Constrained Sets: The Effects of Multi-Layered Environments in Learning App Inventor

by

Lynda Tang

S.B., Massachusetts Institute of Technology(2017)

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

Master of Science in Computer Science and Engineering

at the

MASSACHUSETTS INSTITUTE OF TECHNOLOGY

June 2018

Author . . . .
Department of Electrical Engineering and Computer Science
May 11, 2018

Certified by . . . .
Harold Abelson
Class of 1922 Professor of Computer Science and Engineering
Thesis Supervisor

Accepted by . . . .
Katrina LaCurts
Chairman, Department Committee on Graduate Theses

Constrained Sets: The Effects of Multi-Layered Environments in Learning App Inventor

by

Lynda Tang

Submitted to the Department of Electrical Engineering and Computer Science
on May 11, 2018, in partial fulfillment of the
requirements for the degree of
Master of Science in Computer Science and Engineering

Abstract

MIT App Inventor is a mobile application development platform that seeks to democratize the construction of mobile apps by making app development accessible to people with little to no experience with script-based programming. It uses block-based programming to introduce and teach programming concepts to its users. Users drag and drop functional and visual components onto their planned app in the screen editor, and construct the logic behind those components by using blocks in the block editor. In this thesis, we design and implement Constrained Sets, a system that allows instructors and developers to allow access to only a subset of App Inventor functionality by hiding component and block access. This system allows for the construction of multi-layered interfaces, which we then use to conduct an experiment that explores how novice App Inventor users learn App Inventor in different interface environments. Furthermore, we discuss and test the possibility of using a React based implementation of the App Inventor designer, and what implications that may have on creating more flexible user interfaces.

Thesis Supervisor: Harold Abelson
Title: Class of 1922 Professor of Computer Science and Engineering

Acknowledgments

I'd like to thank all the people who've supported me directly and indirectly in the past year throughout the thesis process. From the MIT App Inventor team, I learned not only about computer science and research techniques, but also life lessons and what it means to be a person who makes a positive impact on the world and on the people around them.

First of all, I'd like to thank Hal Abelson for giving me this opportunity to work with App Inventor and opening my eyes to what it means to be a researcher. I'd also like to thank my current mentors Evan Patton and Mike Tissenbaum for taking time out of their days and going above and beyond in helping make my study a success, even when both of them were overloaded with work. You two are both role models that I aspire to become.

Thank you to everyone on the MIT App Inventor team for the small things everyone did to help me out: Mark Sherman, for helping me generate the accounts I needed for my study, Farzeen Harunani, for giving me emotional support, Hedge Nichols, for taking my panic calls in the middle of the night, Marsha Gordon, for always lending an ear, and Hercules and Kaidan, for always lending a waggy tail and a furry paw. Additionally, thank you to all the students and staff members that tested the system and the curriculum.

I also want to thank Elaine Mou and all the kids that participated in my research. I couldn't have done it without you guys and I will miss your enthusiasm every Saturday.

Lastly, I'd like to thank my former mentor Paul Medlock-Walton, for guiding me through most of the technical challenges in the thesis and making sure that I was well taken care of after his departure.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

User interfaces increase accessibility by providing a visual representation that abstracts the complex systems underneath. For new users, the interface is the first thing they see and interact with when using a piece of software. Therefore, designers and developers should take special care to create an interface that easily onboards new users and primes them for the other features the software offers. This thesis looks at the how integrating a limited interface with tutorials affect learning App Inventor.

App Inventor is a platform that enables users to use block based programming to create Android applications. In this thesis, we create Constrained Sets, which is a system that diversifies and simplifies App Inventor's designer interface by allowing a presentation of only a subset of App Inventor's full functionality. It can be used to limit the environment a new user starts off in to create more beginner-friendly user experiences. Using the aforementioned system, instructors can specify a Constrained Set and display the Constrained Set in the App Inventor GUI. In addition to creating the system, we also use Constrained Sets to investigate how different environments affect how novice users learn and use App Inventor. We discovered that although limited interfaces had little effect on user confidence and their perceived understanding of App Inventor, students who started out in the constrained environment explored a larger range of blocks when they were given access to all of App Inventor's functionality.

1.1 MIT App Inventor

MIT App Inventor (2017) is a development environment for Android applications for smartphones and tablet devices. It combines a visual designer with Blockly, a block-based programming language, and is used monthly by over 1.1 million users around the world, who have created a total of over 24 million apps. App Inventor democratizes app creation by abstracting the complex parts of programming languages behind blocks. Unlike using script-based programming languages, users do not have to worry about using the correct syntax and can focus on learning computational models and paradigms.

A majority of new users learn App Inventor in an informal educational setting through online tutorials and walkthroughs. The tutorials help new users by scaffolding the app development process, familiarizing them with the software, and introducing them to new components and blocks. However, the tutorials only cover a fraction of what App Inventor offers, and the rest is left for the user to figure out and explore.

Applications are created in App Inventor via its project designer. The project designer consists of two parts: 1) the screen editor and 2) the blocks editor. The screen editor determines which components, viewable displays, and functions are on the app, and the blocks editor controls the logic behind the app and its components.

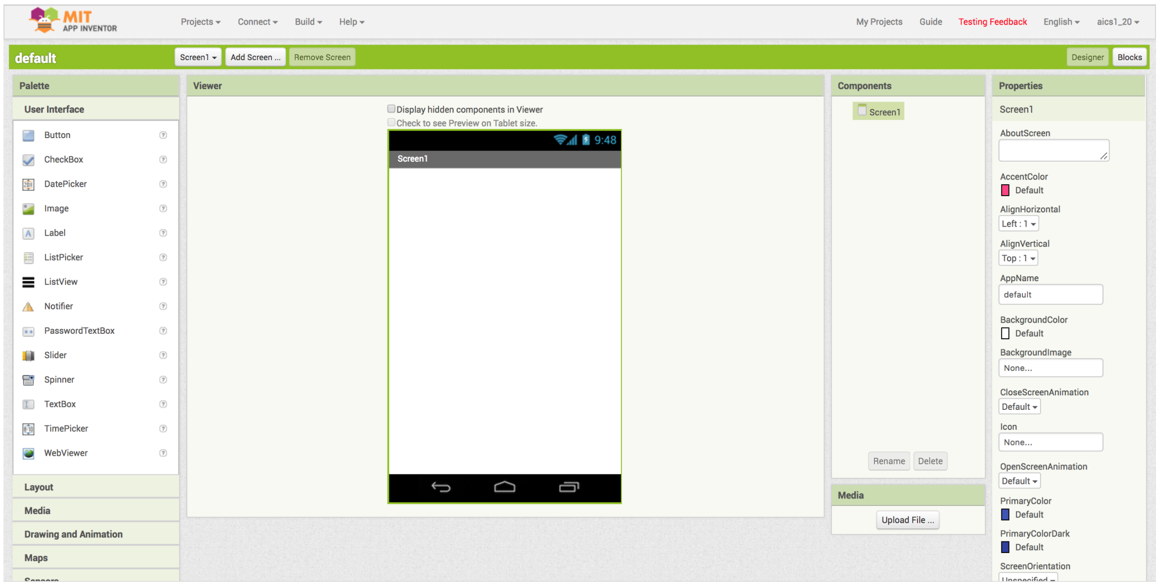
1.1.1 Components

The components of App Inventor are the visual and functional parts that form the modules of the application. They are shown in the Palette panel in the screen editor. Components are added to the application by dragging and dropping them onto the Viewer, and their properties can be modified in the Properties Panel on the screen editor.

1.1.2 Blocks

Blocks control the logic behind the components. There are two types of blocks: built-in blocks and component-specific blocks. Built-in blocks are available in every

Figure 1-1: App Inventor's Screen Editor



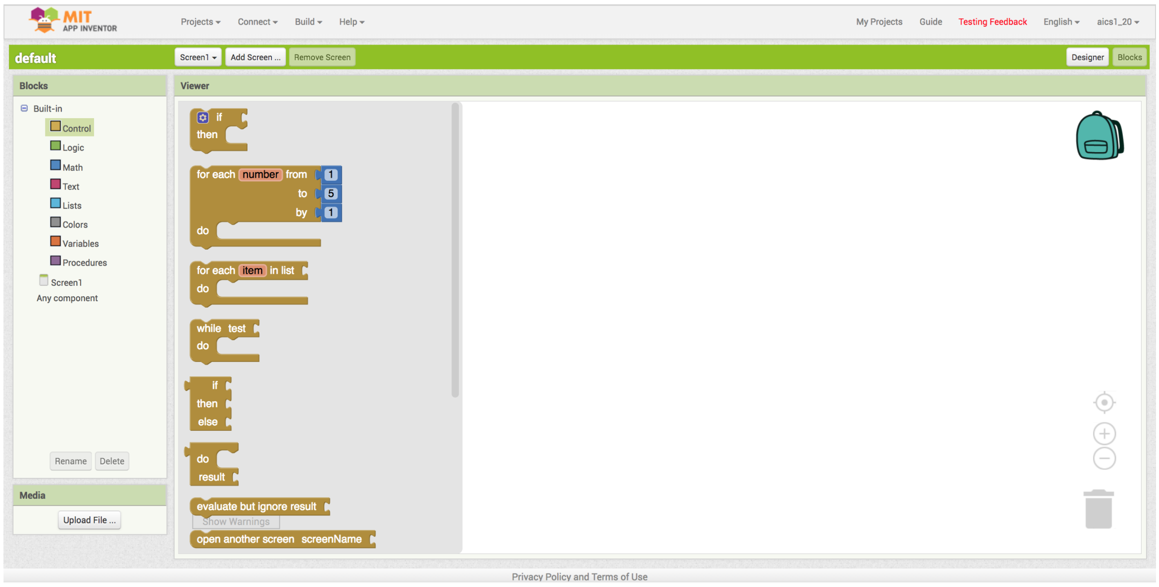
The App Inventor screen editor. Here, users can drag and drop visual components (e.g.:Button, Checkbox) as well as functional components (e.g.: Accelerometer, Timer) onto their Viewer to use on the screen.

project; they handle basic computational logic, such as if-else statements, and the manipulation of basic objects, such as strings and numbers.

Component-specific blocks are blocks that handle calling component methods and setting or getting variables on a specific component object instance. When a component is added to the app, the blocks drawer will modify itself to contain that component's corresponding blocks. One example of a component-specific block is the "When Button Click" block, which detects when a component is clicked. Users first drag a component, such as a Button, into the Viewer, and then use the "When Button Click" block to detect when that Button is clicked to execute the subsequent instructions. Component specific blocks fall into three major types: event, method, and get/set blocks. Event blocks detect when an event occurs, method blocks modify the component, and get/set blocks retrieve and change the component properties.

With a cumulative total of over a thousand blocks and components, App Inventor offers users many tools to build a myriad of apps, but that sheer amount may seem intimidating to the new user and can be frustrating to use(Xie, 2016; Colter, 2016).

Figure 1-2: App Inventor's Blocks Editor



The App Inventor blocks editor. Users use blocks to form the logic behind the app.

