition would suggest that each paintbrush sprite wait for and subsequently follow a different indexed touch. In other words, one paint brush would follow the first touch when it occurs and the second paint brush would follow the second touch when it occurs (left side of Figure 5-5).

This control scheme works for the most part, but there is a pernicious bug for which there is no simple solution. The bug occurs when there are two touches simultaneously

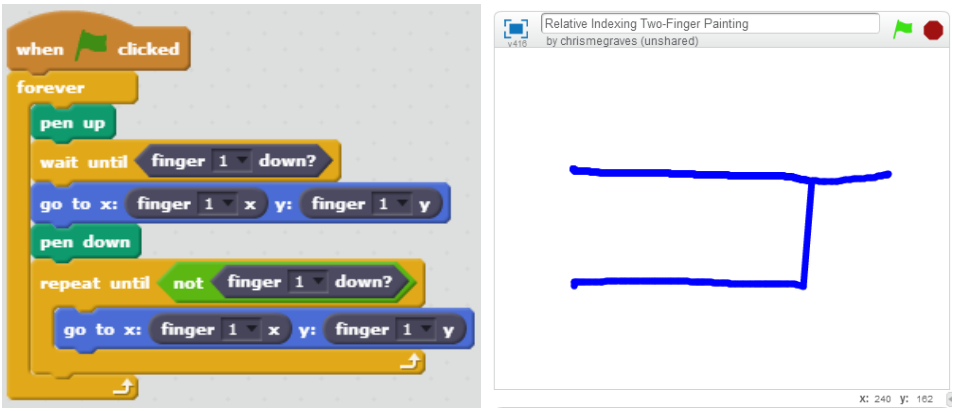

Figure 5-5: On the left is a sample naive paintbrush script using the Relative Indexing Design. On the right is the result of the bug caused by the touch index transferring.

on the tablet, and the touch with index 1 (i.e., the first touch) is lifted before the touch with index 2 (i.e., the second touch). At this moment, the paintbrush following the second created touch stops following it, because "finger (2) down?" would evaluate to *false*. Also, the paintbrush following the first touch, would then start to follow the second touch, since its index changed from 2 to 1. This transfer is fine, but the problem is that the paintbrush following the first touch does not know to lift up its pen before starting to follow the second paintbrush, leading to an unintentional line in the user's painting (e.g., right side of Figure 5-5).

There are reasonable solutions (e.g., left side of Figure 5-6) to the bug that prevent the unintentional line from appearing, but they result in small gaps in the finger painting lines (e.g., right side of Figure 5-6). Furthermore, even these imperfect solutions pose a considerable challenge to experienced Scratchers. A "perfect," practical implementation for multi-finger painting cannot be done using the Relative Indexing Design, because some data will be lost when the touch index transfers occur. This loss of data is due in part to the fact that there is no sure way to differentiate between two similar but importantly distinct events: the event in which the first touch is lifted before the second and the event in which the second touch is lifted before the first. That being said, reasonable assumptions can be made by comparing touch locations, as was the case for the Closest Finger Design.