We introduce the concept of Constrained Sets to help limit the large search space provided by App Inventor's blocks and components.

1.2 Motivation Behind Constrained Sets

Currently, users are always shown an interface that gives them access to all of App Inventor's functionality; while this gives novice users an unlimited sandbox for them to explore and test, it may also increase their cognitive load and impede their ability to learn App Inventor. In a software like App Inventor, when user proficiency of the program is linked with their understanding of app development and computational thinking, does learning in a limited environment affect their ability to understand and use the system later on? We use Constrained Sets to try to answer this question.

1.3 Constrained Sets Use Cases

Although our experiment only investigates the effects of using Constrained Sets in coordination with tutorials for novice users, Constrained Sets can also be utilized to create environments that give experienced users easier access to the tools they frequently use. We propose some more use cases for the system below:

1.3.1 Tutorial Integration

Constrained Sets can be integrated into App Inventor tutorials. This thesis investigates the effect of tutorial integration by looking at what happens when kids are given only the exact components and blocks they need to complete each tutorial and the tutorial's explorations. Using the set generator (see Chapter 3), instructors such as the App Inventor staff and other educators can build custom Constrained Sets for each tutorial.

1.3.2 Limited Exploration Environments

Constrained Sets can be used to create enclosed exploration environments. Instructors can create specific sets for their students that expose only the blocks and components they want their students to use and play around with.

1.3.3 Customized User Interfaces

Benjamin Xie's research on the Progression of Computational Thinking Skills in App Inventor Users (2016) showed that users, even those who are experienced, used on average less than twenty different types of blocks. This suggests that perhaps displaying all of the App Inventor blocks is unnecessary, even for long term users. However, using Constrained Sets in this way would mean that users would need to learn how to create their own Constrained Sets.

1.3.4 Project-Type Specific Interfaces

Constrained Sets can also be used to make project-type specific interfaces. Industry software, such as Blender and Adobe Suite's products, often have different pre-defined layouts that are based on user goals. Blender (2007), for example, uses a selection of default screens to let the user choose which environment is most conducive, or mostly closely aligns to their project type.

1.4 Thesis Overview

This thesis details the process of creating Constrained Sets and investigating the effects of Constrained Sets on computational learning for beginner App Inventor users. Chapter 2 goes further into the background theories behind user interface design. It also looks at other related work with changing the interface on block-based programming languages and learning environments. Chapter 3 discusses the design decisions behind Constrained Sets and how the system was implemented. In chapter 4 we discuss the design and details of the research experiment, and analyze the results of the experiment in chapter 5. Chapter 6 discusses the implications of the experiment, and what it means for App Inventor's future.

Chapter 2

Related Work

This chapter looks at the user interface and educational theories that informed and inspired the creation of Constrained Sets. It also provides examples of subsets and multi-layered interfaces utilized in other learning environments.

2.1 User Interface Theories

As computers moved from largely esoteric machines operated by a select few to a common household item, the importance of interfaces increased. An increase in usability for a program can result in financial savings, decreased task times, and an increase in the number of good customer reviews (Nielsen, 1993). This explains why a larger percentage of software code is devoted to its interface than before. This section will explain cognitive load, one of the challenges of interface usability, and how multi-layered interfaces can minimize it.

2.1.1 Cognitive Load

The idea of cognitive load was first proposed in a paper by George Armitage Miller (1956), where he postulated that our working memory contains around seven chunks of information. Giving it more, he further states, results in a phenomenon known as information overload, and can have an adverse effect on the decisions and judgments that people make.

Sweller (1988) then took the idea of a limited amount of working memory and applied it to an educational context, stating that learners have more trouble acquiring schemas if the learning task demands a high amount of cognitive load. In particular, learners who expend more effort in trying to solve a problem will find that their abilities for schema acquisition may be inhibited (Sweller, 1988). When we frame this idea in App Inventor terms, solving the problem is building the apps, and schema acquisition is understanding the computational logic behind them. Therefore, it is not a reach to say that to improve computational thinking in our users, we should, at least initially, reduce novice users' cognitive load.

Cognitive load appears in three different types: intrinsic, extraneous, and germane. Intrinsic and germane cognitive loads refer to, respectively, the difficulty of the topic and the amount of effort it takes to understand the underlying models that the topic or problem is based upon. Extraneous cognitive load is the cognitive load that is associated with the presentation of the topic to the learner, and the one we seek to minimize using interface design, especially during initial instruction (Chandler and Sweller, 1991).

One of Nielsen's (1993) ideas to reduce extraneous cognitive load in interfaces is to avoid visual clutter, and research has shown that having cluttered interfaces not only causes confusion in beginner users, but also negatively affects the speed of experts. By only showing the user what is relevant to their needs, we remove unnecessary distractions and possible sources of error.

Cognitive load theory would suggest that simplification of an interface is the answer to reducing cognitive load, but when we apply that logic to App Inventor, which has a broad user-base, we run into some issues. Specifically, how do we know what is relevant to a user, and how to we create an interface that caters to both beginners and experienced users?

2.1.2 Multi-Layered Interfaces

One solution we can employ in answering this universal usability conundrum is the concept of multi-layered interfaces. Rather than have everybody use the same interface, we can create a tiered system that starts beginning users off at layer one and advances them to higher layers as they become more competent (Shneiderman, 2003). Multi-layered interfaces can appear in many different variations, and Constrained Sets enable App Inventor developers and instructors to easily create the variations.

Figure 2-1: Multi-Layered Interface Variations

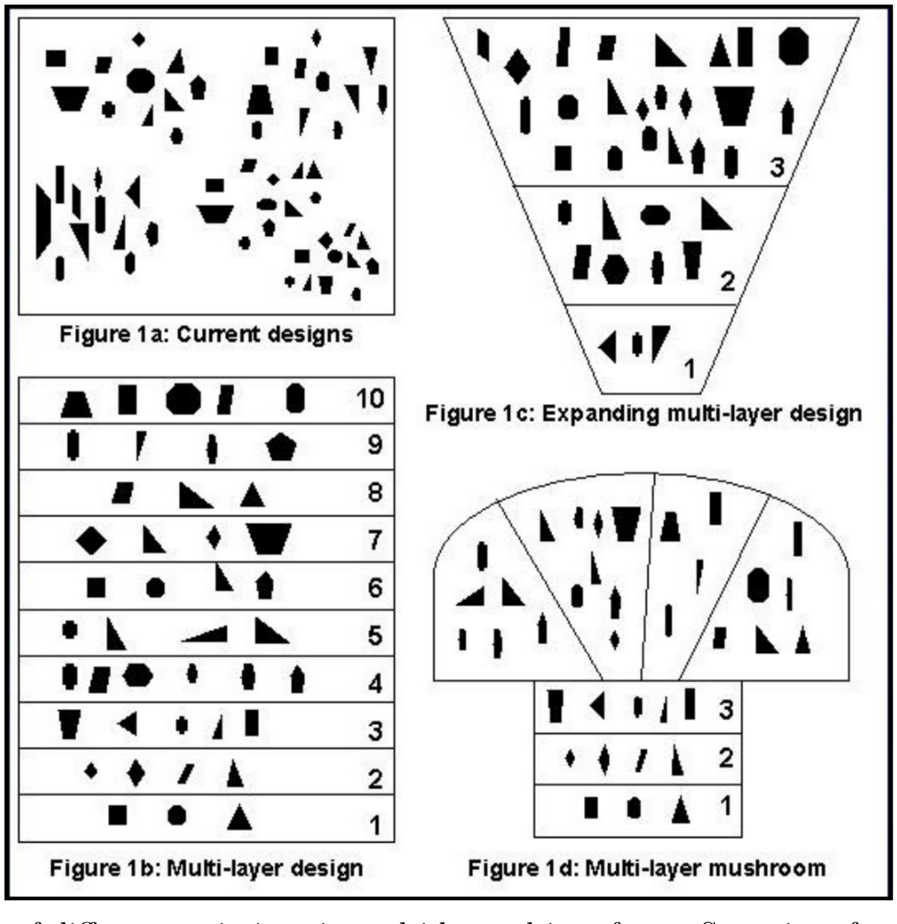


Figure 1a: Current designs
10
9
8
7
6
5
4
3
2
1
Figure 1b: Multi-layer design
3
2
1
Figure 1c: Expanding multi-layer design
3
2
1
Figure 1d: Multi-layer mushroom

Examples of different variations in multi-layered interfaces. Some interfaces such as the standard multi-layer design (1b) provide different functions for different layers, others, like the expanding multi-layer design (1c), use a compounding model, where functions in the previous layer are available in the later layers (Shneiderman, 2003).

A large challenge in using multi-layered interfaces is determining how the layers should be structured and what variation they should appear in. We will discuss this further in Future Work.

2.2 Education Theories

2.2.1 Zone of Proximal Development and Scaffolding

The Zone of Proximal Development (ZPD) is a concept developed by Lev Vygotsky that represents the space of tasks a learner cannot do on their own, but can do with some guidance from an instructor (Daniels, 1996). The classic example of ZPD in action is the process of how children learn to speak. When children learn to speak, the adults around them provide feedback and guidance through reactions and responses, which in turn helps the child develop more language skills until help from the adults is no longer needed. As learners expand their knowledge, their zones of proximal development expand and learners eventually are able to do tasks in the zone independently.

Instructional scaffolding is a way to provide the guidance that learners need to help them accomplish tasks in the zone of proximal development. As students gradually master the tasks, the scaffolds fade away. Instead of having the instructor guide the student from task to task, they are only there to provide small nudges in the right direction; the driver of a student's learning is the student themself. This type of instruction also allows for a student to exercise their independence and creates an environment where students can feel safe to experiment and fail.

The experiment for this thesis uses Constrained Sets as a form of scaffolding for new learners of App Inventor. The explorations for the tutorials created for the experiment switches the teaching model from one of hand-holding to one where students are only given suggestions and must figure out how to implement the rest. The limited subset that they are given serves as the scaffolding for their exploration.

2.3 Prior Work

This idea of using scaffolding through interfaces has been used before in different projects. This section details a few of them.

2.3.1 Gameblox Flexidor and StarLogo Subsets

Gameblox (2014) and StarLogo (2010) are two projects from the MIT STEP Lab that both use a block-based programming to teach their respective users programming concepts through the development of software. Gameblox is a game editor that allows its users to create video games; users can learn game development skills and concepts such as design and prototyping without previous programming knowledge. StarLogo is also a game editor, but it focuses on teaching its users agent-based models and decentralized systems, letting them create simulations with thousands of agents.

Flexidor is a similar implementation of Constrained Sets in Gameblox. It provides a restricted view of the Gameblox editor and allows users to create customized editors and block subsets. It was tested on eighteen MIT undergraduate and graduate students and the results showed that the users completed tasks faster and with less errors when using the Flexidor versus the full editor, although the groups showed little difference in their self-reported understanding and learning scores (Du, 2015). This could be due to the test group being already familiar with programming concepts.

StarLogo is another development environment that uses subsets. It allows instructors and users to create their own block sets that consist of activity based or frequently used blocks. After the subsets are created, users can save the subsets for future use, allowing for a personalized environment(StarLogo, 2010).

2.3.2 Quizly

Quizly is an assessment platform that allows users to test themselves with small App Inventor block-based tasks such as "Set the text color of Canvas to magenta" and presents the user with a limited amount of block drawers for them to choose blocks from (Maiorana et al., 2015). Instructors can use Quizly's Quiz Maker to create questions and block drawers for their students, and students are given automatic feedback when they make a submission. Quizly uses a similar approach to Constrained Sets in that it allows the instructors to limit the blocks the students are exposed to to only contain the ones needed. Constrained Sets, however, places the blocks in App Inventor's app building context instead of removing the assessment from the environment.

2.3.3 Reduced GUI in Geometry Teaching

Researchers have also looked at how multi-layered interfaces affect learning in other educational software. Previous research on iGeom, an interactive geometry program, has shown that students felt frustrated with the iGeom interface due to the high number of features it has(Borges et al., 2016). The researchers developed two interfaces for iGeom: a complete interface and a reduced interface. The complete interface displayed all of the features of the software, while the reduced interface only showed a subset of the features. For the study, they recruited 69 undergraduate students, gave them the interfaces, and asked them to solve twenty geometry problems (the pretest) with aid from software. Then the students were given a series of video lessons that explained each of the software's features and asked again to solve twenty geometry problems (the posttest). The researchers found that the students in the reduced interface scored higher on average in the pretest, but the average scores of the students using the complete interface were higher in the post-test. This suggests that to novice users of a new program, a simplified user interface was more useful in learning the concepts(Borges et al., 2016).

Chapter 3

Constrained Sets Design and Implementation

Constrained Sets is a system that allows for only a subset of App Inventor's blocks and components to be shown to the user. This section will go into the design of Constrained Sets, the reasoning behind design and architectural choices, and how it is implemented in App Inventor.

3.1 Design

A "set", as used in this thesis, is defined as a collection of pre-specified App Inventor blocks and component types.

The Constrained Sets systems consists of two main modules: the set generator and the display. The set generator takes in a specified set and generates a representation of that set. That representation is then inputted into the display, where it changes the App Inventor interface to reflect the set.

3.1.1 Constrained Set Representation

A Constrained Set is represented as a JSON string that specifies which components and block types are in the shown set:

```
{"shownComponentTypes":
        {"USERINTERFACE":[{"type":"Button"},{"type":"CheckBox"}],
        "LAYOUT":[],
        "MEDIA":[],
        "ANIMATION":[],
        "SENSORS":[],
        "SOCIAL":[],
        "STORAGE":[],
        "CONNECTIVITY":[],
        "LEGOMINDSTORMS":[],
        "EXPERIMENTAL":[],
        "EXTENSION":[]},
"shownBlockTypes":
        {"ComponentBlocks":
                {"Button":
                        [{"type":"component_event",
                                "mutatorNameToValue":
                                 ↪  {"component_type":"Button","event_name":"Click"},
                                "fieldNameToValue":{}},
                        {"type":"component_set_get",
                                "fieldNameToValue":{"PROP":"BackgroundColor"},
                                "mutatorNameToValue":{"component_type":"Button","property⌋
                                 ↪  _name":"BackgroundColor","set_or_get":"set"}},
                        {"type":"component_set_get",
                                "fieldNameToValue":{"PROP":"BackgroundColor"},
                                "mutatorNameToValue":{"component_type":"Button","property⌋
                                 ↪  _name":"BackgroundColor","set_or_get":"get"}}],
                "CheckBox":
                        ...
                        },
        "Logic":[],
        "Control":[{"type":"controls_if"},
                {"type":"controls_forEach"},
                {"type":"controls_while"}],
        "Math":[],
        "Text":[],
        "Lists":[],
        "Colors":[],
        "Variables":[],
        "Procedures":[]}}
```

The field value of shownComponentTypes represents which components are shown and the field value of shownBlockTypes represents which blocks are shown. The value of ComponentBlocks lists which component-specific blocks are displayed.

In order to make it easy for developers and instructors to create Constrained Set models, it is necessary to create a generator that allows them to convert a set into its representation. This generator is described below.

3.1.2 User Experience and Project Dependence

We created the SubsetJSON project property field, which holds the Constrained Set representation for that project. If a user has a set and wants to display it in an App Inventor project, they first would create the set representation using the set generator, and then take the output from the generator and input it into the SubsetJSON project property box.

By having each project hold its own Constrained Set through a project property, we allow Constrained Sets to be project dependent.

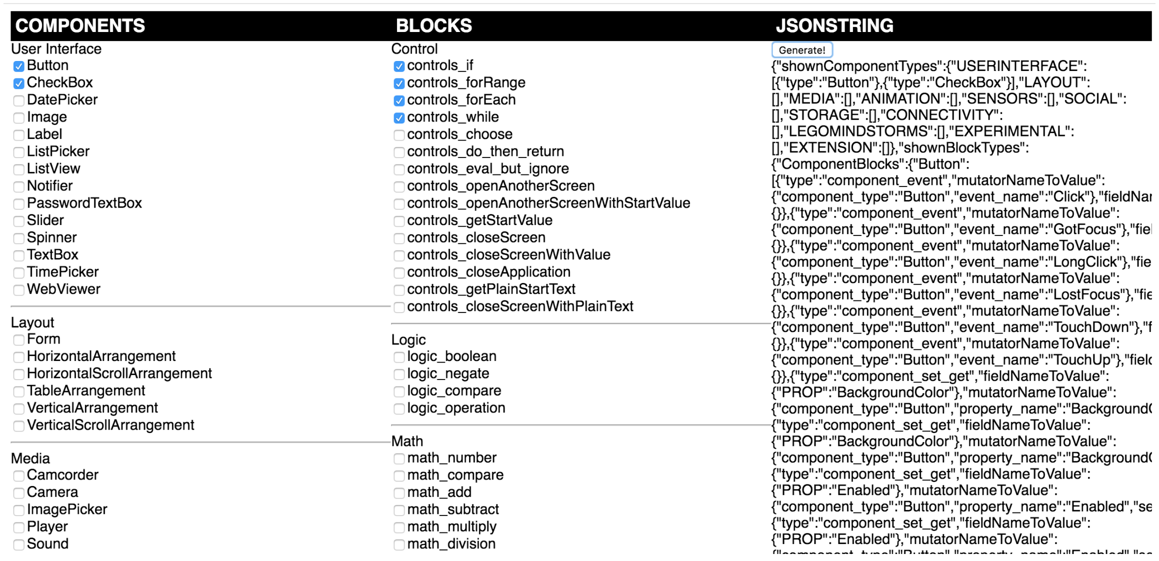
Constrained Sets is project dependent for a variety of reasons:

- Users, regardless of their experience with App Inventor, should have exposure to all functions.
- Since new users would not be aware if their environment was limited, Constrained Sets should be opt-in.
- A single user could have different projects that use different components, so we want to allow them to be able to switch between the contexts easily.

3.2 Implementation

This section goes into the implementation of Constrained Sets in App Inventor's current GWT-based designer. Constrained Sets was also implemented in a new React based designer, which is described in more detail in Appendix A.

3.2.1 Constrained Set Generator



COMPONENTS

User Interface
- ☑ Button
- ☑ CheckBox
- ☐ DatePicker
- ☐ Image
- ☐ Label
- ☐ ListPicker
- ☐ ListView
- ☐ Notifier
- ☐ PasswordTextBox
- ☐ Slider
- ☐ Spinner
- ☐ TextBox
- ☐ TimePicker
- ☐ WebViewer

Layout
- ☐ Form
- ☐ HorizontalArrangement
- ☐ HorizontalScrollArrangement
- ☐ TableArrangement
- ☐ VerticalArrangement
- ☐ VerticalScrollArrangement

Media
- ☐ Camcorder
- ☐ Camera
- ☐ ImagePicker
- ☐ Player
- ☐ Sound

BLOCKS

Control
- ☑ controls_if
- ☑ controls_forRange
- ☑ controls_forEach
- ☑ controls_while
- ☐ controls_choose
- ☐ controls_do_then_return
- ☐ controls_eval_but_ignore
- ☐ controls_openAnotherScreen
- ☐ controls_openAnotherScreenWithStartValue
- ☐ controls_getStartValue
- ☐ controls_closeScreen
- ☐ controls_closeScreenWithValue
- ☐ controls_closeApplication
- ☐ controls_getPlainStartText
- ☐ controls_closeScreenWithPlainText

Logic
- ☐ logic_boolean
- ☐ logic_negate
- ☐ logic_compare
- ☐ logic_operation

Math
- ☐ math_number
- ☐ math_compare
- ☐ math_add
- ☐ math_subtract
- ☐ math_multiply
- ☐ math_division

JSONSTRING

```
{"shownComponentTypes":{"USERINTERFACE":
[{"type":"Button"},{"type":"CheckBox"}],"LAYOUT":
[],"MEDIA":[],"ANIMATION":[],"SENSORS":[],"SOCIAL":
[],"STORAGE":[],"CONNECTIVITY":
[],"LEGOMINDSTORMS":[],"EXPERIMENTAL":
[],"EXTENSION":[]},"shownBlockTypes":
{"ComponentBlocks":{"Button":
[{"type":"component_event","mutatorNameToValue":
{"component_type":"Button","event_name":"Click"},"fieldNar
{}},{"type":"component_event","mutatorNameToValue":
{"component_type":"Button","event_name":"GotFocus"},"fiel
{}},{"type":"component_event","mutatorNameToValue":
{"component_type":"Button","event_name":"LongClick"},"fie
{}},{"type":"component_event","mutatorNameToValue":
{"component_type":"Button","event_name":"LostFocus"},"fie
{}},{"type":"component_event","mutatorNameToValue":
{"component_type":"Button","event_name":"TouchDown"},"f
{}},{"type":"component_event","mutatorNameToValue":
{"component_type":"Button","event_name":"TouchUp"},"fiel
{}},{"type":"component_set_get","fieldNameToValue":
{"PROP":"BackgroundColor"},"mutatorNameToValue":
{"component_type":"Button","property_name":"Background
{"type":"component_set_get","fieldNameToValue":
{"PROP":"BackgroundColor"},"mutatorNameToValue":
{"component_type":"Button","property_name":"Background
{"type":"component_set_get","fieldNameToValue":
{"PROP":"Enabled"},"mutatorNameToValue":
{"component_type":"Button","property_name":"Enabled","se
{"type":"component_set_get","fieldNameToValue":
{"PROP":"Enabled"},"mutatorNameToValue":
```

Figure 3-1: The App Inventor designer

The set generator is built as a standard website and uses two existing App Inventor JSON files, *global_ block_ checkbox.json* and *simple_ components.json*, to generate the default block and component dependent checkboxes.

When a user clicks on a component, all of the component blocks are automatically added to the shown blocks list. This is designed so that removing blocks is an intentional action (See Specifications and Exceptions).