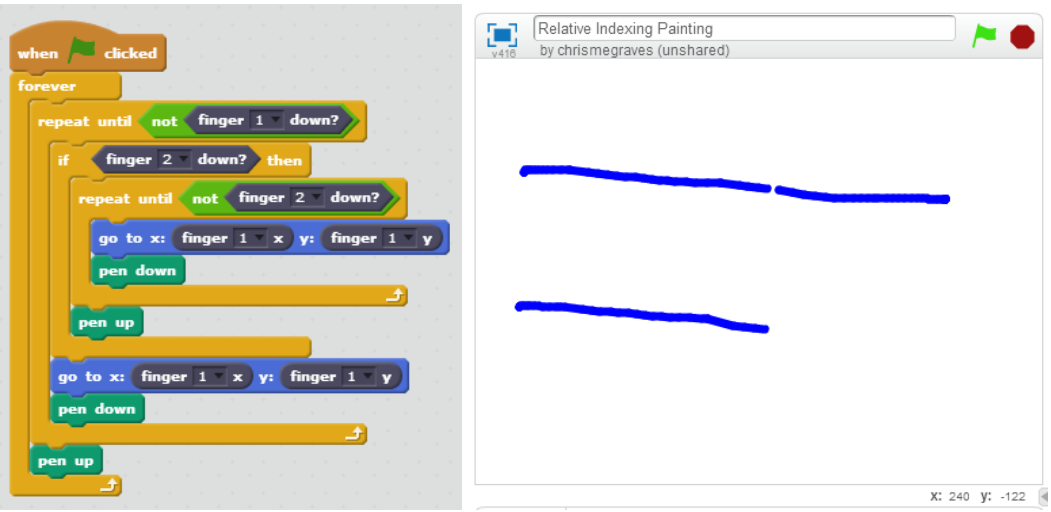

Figure 5-6: On the left is a sample "wary" script for the paintbrush using the Relative Indexing Design. On the right is the result of the bug that still exists in the "wary" script.

5.2.3 High Ceiling

Since the Relative Indexing Design makes it impossible to practically implement a "perfect" two-finger painting project, the same holds for the ten-finger painting project. Also, as the reader might imagine, the added logic needed to minimize (but not eliminate) the bugs in the two-finger project becomes much more convoluted when support for more fingers is added, because all touch index transfers must be accounted for. That being said, the level of complexity for the needed, additional logic is in line with the expected amount of difficulty for high ceiling projects. Although assembling the blocks would be tedious, the logic behind handling the transfers follows a pattern that can be repeated.

Similarly, the advanced two-player Pong game is quite difficult to implement using the Relative Indexing Design, but it can be done, and the amount of work and expertise needed is appropriate for a high ceiling project. Unlike the ten-finger painting project, however, the advanced two-player Pong game can be implemented exactly as specified with the Relative Indexing Design. The script for the advanced two-player Pong game paddle controls can be a little complicated (Figure 5-7), but the logic is fairly straightforward.

Figure 5-7: Here is a sample script for controlling the paddle in an advanced two-player Pong game using the Relative Indexing Design.

The controls can be implemented using one "forever loop" to control the movement and four "when finger (*index*) pressed" stacks to wait for bump-initiating taps. The "forever loop" would first check if the paddle is currently following a finger. If the paddle is not following a finger, its script will query the touches with indices 1 through 4 (since we are assuming no more than four simultaneous touches) and decide to follow the first finger index it queries that is both down and within the paddle's control range. After deciding to follow a finger index, the paddle will follow that finger index until it is either lifted or moved out of range.

The four "when finger (*index*) pressed" blocks will watch the finger indices 1 through 4. Their connected stacks will check for three criteria and, if they are met, will initiate a bump. The first criterion is that the paddle is currently following a touch. The second is that the touch being followed is not the touch that triggered the execution of the stack. Finally, the last criterion is that the touch that triggered the stack is within the paddle's control range.

As evidenced by the high ceiling case projects, the Relative Indexing Design does not make implementing complex projects particularly easy. However, the design does make their implementations possible (or at least almost possible, as is the case for the multi-finger painting projects). The intuitions for these implementations are

almost always straightforward, but the actual assembly of the implementations can be tedious. Furthermore, there are subtle quirks with the Relative Indexing Design that are the basis for some tricky (albeit oftentimes surmountable) bugs.

Chapter 6

Absolute Indexing Design

Without a doubt, the multi-finger painting case projects have proven to be the most troublesome to implement using both of the two previously presented designs. It is impossible to elegantly implement a multi-finger painting project using the Closest Finger Design, because the design does not offer a method to differentiate between individual touches other than through the comparison of touch locations. The Relative Indexing Design suffers from a similar inability. Although the Relative Indexing Design makes it easy to distinguish all of the present touches (using their relative indices), the design does not enable scripts to determine when touches switch indices. As a result, it is difficult to program paintbrushes to follow specific touches, since the touches' associated indices are subject to change. If every touch were given a unique index that it kept from its creation to its end (i.e., an ID number), then it would not be difficult to distribute touches amongst the paintbrush sprites so that every touch is followed by exactly one paintbrush.