3.2.2 Displaying the Blocks and Components

When a user inputs the set representation into the SubsetJSON project property and refreshes the page, the project will be rerendered with the Constrained Set. When the subset property is changed, App Inventor's auto-save function automatically writes it onto the server. Because the version that we implemented on GWT is not reactive, the page will need to be reloaded in this version to show the new component sets. This issue is fixed inside the React version.

Components

The onFileLoaded() function is modified to load in the new components when a user refreshes the project. It first checks whether the project's SubsetJSON property contains a valid set representation. If so, it loads its respective component set by only adding the components in the Constrained Set instead of adding all the components.

Blocks

Since Blockly exists as its own Javascript library, if a change in the blocks set is made, the project does not need to be reloaded.

In implementing Constrained Sets, we changed how block drawers were displayed in Blockly so that drawers contained JSON representations of the blocks. In the previous system, Blockly instantiates every block type in that category and iterates over the order the keys were added, so the block order is determined by what is defined in the file. This means that block order could not be changed dynamically, or easily reordered or rearranged on the go.

Using previous work done on App Inventor by Janice Chui, we added functionality that converted JSON representations of blocks to their respective XML formats. This means that blocks in drawers can be reordered and changed easily, and also allows block sets to be easily created since the representation of a Constrained Set is already a JSON object. The drawer can create exactly the blocks it needs from the set model directly, and little modification on the code is needed.

3.2.3 Specifications and Exceptions

We designed the specifications and exceptions in a way that prioritized giving the user more access in these situations, especially since novice users will more easily find a block or component than figure out how to change their interface to show the block or component they need. We always want the user to be in a state where they can complete their project, and Constrained Sets is designed in a way that tries to satisfy that condition. Therefore if the Constrained Set properties box is empty or contains an malformed input, it will automatically default to the full App Inventor set.

Chapter 4

User Experiment and Data Analysis

In this section we discuss the design and execution of a research workshop that explores the effects of integrating Constrained Sets with tutorials as a teaching tool to onboard new learners of App Inventor.

4.1 Hypotheses

Before we conducted the workshop, we made the following hypotheses:

1. Students in the control group would use a larger variety of blocks and components because they have chances to explore the open environment for longer periods of time.

2. Students in the experimental group complete more of the explorations (see Tutorial Phase) since they had a smaller set of combinations of components and blocks, had lesser cognitive load, and were constrained in their explorations.

4.2 Study Participants

We recruited nine participants for our study through a workshop we conducted with MIT's Spring HSSP[1]. The students that signed up for the workshop indicated interest

[1]MIT Spring HSSP (2017) is a six to eight week long program in which middle and high schoolers sign up to attend courses taught by members of the MIT community.

in creating mobile applications, which fit the App Inventor target demographic. The participants were not required to have prior programming knowledge. Students who signed up for the workshop did not need to participate in the study, and not all of the members of the workshop chose to participate.

4.3 Study Overview

The workshop spanned six weeks, with one ninety-minute session per week. Participants worked individually, but natural collaboration was permitted. There were also instructors present to guide participants if they got stuck, but the instructors were asked to push the students to explore and figure out solutions by themselves first.

The nine participants were split randomly into two groups: five in the experimental group, and four in the control group. They were all given the same computers, tutorials, and instructions, the only variable between the two groups was the environment in which they did the tutorials.

The study was split into two phases: the tutorial phase and the final project phase, both which are described below:

4.3.1 Tutorial Phase

The tutorial phase consisted of the first three weeks of the workshop, where students were taught App Inventor through a series of tutorials. Since a majority of App Inventor learners teach themselves App Inventor in an informal environment via tutorials, this is a good representation of App Inventor's general user onboarding experience.

The tutorials used in this phase consisted of four original App Inventor tutorials, and two tutorials built in collaboration with Youth Radio [2]. The curriculum is detailed below:

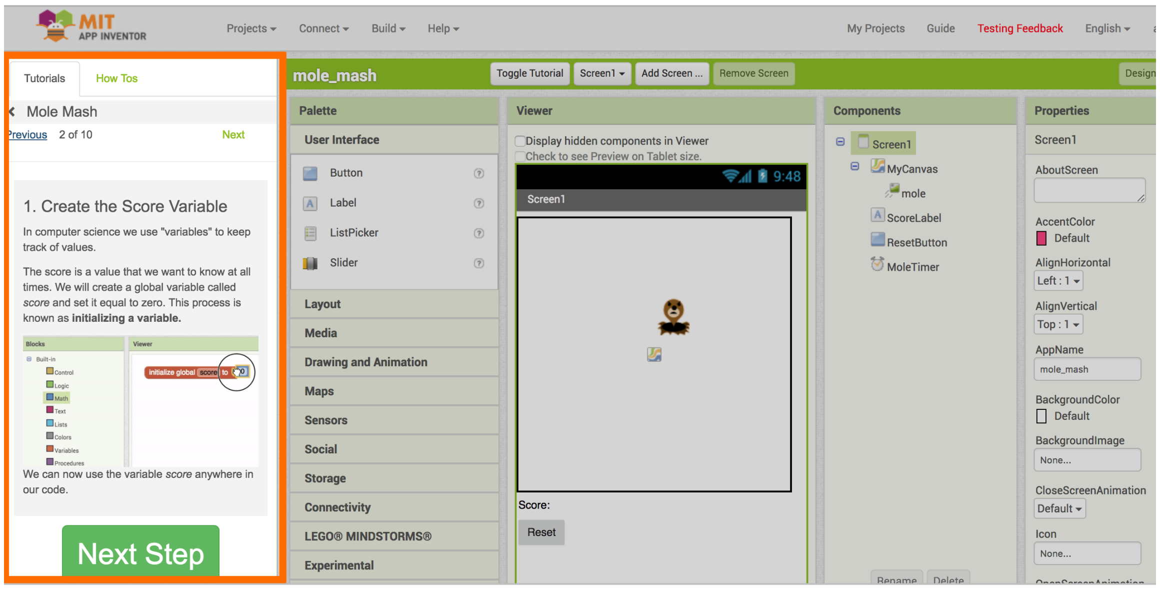
[2] Youth Radio (2016) is non-profit media company located in Oakland, California dedicated to encouraging youths to use media to improve their community.

Table 4.1: Workshop Tutorial Curriculum

Week	Tutorials
Week 1	Talk To Me, Magic 8 Ball
Week 2	Translation App, Snapchat Remix
Week 3	Mole Mash, Get The Gold

The tutorials were displayed to the students through a tutorial side panel that guided the students through how to build a specific app with step-by-step instructions and animated GIFs. It included which blocks they needed to use and directions on how to access those blocks.

Figure 4-1: The Tutorial Panel



1. Create the Score Variable

In computer science we use "variables" to keep track of values.

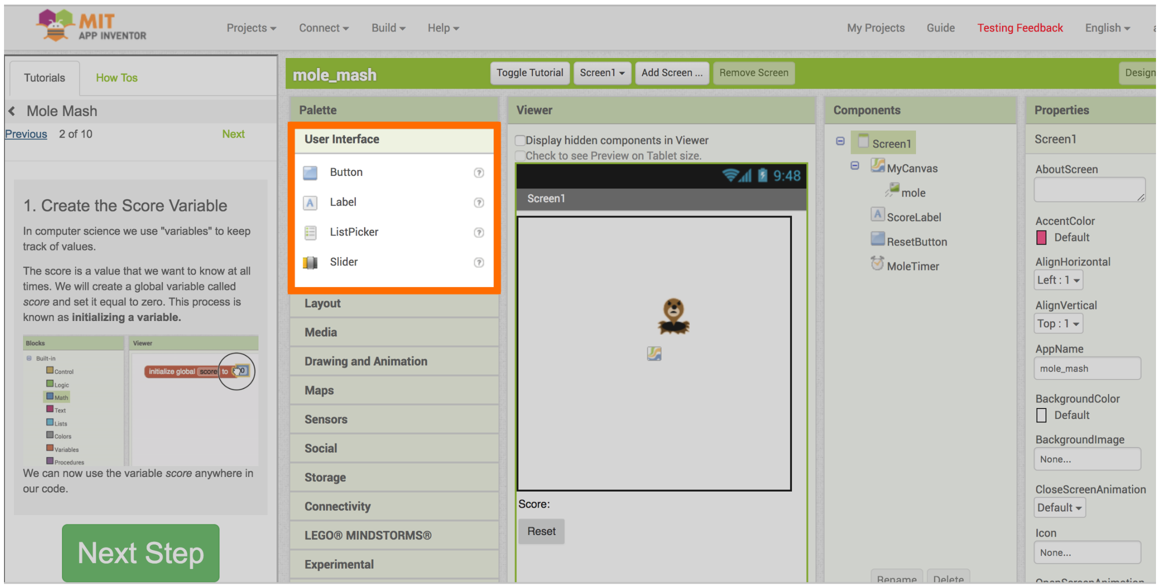
The score is a value that we want to know at all times. We will create a global variable called *score* and set it equal to zero. This process is known as **initializing a variable.**

We can now use the variable *score* anywhere in our code.

The highlighted section shows the tutorial for Mole Mash in the App Inventor designer. Tutorials on the App Inventor website, originally in PDF format, were divided into individual chunks and processed in this panel format so that students could reference and follow the tutorial in the same window.

At the end of every tutorial is an extended exploration section that suggests ways that participants could extend or add functionality to the app the students had just built. The explorations are not guided like the rest of the instructions, so students must figure out how to build them for themselves. The experimental group completed the tutorials in a constrained environment that only gave them the blocks and components they needed to complete the tutorials and explorations, while the control group was given access to all of App Inventor's blocks and components.

Figure 4-2: Design Editor Constrained vs Full Comparison



Mole Mash

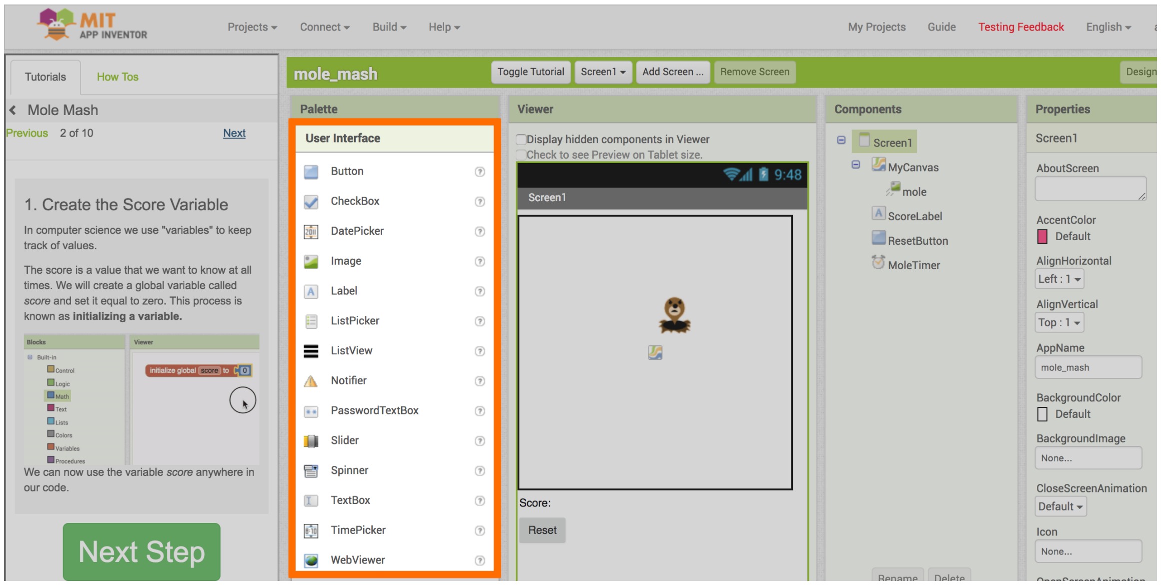
Previous 2 of 10 Next

1. Create the Score Variable

In computer science we use "variables" to keep track of values.

The score is a value that we want to know at all times. We will create a global variable called *score* and set it equal to zero. This process is known as **initializing a variable.**

We can now use the variable *score* anywhere in our code.



Mole Mash

Previous 2 of 10 Next

1. Create the Score Variable

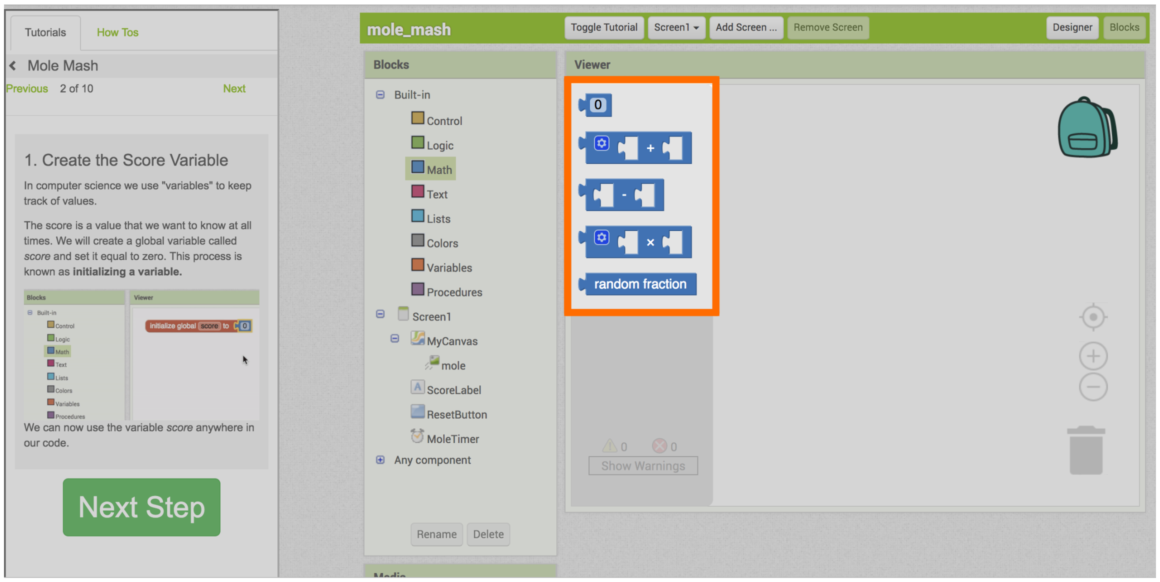
In computer science we use "variables" to keep track of values.

The score is a value that we want to know at all times. We will create a global variable called *score* and set it equal to zero. This process is known as **initializing a variable.**

We can now use the variable *score* anywhere in our code.

Images showing the comparison in the design page between the constrained environment (top) and the full environment (bottom). Note the difference in the number of components that are available to the users.

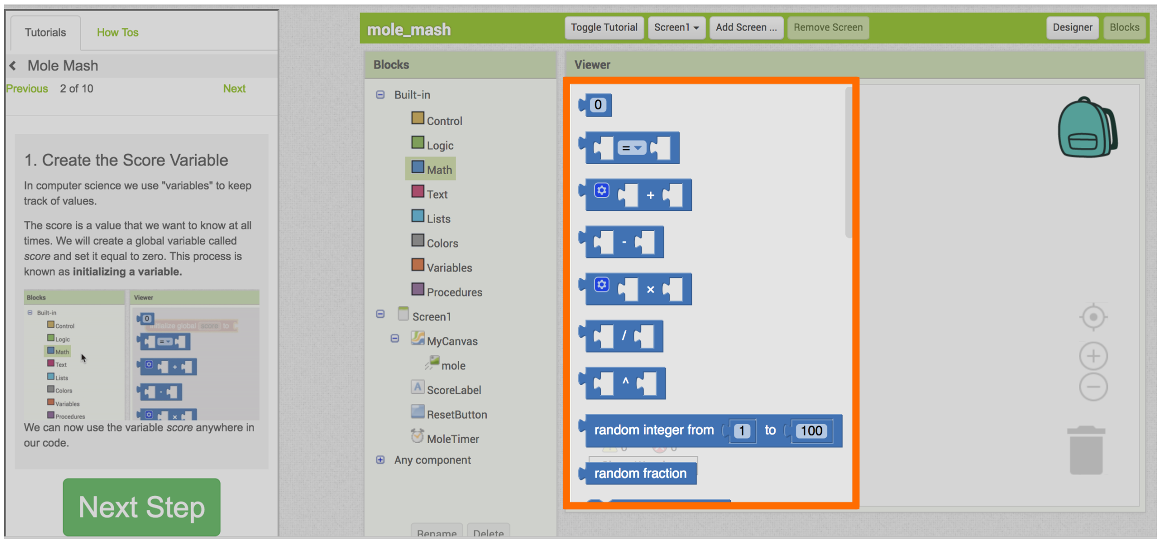
Figure 4-3: Blocks Editor Constrained vs Full Comparison


1. Create the Score Variable

In computer science we use "variables" to keep track of values.

The score is a value that we want to know at all times. We will create a global variable called *score* and set it equal to zero. This process is known as **initializing a variable.**

We can now use the variable *score* anywhere in our code.


1. Create the Score Variable

In computer science we use "variables" to keep track of values.

The score is a value that we want to know at all times. We will create a global variable called *score* and set it equal to zero. This process is known as **initializing a variable.**

We can now use the variable *score* anywhere in our code.

Comparison between the constrained environment (top) and the full environment(bottom) for the Math block drawer. Users in the constrained environment are only given the blocks that were needed to complete the tutorial.

4.3.2 Project Phase

Once the participants completed the tutorials, they were then asked to design and create their own app using App Inventor. Because the participants expressed interest in creating games, the theme of the final project was game-based, but they were also allowed to create apps with different themes as well if they desired. They were given thirty minutes to brainstorm their apps, and to aid them with the process, we gave them worksheets to help them scaffold their designs. In this phase, all the participants were given access to the full functionality of App Inventor.

4.4 Data Collected

We collected three forms of data for all participants: surveys, project files, and screen recordings. Additionally, we conducted interviews with some of the participants at the end of the workshop, and instructors also noted down significant events such as students asking for help from either the instructors or their neighbors.

We gave participants questionnaires at three stages: before the workshop, after the tutorial phase, and after the workshop. The pre-workshop survey asks them about their previous programming experiences, while the post-tutorial and post-workshop survey asks them about their experience learning App Inventor and using the App Inventor interface. After each tutorial, participants were additionally asked to rate their understanding and write a segment on things they had learned. All questionnaires are included in Appendix B.

4.5 Data

4.5.1 Classroom Statistics

The participants of the workshop were 7^{th} and 8^{th} graders between the ages of 12 and 13. Most of them have been exposed to programming before, 40% of the students in the experimental group and 75% in the control group had done so in a classroom environment.

In the month prior to the workshop, members of the experimental group programmed an average of 1.8 hours per person. All but one member in that group programmed for an hour or less both inside and outside the classroom; the last person programmed between 2 to 4 hours outside of class. The experimental group had a pretty even distribution with their past programming experience.

On the other hand, in the month prior to the workshop, members of the control group programmed an average of 7 hours per person. This average was greatly skewed by one of the members, who programmed for more than 10 hours in the past month, both inside and outside the classroom. If we exclude this outlier, the rest of the control group would have programmed for an average of 2.6 hours per person. The control group showed a more varied distribution in their prior programming experiences, from the aforementioned student to a student that had no prior programming experience.

Because attendance was not mandatory, one of the participants in the experimental group left after the tutorial phase, so we did not include them in the calculations that required final project data.

4.5.2 Project Statistics

Counting Block Types

We counted block and component types by using App Inventor's AIATools [3]. We felt that the three broad default types for component blocks: **component_get_set**, **component_event**, and **component_method** did not provide enough granularity since each of those default types can cover a large range of component blocks. Therefore, we separated blocks by not only their default type, but by the mutation they have. Event blocks are now divided based on what event they have and method blocks are now divided based on what method they call. We have opted to keep the get/set blocks under one category because we believe the concept of getting and setting a variable is universal. This is similar to the way Benjamin Xie counted his blocks as well since he states that many of his blocks came from component functionality (Xie, 2016).

[3]AIATools is a Python library developed by the App Inventor team that summarizes and analyzes data from .aia files (The file type that App Inventor projects are saved in)

Tutorial Phase

Students in the experimental group completed on average 80% of the tutorials they started, while students in the control group completed 100% of the tutorials they started. We define tutorial completion as having a complete, working, and bug-free version of the app shown in the tutorial. On average, students in the experimental group completed 1.56 explorations per tutorial while students in the control group completed 1.52 explorations per tutorial.

	Experimental	Control
Blocks	24.56	25.58
Components	7.83	8.29
Block Types	7.83	8.58
Component Types	6.02	6.06

Table 4.2: The average number of blocks, components, block types, and component types, used per tutorial project in the two groups.

Final Project Phase

	Experimental	Control
Blocks	122	128.75
Components	20.5	21
Block Types	19.5	15
Component Types	8.5	6.25

Table 4.3: The average number of blocks, components, block types, and component types used in the final project in the two groups

It is interesting to note that for the final project, compared to the control group, users in the experimental group used around the same number of blocks and components, but a higher number of block and component types. Two of the participants in the experimental group also mentioned that they used App Inventor outside of the workshop because they wanted to know more.

Wilcoxon Rank Sum: To figure out if the numerical difference in block types is statistically significant we used a Wilcoxon Rank Sum test. We chose this test since different participants were used in each group and the block type distributions were not likely to be normal. The test revealed that the number of block types used in the experimental group was statistically significantly different than the number of block types used in the control group $W_s = 10, Z = -2.38, p = 0.02, r = -0.84$. Median for experimental = 4.5, median for control = 0.5.

Block and Component Type Usage Breakdown

The following tables detail the cumulative amount of block and component types used by the participants throughout the workshop.