This concept, assigning each created touch a unique, permanent number, is the main premise behind the *Absolute Indexing Design*, the third and final multi-interaction set design. Touches in the Absolute Indexing Design are indexed in order of initiation relative to all touches that have been made since the inception of the project. As a result, a touch's associated index can never change in the Absolute Indexing Design, because the nth touch pressed in the execution of a project will forever be the nth touch pressed.

While the Absolute Indexing Design can facilitate the implementation of certain complex multi-touch interactions, it can make implementing simpler single-touch interactions more difficult. In this chapter, I will again first specify the design and afterwards evaluate it using the three iterations of case projects described in Chapter 3.

6.1 Interaction Set Specification

The Absolute Indexing Design consists of nine blocks (Figure 6-1), only two of which are wholly original. In contrast to the previous chapter, I will specify the familiar blocks before presenting the newer ones.

The seven familiar blocks in the Absolute Indexing Design each have counterparts in the Relative Indexing Design, but with two main differences. The first difference between these blocks and their Relative Indexing Design counterparts is that their parameters are text inputs rather than drop-down menus. Since the touch indices are not bounded by 10 in the Absolute Indexing Design, a drop-down menu would not make sense. Also, these parameters are best thought of as (*ID*)s instead of (*index*)es. The second difference is that the term "finger" is replaced with "touch" in the block labels. Although the use of the term "finger" was not perfectly suitable in the Relative Indexing Design either, the term "finger" makes no sense at all when using absolute indexing. For example, the term "finger (73)" implies that the user has at least 73 fingers, whereas "touch (73)" accurately implies the 73rd touch pressed on the tablet.

Thus, the seven familiar blocks are named: "touch (*ID*) down?", "touching touch (*ID*)?", "distance to touch (*ID*)," "point towards touch (*ID*)," "go to touch (*ID*)," "touch (*ID*) x," and "touch (*ID*) y." Functionally, these seven Absolute Indexing Design blocks work in a manner similar to their Relative Indexing Design counterparts, just with different touch indexing. For example, the "touch (*ID*) down?" block reports the value of *true* if the (*ID*)th touch in the project's execution has been created and is still touching the tablet and otherwise reports the value of *false*. Similarly, the remaining six familiar blocks all correspond to the location of the (*ID*)th touch in the

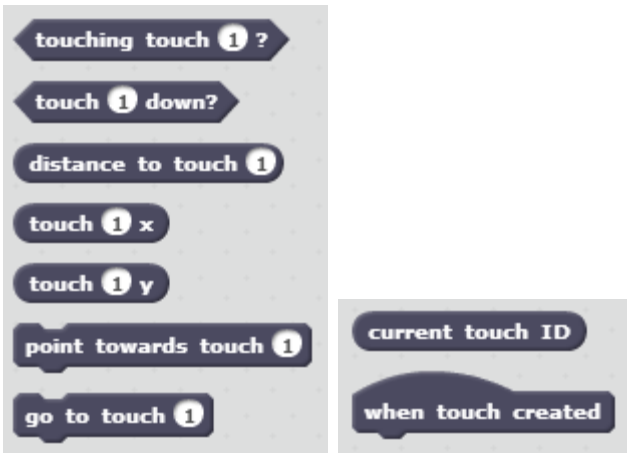

Figure 6-1: The seven familiar blocks in the Absolute Index Designs are on the left, while the two new blocks are on the right.

project's execution. If the (ID)th touch has yet to be made, its location is considered to be $\{0, 0\}$. Also, if the (ID)th touch has already been both made and lifted, its location is considered to be its last touched location. If the inserted value of (ID) is anything other than a positive integer, the location of interest is always defaulted to $\{0, 0\}$.