Table 4.4: Cumulative Block Types Used

Student ID	Block Types
Student A	31
Student B	47
Student C	40
Student D	47

Student ID	Block Types
Student E	39
Student F	32
Student G	38
Student H	36

The number of block types used over the course of the full workshop in the experimental group (left) and the control group (right)

Table 4.5: Cumulative Component Types Used

Student ID	Component Types
Student A	12
Student B	17
Student C	16
Student D	13

Student ID	Component Types
Student E	16
Student F	8
Student G	14
Student H	12

The number of component types used over the course of the full workshop in the experimental group (left) and the control group (right)

The next two tables detail the "new" block and component types used by the two groups, where "new" is defined as types that the participants used in the final project but not in any of the previous tutorials.

Table 4.6: New Block Types Used

Student ID	Block Types
Student A	6
Student B	4
Student C	1
Student D	5

Student ID	Block Types
Student E	11
Student F	1
Student G	0
Student H	0

The number of block types used in the final project by each student in the experimental group (left) and control group (right) that were not used in any of the previous tutorials

Table 4.7: New Component Types Used

Student ID	Component Types	Student ID	Component Types
Student A	1	Student E	3
Student B	2	Student F	0
Student C	2	Student G	0
Student D	1	Student H	0

The number of component types used in the final project by each student in the experimental group (left) and control group (right) that were not used in any of the previous tutorials

4.5.3 Survey Statistics

Below are the average survey results for the surveys we gave to the students after the tutorial phase and after the whole workshop. We asked them to rate how much they agree with the following statements on a scale of 1 (Strongly Disagree) to 4 (Strongly Agree).

Table 4.8: Post Tutorial Survey Results

Statement	Experimental	Control
It was easy to complete the tutorial apps.	2.8	3.3
I easily found everything I needed in App Inventor to complete the tutorials.	3.4	2.3
It was easy to build the extensions to the tutorial apps.	2.8	3
I easily found everything I needed in App Inventor to complete the tutorial extensions.	3.2	2.7
The interface was enjoyable to use.	3.8	3.3

Table 4.9: Post Workshop Survey Results

Statement	Experimental	Control
App Inventor was easy to learn.	3.5	3.5
The tutorials prepared me well enough to build my own App Inventor application.	4	4
I want to make more apps with App Inventor.	3.5	4
I know how to figure out how to do something in App Inventor.	4	4
I am familiar with App Inventor	4	4
I understand how the blocks interact with each other	4	4
I understand the App Inventor interface	3.5	3.5
The constrained interface was easier to use	3	N/A
The constrained interface was more enjoyable to use	1.5	N/A
The constrained interface helped me learn the full interface	2	N/A

4.5.4 Case Studies of Three Categories of Participants

Although the students were randomized into the two groups, we can categorize students to belong to one of three different categories: Experimental Group - Beginner, Control Group - Experienced, and Control Group - Beginner. In this section we will go into some case studies of the students in those respective categories.

Experimental Group - Beginner

Students A through D all fell inside this group; for the sake of brevity in this case study, we will only look at Student A and Student B.

Student A and Student B are both middle schoolers in the experimental group, who have had both done some amount of programming before, but only programmed for less than one hour in the previous month. This indicates that while they have had exposure to programming concepts, they were still novice programmers.

Student A completed 40% of the tutorials and seemed to dislike the limited amount of blocks and components they got, especially in the earlier tutorials. However, once the access for blocks and components got bigger, they were more eager to explore around the interface. They were among the first to notice that the components and blocks subsets changed and were disappointed that one of the components they used in a previous tutorial was not there for a later tutorial.

Even though Student A got the constrained environment, there were many times when they wanted to implement something in the final project and asked for instructor assistance. The instructor never gave them direct assistance, and instead would prompt Student A to try figuring it out by themselves, which they did. Despite these successes, Student A still continued to ask for instructor assistance.

Both Student A and Student B explored App Inventor outside of the workshop. In between workshop sessions 3 and 4, we sent a notification to the students to let them know that we were transitioning from the tutorial phase to the project phase, and asked them to think of app ideas that they would want to make for their final project. Student A brought in an App Inventor project that they had worked on outside of the workshop and asked to continue building on it for their final project. They said that they had used lessons that they had learned from tutorials 5 and 6. Since the tutorial interface for those specific tutorials is not readily accessible on the public App Inventor server, it is assumed that Student A recreated the lessons from memory or referenced online materials.

Student B also worked on their final project outside of the workshop. They wanted to implement a mechanic that used the phone's accelerometer but didn't know how to do it since it was not covered in any of the tutorial materials. Because they felt that they didn't have enough time in class to create their final project, they asked for and were given permission to work on it outside of class. They were eventually able to implement their desired function.

Control Group - Experienced

Student E and Student F are two students in the control group who have both had experiences with programming and programming languages before coming to this course. They are examples of students who may be novices to App Inventor but are familiar with coding paradigms. Prior to the workshop, Student E had spent more than 10 hours, both inside and outside the classroom, programming. They had also stated that they have had experience using standard text-based programming languages, such as Python. Student F had spent less time programming in the prior month than Student E; they had only spent 2 to 4 hours programming outside of class. However, Student F told us that they spent a lot of time making games in Scratch, which made them more familiar with block-based programming, programming concepts, and computational logic. For this reason, we have placed them in the experienced group.

During the tutorial phase, Student E and Student F completed an average of 1.875 extensions per tutorial project.

When brainstorming ideas for their final project, Student E and Student F chose to work on their own instead of sharing and discussing ideas with the other participants.

Student E usually asked questions on how to do something with App Inventor that was not immediately obvious. For example, they wanted to pass data between screens and could not find the component and blocks that did what they needed, so they asked an instructor for help. Student F, on the other hand, was mostly able to find the blocks that they needed for their final project, but would ask the instructors for debugging help. The instructors would ask Student F to walk through their code, at which point Student F was usually able to figure out the issue themself.

Control Group - Beginner

In contrast to Students E and F, Students G and H were participants in the control group with little previous programming experience before coming to the workshop. Student G programmed for similar amounts of time with Student F, but unlike Student F, he had less experience with other block-based programming languages. Student H, on the other hand, had no prior experience with programming before coming to this workshop.

While Student F and Student G would sometimes collaborate and help each other find blocks in the tutorials, Student H would look at Student G's screen for aid and direction. Whenever Student H needed help, he would often ask Student G, or copy off what Student G had. Student H's final project was also a similar app to that of Student G's, suggesting that this pattern had carried over from the tutorial phase. Student H also experienced some struggles with their final project. When they asked Student G for help, Student G told them that the implementation for the idea was similar to that of one of the previous tutorials. Even though Student H was able to complete all of the tutorials, they were not able to recreate and modify the tutorial projects later on.

Chapter 5

Discussion and Conclusion

5.1 Discussion

Due to the small sample size of our experiment, it is important to note that these findings are not conclusive and more experiments are needed. We hope these findings will inspire further research into how interfaces can affect learning software and computational thinking.

Students in the experimental group used a larger variety of blocks in their final projects than students in the control group

Contrary to our hypothesis, students in the experimental group used a larger variety of blocks in their final projects than students in the control group. While we initially thought this could be due to the fact that students in the control group had already explored more of App Inventor during the tutorial phases and therefore were more focused in their final project, the cumulative data seems to contradict this argument. If this was indeed the case, then we should see equal or higher number of block types used cumulatively by the control group students. We also manually reviewed some of the screen recordings of members in the control group to verify that they had not used or explored any blocks that were outside the subset of those required to complete the tutorials and their extensions.

Additionally, we calculated the ”new” block types used by the two different groups as a way to establish a minimum baseline for exploration of different types of functions. We found that on average, students in the control group explored and used more "new" blocks than students in the experimental group. The one anomaly in the control group data, Student E, could be explained by the fact that they were absent from one of the tutorial sessions and they were not a novice programmer. As they stated in their interview, it was easy for them to catch up and figure out what they needed to do given their programming background.

We offer two possible explanations for this behavior:

1. Backlash from using the restrictive constrained environment

   The participants in the experimental group could be eager to finally use the rest of App Inventor after being constrained in the limited environment. They may have felt that they didn't get the same chance to explore as those in the control group, and as a result, made up for it by trying more things.

2. Students felt that they had built a solid foundation in the constrained sets and were not scared to try out new things.

   This explanation is more aligned with Vygotsky's theory on the Zone of Proximal Development and how students learn. The limited environment could have served as adequate scaffolding for students to expand their Zone of Proximal Development, and as a result, students in the experimental group were less scared to try out new things when they were in the open environment. Even though the two groups self-reported the same amount of learning and comprehension in each of the projects, students in the experimental group could have felt that they were more familiar in their limited environment and were ready to explore the rest of App Inventor. This hypothesis is hard to prove without further supporting evidence, and will require more research.

A constrained environment has little effect on the number of explorations a student completes

We also hypothesized that a constrained environment would increase student efficacy in using App Inventor since there were only a limited number of combinations students could use. This was not the case, however, as students from the two groups completed approximately the same number of tutorial explorations. Even though the experimental group more easily found the things they needed to complete the tutorials and explorations, they did not find completing the tutorials easier, suggesting that the cognitive load required to locate the needed components/blocks did not severely impact learning App Inventor.

5.1.1 Other Findings

Although the constrained interface was easier to use, it was neither enjoyable nor helpful in learning the full interface.

Participants in the experimental group mentioned that while they did find that the constrained interface was easier to use, they did not feel that it helped them learn the full interface of App Inventor, nor was using it enjoyable for them to use. Interestingly enough, in the survey after the tutorial phase was completed, the participants in the experimental group answered that they found the interface to be easy and enjoyable to use. It was only until they were exposed to the full interface that they came to dislike the constrained interface.

In interviews conducted with some of the participants after the workshop, they stated that they thought the constrained interface was too restrictive, and while they believe that constrained interfaces would help people learn better, they also wanted a larger degree of freedom. The overall feeling was that they were unsatisfied with the sizes of the sets that they were given. Participants in the experimental group also felt negatively about the fact that sometimes they could not access blocks and components that they had used previously.

"I was mad about that." — Student A, when asked how they felt when they realized

that some of the blocks they used in the previous tutorial were no longer available to them.

The study conducted with Gameblox Flexidor also revealed that participants enjoyed using Flexidor less than using the full editor, even though using Flexidor was easier (Du, 2015). Participants in the Flexidor study, however, said that Flexidor helped them learn more of the full interface. This could be due to the fact that students in the Constrained Sets workshop may have interpreted "interface" as App Inventor's functionality and not purely the GUI. Participants in this workshop are also novice programmers as opposed to the participants in Du's study, who were MIT undergraduates with prior programming experience (Du, 2015).

5.2 Conclusion

This thesis detailed the design and implementation of a system that creates different environments in App Inventor by displaying only a subset of App Inventor's tools. We proposed different ways of using Constrained Sets to improve the user experience, and also conducted research using it to test the effects of limited environments on learning. We held a six week workshop that introduced kids to App Inventor and asked ourselves two questions:

1. Does using a limited environment in App Inventor increase the efficacy and comprehension of App Inventor?
2. Does a limited environment affect the breadth of the component types or block types that they explore?

Via our experiment, we have verified that interfaces must be finely tuned in order to aid learning, and restrictive environments may not necessarily help users learn better even though they are easier to use, which aligns with previous research. We have discovered, however, that when novice users transition from a tutorial space to a final project and are given all of the software's tools, users who started out with the restrictive environment used a larger variety of block types. Because of the limited size of our workshop, further research is required to verify this finding.