Now that the familiar blocks have been specified, I will introduce the two original blocks. The first of the entirely new blocks is a function block, labeled "current touch ID." Simply put, the "current touch ID" block reports the unique ID of the most recently made touch. When touches have yet to be made, the "current touch ID" block reports the value 0, since the first touch in the execution of a project has an ID of 1. An important aspect of the behavior of the "current touch ID" is that it increases by at most 1 per execution iteration of the Scratch programming environment. In other words, if a loop in Scratch is constantly checking the value of the "current touch ID" block at each iteration (without having inner loops or "wait" blocks), the value will not jump from x to $x+2$ without being valued as $x+1$ first, even if two touches are made simultaneously. Thus, it is possible to watch the "current touch ID" block without having to worry about missing a touch.

The second new block is a trigger block labeled, "when touch created." In many ways the "when touch created" block is analogous to the "when finger pressed" block

from the extended version of the Closest Finger Design and the "when finger (*index*) pressed" block from the Relative Indexing Design. However, its behavior is very specific and important to the Absolute Indexing Design. Like the "when finger pressed" block, the "when touch created" block is triggered by the initiation of every touch. However, unlike the "when finger pressed" block, the "when touch created" block does not create a new thread for every touch. Instead, the "when touch created" block handles simultaneous triggerings in a way similar to the "current touch ID" block. If, for example, two simultaneous touches are made, the "when touch created" block would first execute its first stack as far as it can in one Scratch execution iteration (stopping only at a "wait" or the end of a loop), before restarting the stack execution in the next Scratch execution iteration. Therefore, if the first block in a "when touch created" block's connected stack contains a reference to the "current touch ID" block, the "current touch ID" block would have the ID of the (arbitrarily chosen) first touch during the first execution and the ID of the second touch during the second execution.

Like many aspects of the Scratch programming language, the "current touch ID" and "when touch created" blocks are designed specifically so that Scratchers who are not concerned with concurrency bugs do not generally need to be concerned, and Scratchers who are concerned can take necessary precautions to avoid them.

6.2 Evaluation

Although Absolute Indexing Design can be used to elegantly implement complex multi-touch interactions, it can be difficult for beginners to understand. Indexing touches by their order of creation relative to the entirety of the project is an abstract concept that can be difficult for Scratchers to gain an intuition for. However, once such an intuition is acquired, Scratchers can use the Absolute Indexing Design to efficiently and naturally create as complex multi-touch interactions as they can imagine.

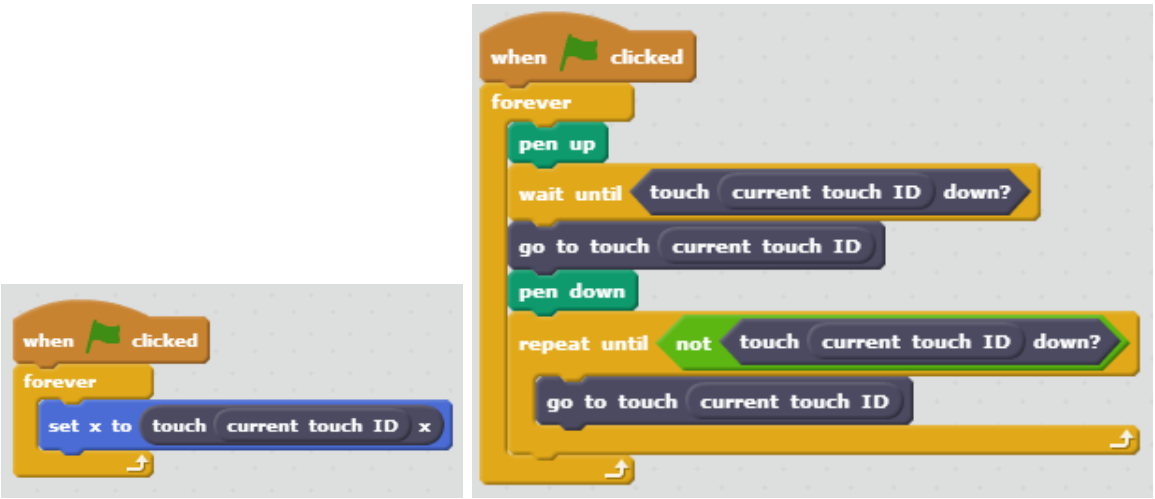

Figure 6-2: A sample Absolute Indexing Design script for the paddle controls in a one-player Pong game is on the left. On the right is a sample paintbrush script for one-finger painting.