Chapter 6

Future Work

6.1 Future Research

6.1.1 Varying Range of Subsets for Learning

Subjects from the study commented that while they did find that the Constrained Sets helped them learn the interface of App Inventor, they found that the sets we provided were too limiting. Since we only gave the experimental participants the subset of blocks and components that were required to create the tutorials and extensions, they were not able to explore outside of that range.

As mentioned previously in Chapter 2, a large challenge in using Multi-Layered Interfaces is determining the granularity of the layers and the sets in each layer. Future research with subsets could look into how different ranges, instead of our current binary system, could affect learning App Inventor.

Below we propose some more research directions using Constrained Sets:

- **Provide a larger amount of blocks in the Constrained Set**

  Users in this study were only given the blocks they needed to complete the tutorials and their respective extensions, meaning that if they had extra time, they couldn't make their own extensions. While Constrained Sets allows for the precision of choosing the exact blocks that were shown, when we counted the blocks, we grouped them into pre-set block types. One way we can increase the amount of blocks that users got is to select block types instead of specific blocks.

- **Use compounded block and component sets**

  Participants in the experimental group noticed that they had different blocks and components for every tutorial, so there were times in which they wanted to recreate a feature they learned from a previous tutorial but couldn't because the current set didn't allow for it. If we use compounded block and component sets, this would allow users to revisit materials they've learned in previous lessons.

- **Use the same constrained sets but decrease tutorial hand-holding**

  Even though the explorations were unguided, the tutorials contained a lot of hand-holding, which meant that participants could complete the tutorials simply by following the instructions and not really understand their actions. In this tutorial context, the constrained sets we gave may feel restrictive, but if participants were given tutorials that only contained brief instructions similar to Quizly's (2015) tasks, perhaps the constrained sets could serve as stronger scaffolding.

6.1.2 Using Constrained Sets to create diverse App Inventor workspaces

As mentioned in the introduction, Constrained Sets could also be used to provide different workspaces for different goal-oriented projects. For example, users who use App Inventor to build IoT devices would use a different set of blocks than users who want to build games. Our research showed that it was easier for novice users in the limited environment to find the blocks and components they needed, so it is possible that we use Constrained Sets as a way to improve the user experience for experienced users who know which components and blocks they frequently use.

Appendix A

App Inventor with React: An alternative approach to WYSIWYG programming with Constrained Sets

A.1 A React/Redux Project Designer

In addition to implementing the Constrained Sets system and conducting experiments on the effects of limited environments on learning App Inventor, another significant portion of the work was dedicated to figuring out how to substitute the GWT implementation of the App Inventor designer with a React/Redux implementation through injection. Converting the designer codebase to React/Redux makes UI changes like the one detailed in this thesis easier and more accessible to novice App Inventor developers.

A.1.1 Why React/Redux?

Currently, all of App Inventor is built with Google Web Toolkit (GWT), which makes interface innovation difficult as it requires developers to recompile the codebase anytime a change is made. This process must be done because the interface and server use the same code, and each recompilation can take anywhere from two to eight minutes, making this process a huge time sink. When changes are made in React, however, the page is automatically updated in a few seconds. Additionally, since most modern websites are built in Javascript instead of GWT and the majority of web developers nowadays are more familiar with React, finding new student developers for App Inventor becomes easier. Most importantly, converting the editor to React offers us more flexibility in user interface layouts. If we want experiment more with the layout and move or rearrange parts of the interface beyond the scope of the changes in this thesis, it is much easier for developers to implement them in React than in GWT.

A.1.2 Introduction to React/Redux

The React/Redux Modules

React and Redux are interface libraries built in JavaScript that help create reactive interfaces. By only rendering the modules that change, it saves a lot of computing time. This section goes briefly over the React and Redux data flow process to provide some background information that may be useful in understanding the implementation process. There are a few important modules:

- **Store**

  The store keeps the state (or data) of the application. The state is used to display the components.

- **Components**

  The components are the visual elements that comprise the interface. They are processed by containers, which gets the data from the store and uses that to generate and render the components appropriately.

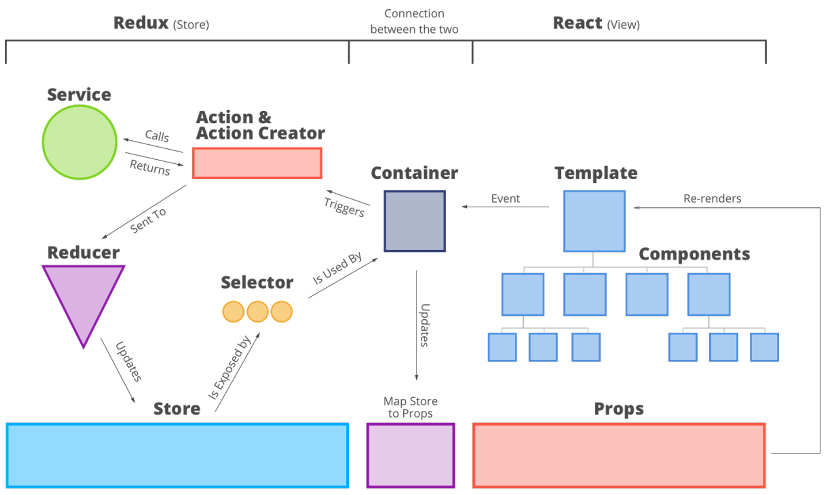
- **Actions**

  Actions are Javascript objects that provide information and are dispatched to the store. They are called when the user interacts with the components, or a data entry in the store needs to be updated.

- **Reducers** Reducers tell the store what to do when an action is dispatched to it. It is a function that takes in (currentState, action) and returns (newState), where newState is the new state of the store after an action has been applied.

The React/Redux Data Flow System

Figure A-1: React/Redux Data Flow Diagram



When a user clicks on a component (for example, changing the color of an element to blue), an action is sent and dispatched by the reducer to the store. The store then looks at its current state, sees that an action has been called to change the color of an element, and updates itself so that in the new state the color of that element is blue. Once the components see that the state has been changed, they rerender themselves (Pini, 2016).

A.1.3 Background Work

Injecting the React editor is mainly based on two previous App Inventor projects: Shelby Pefley's UAP Project "Introducing a Javascript Based Editor into the MIT App Inventor User Interface", and a version of App Inventor in React created by our collaborators at Youth Global Network [1]. Pefley's project was a proof of concept

[1]Youth Global Network (2018) is an organization located in Hong Kong dedicated to empowering youths to create positive changes in their community

that demonstrated that React could be injected into GWT App Inventor; it set a foundation for injecting React code into the current App Inventor's Document Object Model (DOM). Although we have a version of App Inventor built in React, we choose to only inject the designer in because the React version and our current GWT version sends calls to two different servers with their different data formats. React App Inventor sends remote procedural calls to its own server, but the data that we want to access and display is contained in our App Inventor server and formatted differently. Therefore we've opted to inject the React version of the editor into the current App Inventor DOM; this lets us preserve the current data representation and server calls while keeping the benefits React has.

A.1.4 Injecting React into GWT App Inventor

The editors are injected in three main steps:

1. The React App Inventor designer is first isolated to prepare it for injection.
2. We then compress the React editor into a few Javascript chunks and inject them into the GWT DOM using Pefley's method.
3. Finally we populate the React editor with information from the GWT server and connect the corresponding server calls.

Isolating the React Editors

The version of React App Inventor given to us by Youth Global Network includes all of App Inventor, so before we inject the App Inventor editors, we must first extract them from the rest of App Inventor and disconnect their server connections. In order to get the React designer to an injectable state, the designer should be able to be properly loaded on the index page, and have all the components and blocks already working.

Since the React editors provided by Youth Global Network are connected and populated by data from YGN's Node server, we first need to isolate the editors to prepare them for injection. This is done by disconnecting the Node server calls from the editor and creating a dummy project, based off of the project model that the Node server used. Once the dummy project is created, the React designer can use the data from the dummy to render the elements. The data from the dummy project is eventually replaced with actual App Inventor project data.

Injecting the React Editors

Before injecting the isolated React designer, we need to set up the GWT environment for injection. The React designer is injected in the following manner:

- The React designer is first compiled and built. We configure it so that all of the React designer is compressed into 4 files.
- A bash script is then run to move the compiled React files into the source code of the GWT version of App Inventor
- The compiled files are inserted into the location where the designer is via a script injector
- GWT App Inventor is then rebuilt with the new files

Since the React designer includes both the screen editor and the blocks editor, the blocks editor inside the GWT version needs to be disabled as well.

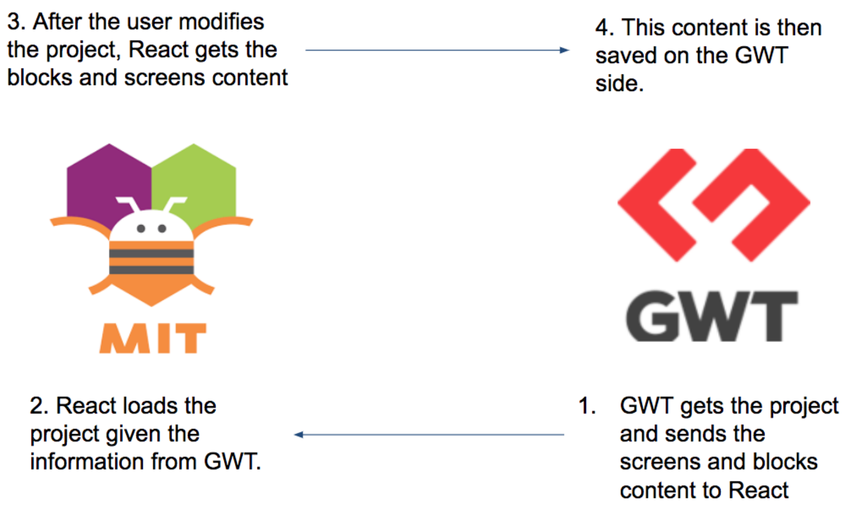
Connecting the RPC Calls

Once the React designer is injected into the GWT DOM, we hook up the data from GWT App Inventor to the React designer, so that the React designer replaces the aforementioned dummy project data with actual project data from the GWT server.

This is done by creating JavaScript hooks that allow React to load and save contents. Given a project served by the GWT server, its blocks and screens content is first converted into a format that React can use, and then sent into the React editor by exposing the data in a JavaScript window function. The React editor then loads up the project that it receives. When a user makes changes inside the React editor and saves the project, the React editor exposes the changed blocks and screens content to GWT using the same method, and calls a function on GWT's side to save the project to the GWT server.

App Inventor projects are stored as aia files, which contain three types of files: .scm, .bky, and .yail. The .scm file stores information about the designer editor contents, and the .bky file stores information about the block editor contents. These are the contents that we send to the React designer so that it can load with them.