6.2.1 Low Floor

By using the "current touch ID" block to get the ID of the latest touch, both of the low floor case projects can be implemented in essentially the same way as was done using the previous two designs (Figure 6-2). Since we are assuming there is at most one touch on the screen at any time, the "current touch ID" will always refer to the only touch on the tablet (if there is one).

Although these implementations look nearly as simple as those done with the Closest Finger Design, it might not be obvious to novice Scratchers that they can get access to the latest touch by inserting the "current touch ID" block into the other touch blocks' (*ID*) parameters. Blocks like "touch (*ID*) x" by themselves are neither tinkerable nor explorable, so it is difficult for Scratchers to learn how to use them without outside instruction. Unless the given (*ID*) value happens to be an ID of one of the current touches, many of the Absolute Indexing Design blocks will appear unresponsive to touches.

One idea, to make the Absolute Indexing Design more explorable, would be for the block palette to feature "touch (*ID*) x" and "touch (*ID*) y" blocks with the "current touch ID" block already inserted in their (*ID*) parameter. While this would

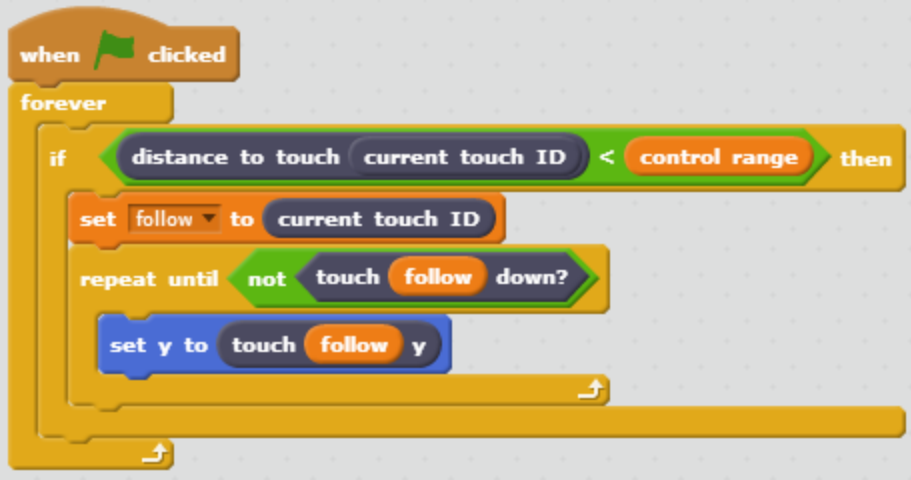

Figure 6-3: Here is a sample script for controlling the paddle in a basic two-player Pong game using the Absolute Indexing Design.

definitely increase the explorability of the blocks, it could cause some confusion, because preassembled blocks have never before been featured in Scratch.

6.2.2 Middle Height

Once Scratchers get over the barrier of learning the intuition behind the absolute indexing logic, they can, without too much difficulty, figure out how to implement both of the middle height case projects.

To begin, the controls for the basic two-player Pong game can be implemented by having both paddles constantly repeating a simple process (Figure 6-3). First, each paddle's script can check to see if the touch with the "current touch ID" is within their paddle's control range and store the ID in a variable if it is. Then, with the stored touch ID, the script can set the paddle to follow the touch with the stored ID until it is lifted.

Although this implementation of the basic two-player Pong game is complicated by the necessary use of a variable, experienced Scratchers generally have a basic understanding of how to use variables. As a result, barring the difficulty of understanding how the absolute indexing works, the complexity of this implementation is appropriate for a middle height case.

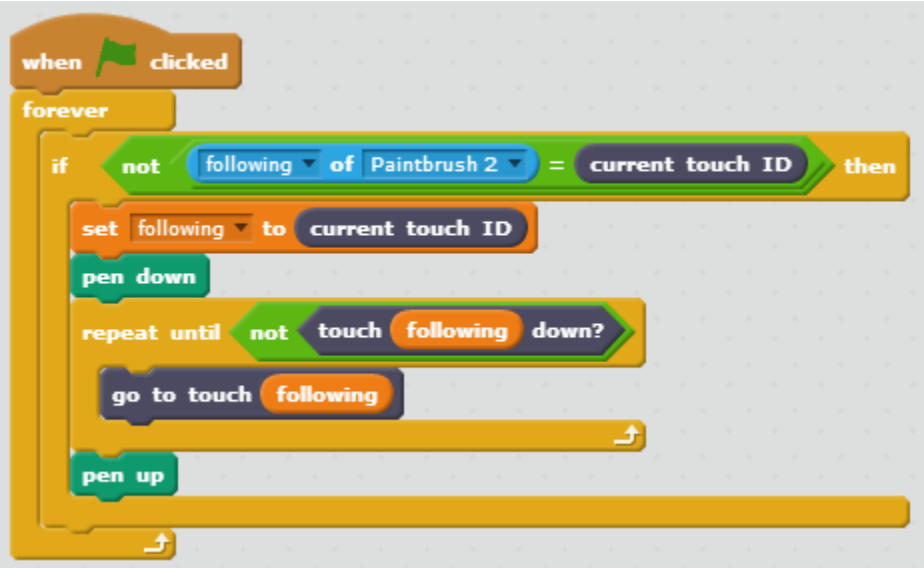

Figure 6-4: Here is a sample script for controlling one of the paintbrushes in a two-finger painting project using the Absolute Indexing Design.

Like the basic two-player Pong game, the two-finger painting project can also be implemented using the Absolute Indexing Design by programming both of the paintbrushes to forever repeat a simple process while keeping a variable in which they store the ID of the touch they currently are following (Figure 6-4). To begin each iteration of the repeated process, each paintbrush first checks to see if the touch with the "current touch ID" is down and not being followed by the other paintbrush. If both criteria are met, then the paintbrush sets its variable to the current touch ID (thus claiming it) and follows the touch with its pen down until the touch is lifted.

Note that this implementation does rely on some of the specifics regarding Scratch's thread-switching protocol. However, the naive assumption that there will not be any race conditions in this case turns out to be true. As a result, this implementation is not too complicated for a middle height project. It is particularly noteworthy that two-finger painting can be implemented perfectly using the Absolute Indexing Design with an implementation that is no more complex as the imperfect implementations that could be created using the other two designs.

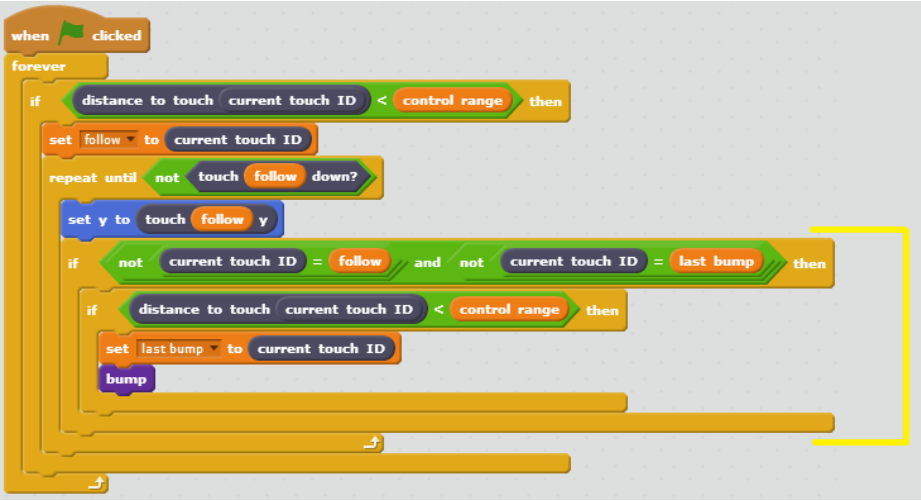

Figure 6-5: Here is a sample script for controlling the paddle in an advanced two-player Pong game using the Absolute Indexing Design. The subroutine that is added to the basic two-player Pong game is highlighted.

6.2.3 High Ceiling

When using the Absolute Indexing Design, the gap in complexity between the middle height and high ceiling case projects disappears. Once Scratchers have gained an intuition for how to use the Absolute Indexing Design to make multi-touch interactions with moderate complexity (which, granted, can be an arduous task), it becomes a natural and iterative process for them to discover how to make more complex ones.

For example, the progression from the basic two-player Pong game to the advanced two-player Pong game is a particularly natural one when using the Absolute Indexing Design. To add the new bump interaction, each paddle's script can be augmented with an additional variable and a simple subroutine to be run when the paddle is following a particular touch (Figure 6-5). The additional variable is used to store the ID of the touch that triggered the last bump, so that the same touch cannot trigger multiple bumps. Thus, the subroutine just checks on the current touch to see if it is different from both the touch the paddle is following and the touch that triggered the last bump. If both criteria are met, the subroutine makes a final check to see that the current touch is within the paddle's control range and initiates a bump if that is the case.

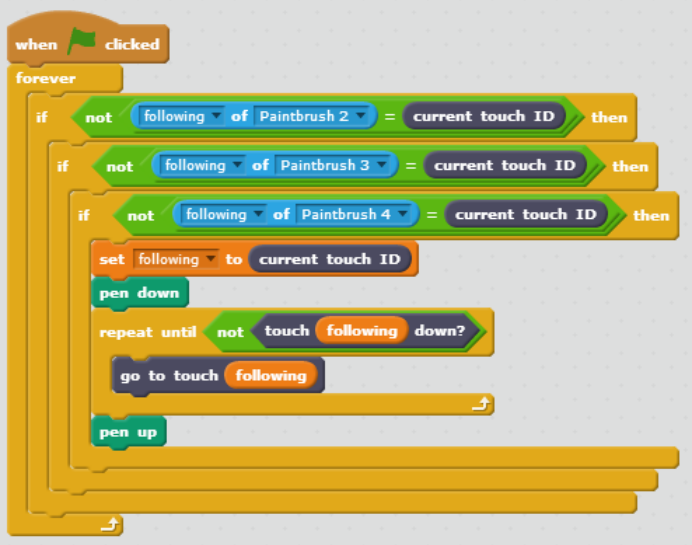

Figure 6-6: Here is a sample script for a paintbrush sprite in a four-finger painting project using the Absolute Indexing Design, but without using clones. To add more fingers, one need only add the additional paintbrush sprites and if statements.

Similarly, the implementation for two-finger painting using the Absolute Indexing Design can easily be upgraded to ten-finger painting by simply adding more paintbrush sprites and more checks in each of their scripts to make sure that no paintbrush follows an already followed touch (Figure 6-6). While this is an extremely simple augmentation, the script copying can become tedious.

Hence, an experienced Scratcher may be tempted to use Scratch's cloning feature to make the implementation more elegant. Using the "when touch created" block, a Scratcher can create a new paintbrush clone specifically to follow each new touch. The paintbrush clone would then follow its assigned touch until the touch is lifted, at which point the clone selflessly kills its own script to preserve computational resources. While the implementation using cloning is more elegant and uses fewer blocks, both implementations have the same functionality.

In comparison to the other two designs I have presented, the Absolute Indexing Design has a steep learning curve. It is very unlikely that someone could quickly figure out how the design's blocks work just by playing around with them. However, as multi-touch interactions become more complex, it becomes easier to implement them using the Absolute Indexing Design than with the other two designs.

Chapter 7

Conclusion

In this chapter I will first present my opinions

7.1 Closing Remarks

7.2 Future Work

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.ht

[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.