Figure A-2: How the React Designer Loads and Saves Projects from the GWT Servers


3. After the user modifies the project, React gets the blocks and screens content
4. This content is then saved on the GWT side.
MIT
GWT
2. React loads the project given the information from GWT.
1. GWT gets the project and sends the screens and blocks content to React

One of the things we had to account for while connecting the RPC calls was changing the React project model to support GWT's. The GWT implementation of App Inventor uses a unique object to store each App Inventor project for a specific user and only saves the project on the server when the page is refreshed. The React version, however, uses one object to store every project, and saves the project when the screen is changed. Because of the differences in project models described above, modifications in the React version were not displayed when the project was opened again. To solve this issue, we had to replicate GWT's multiple project objects inside React, by storing the state of every project opened with the React designer in the store.

Load - GWT first gets all of the designer and blocks content from the server and makes them available through the window functions. It then checks if the React editors are ready, and if so, calls load on the React editors with the designer and block content. The React editors then render with the content that was given to them.

Save - When React saves a project, it sends a call to the GWT server to let it know that the blocks and screens data for that project has been changed, or "dirty". Periodically, GWT will go through all the dirty data and rewrite it with data from the React model.

New Project - When GWT creates a new project, it creates its data and then sends it to React, where a new project is created on the store.

Screen - When React switches screens, it sends an RPC call to GWT to get the content of the new empty screen. It then rerenders itself with the new content.

Upload - Files can also be uploaded through the React system. The React front end first uses a FileReader to read the uploaded file as a DataURL, which is then uploaded to the GWT server. Once the file is loaded, it sends a callback to the React side to rerender the media files list to display the newly uploaded media.

Issues Experienced

Although we would have liked to use the React version to conduct our workshop, we ultimately had to default to using the GWT version of App Inventor due to some of the issues that were discovered. During development of the injection process and connecting the server calls, we uncovered some issues on the React designer. Switching between the injection context and the React designer context was time consuming, and it was difficult to get the up-to-date code for the designer. Additionally, there were some features of App Inventor such as collaboration that were not included inside the React designer.

Since the main goal of this thesis was to look at how limited environments affect learning and exploration in novice App Inventor users, we decided that it would be better to opt for the safer option and run the experiment with the GWT implementation of Constrained Sets.

A.1.5 Implementation of Constrained Sets in React

One of the benefits of React is that it rerenders changes on the go, which means that changes to subsets are automatically displayed and do not require the web page to be refreshed. As we mentioned before in Chapter 4, the implementation of Constrained Sets in GWT App Inventor requires a page refresh in order to display the component subsets.

Components

When the SubsetJSON property box is changed, it dispatches an action to change the subset property in the React store. Once the React store subset property changes, React rerenders the interface with the new component sets by only adding the components that were in the set.

Blocks

In order to get constrained block sets to work in React, we took the constrained block set implementation in GWT App Inventor (See Chapter 3) and compiled it into the file blockly-compiled.js. We then took the compiled file and placed it inside the React codebase.

Appendix B

Surveys

Post Task Reflection + Questionnaire :)

* Required

1. **What is your participant Id? ***

2. **What was the task/tutorial you just finished? ***
   Mark only one oval.
   - ◯ Talk to Me
   - ◯ Magic 8 Ball
   - ◯ Hello Bonjour Translator
   - ◯ Snapchat Remix
   - ◯ Mole Mash
   - ◯ Get the Gold
   - ◯ Your Own Project :D

3. **How difficult did you think this task was going to be? ***
   Mark only one oval.

	1	2	3	4	5	
Very Easy	◯	◯	◯	◯	◯	Very Difficult

4. **How difficult was it actually to complete this task? ***
   Mark only one oval.

	1	2	3	4	5	
Very Easy	◯	◯	◯	◯	◯	Very Difficult

5. **Rate your sense of accomplishment at finishing this task ***
   Mark only one oval.

	1	2	3	4	5	
None	◯	◯	◯	◯	◯	Very High

6. **Rate your understanding of the app that you created** *

Mark only one oval.

	1	2	3	4	5	
I don't understand it - I couldn't remake it	⬭	⬭	⬭	⬭	⬭	I understand everything - I could remake and extend it without instructions

7. **What did you learn/figure out while doing this task?** *

8. **Did you experience any difficulties in building the app? If so, how did you figure out how to solve it, if you did?** *

CS Pre-Workshop Questionnaire

Please fill out and answer the questions below.

* Required

1. **Participant code:** *

2. **Age:** *

3. **Grade Level:** *
   Mark only one oval.
   - ⬭ 5
   - ⬭ 6
   - ⬭ 7
   - ⬭ 8
   - ⬭ 9
   - ⬭ 10
   - ⬭ Option 7

4. **I am a:** *
   Mark only one oval.
   - ⬭ Boy
   - ⬭ Girl
   - ⬭ I don't identify as either
   - ⬭ I'd prefer not to say

5. **Have you ever done any programming?** *
   Mark only one oval.
   - ⬭ Yes *Skip to question 6.*
   - ⬭ No *Skip to question 10.*

If you have done programming before:

6. **Did you do any programming in class?** *
   Mark only one oval.
   - ⬭ Yes *Skip to question 7.*
   - ⬭ No *Skip to question 9.*

Skip to question 10.

If you have done programming in class:

7. **In the last month, how many hours did you spend in a programming class?** *
Mark only one oval.

- ◯ 1 hour or less
- ◯ 2-4 hours
- ◯ 5-10 hours
- ◯ 10+ hours

8. **In the last month, how many hours did you spend programming out of class?** *
Mark only one oval.

- ◯ 1 hour or less
- ◯ 2-4 hours
- ◯ 5-10 hours
- ◯ 10+ hours

Skip to question 10.

If you haven't done programming in class:

9. **In the last month, how many hours did you spend programming?**
Mark only one oval.

- ◯ 1 hour or less
- ◯ 2-4 hours
- ◯ 5-10 hours
- ◯ 10+ hours

Skip to question 10.

How much do you agree with the statements below?

Remember, there are no wrong answers :)

10. *

Mark only one oval per row.

	I agree a lot	I agree a little	I disagree a little	I disagree a lot
I can learn how to program	⬭	⬭	⬭	⬭
I am good at programming	⬭	⬭	⬭	⬭
I have the skills to program	⬭	⬭	⬭	⬭
I have confidence in my programming ability	⬭	⬭	⬭	⬭
I think of myself as someone who programs	⬭	⬭	⬭	⬭
I want to use programming to help solve problems in the world	⬭	⬭	⬭	⬭
I want to use programming to make people’s lives better	⬭	⬭	⬭	⬭
I can use programming to make daily life easier	⬭	⬭	⬭	⬭
I love designing technical/techie things	⬭	⬭	⬭	⬭
I want to learn as much as possible about computer science	⬭	⬭	⬭	⬭
Designing new things makes me feel excited	⬭	⬭	⬭	⬭
I enjoy talking about how technical things work with friends or family	⬭	⬭	⬭	⬭

11. **I think...** *

Mark only one oval per row.

	I am very good at	I am good at	I am OK at	I am not good at
Figuring out how to fix things that don’t work	⬭	⬭	⬭	⬭
Explaining my solutions to technical problems	⬭	⬭	⬭	⬭
Solving problems	⬭	⬭	⬭	⬭
Coming up with new ways to solve technical problems	⬭	⬭	⬭	⬭
Coming up with new ideas when working on problems	⬭	⬭	⬭	⬭

CS Post Tutorial Questionnaire

Please fill this out to the best of your ability!

* Required

1. **User ID:** *

2. **How much do you agree with the statements below?** *
Mark only one oval per row.

	I agree a lot	I agree a little	I disagree a little	I disagree a lot
I can learn how to program.	◯	◯	◯	◯
I am good at programming.	◯	◯	◯	◯
I have the skills to program.	◯	◯	◯	◯
I have confidence in my ability to program.	◯	◯	◯	◯
I think of myself as someone who programs.	◯	◯	◯	◯
I want to use programming to help solve problems in the world.	◯	◯	◯	◯
I want to use programming to make people's lives better.	◯	◯	◯	◯
I can use programming to make daily life easier.	◯	◯	◯	◯
It was easy to complete the tutorial apps.	◯	◯	◯	◯
I easily found everything I needed in App Inventor to complete the tutorials.	◯	◯	◯	◯
It was easy to build the extensions to the tutorial apps.	◯	◯	◯	◯
I easily found everything I needed in App Inventor to complete the tutorial extensions	◯	◯	◯	◯
The interface was easy to use.	◯	◯	◯	◯
The interface was enjoyable to use.	◯	◯	◯	◯

Powered by Google Forms

CS Post Workshop Questionnaire

* Required

1. **User ID** *

2. **How difficult was it to build your final app?** *
Mark only one oval.

	1	2	3	4	5	
Very easy	⬭	⬭	⬭	⬭	⬭	Very difficult

3. **How much do you agree with the statements below?** *
Mark only one oval per row.

	I agree a lot	I agree a little	I disagree a little	I disagree a lot
I can learn how to program	⬭	⬭	⬭	⬭
I am good at programming	⬭	⬭	⬭	⬭
I have the skills to program	⬭	⬭	⬭	⬭
I have confidence in my ability to program	⬭	⬭	⬭	⬭
I think of myself as someone who programs	⬭	⬭	⬭	⬭
I want to use programming to help solve problems in the world	⬭	⬭	⬭	⬭
I want to use programming to make people's lives better	⬭	⬭	⬭	⬭
I can use programming to make daily life easier	⬭	⬭	⬭	⬭

About App Inventor

This section will ask about your App Inventor experience. Please do your best to fill this out as this will help us make App Inventor better!

4. **How much do you agree with the statements below?** *

Mark only one oval per row.

	I agree a lot	I agree a little	I disagree a little	I disagree a lot
App Inventor was easy to learn	⬭	⬭	⬭	⬭
The tutorials prepared me well enough to build my own App Inventor application	⬭	⬭	⬭	⬭
I want to make more apps with App Inventor	⬭	⬭	⬭	⬭
I know how to figure out how to do something in App Inventor	⬭	⬭	⬭	⬭
I am familiar with App Inventor	⬭	⬭	⬭	⬭
I understand how the components interact with the blocks	⬭	⬭	⬭	⬭
I understand how the blocks interact with each other	⬭	⬭	⬭	⬭
I understand the App Inventor interface	⬭	⬭	⬭	⬭

App Inventor Constrained Interface

You may have realized that you use a limited interface when you did your tutorials than when you worked on your final project. The following questions will ask you about your experience with that interface

5. **How much do you agree with the statements below?** *

Mark only one oval per row.

	I agree a lot	I agree a little	I disagree a little	I disagree a lot
The constrained interface was easier to use	⬭	⬭	⬭	⬭
The constrained interface was more enjoyable to use	⬭	⬭	⬭	⬭
The constrained interface helped me learn the full interface	⬭	⬭	⬭	⬭

Powered by Google Forms

References

Blender (2007). Screens. `https://docs.blender.org/manual/en/dev/interface/window_system/screens.html`.

Borges, S., Reis, H., Marques, L., Durelli, V., Bittencourt, I., Jaques, P., and Isotani, S. (2016). Reduced gui for an interactive geometry software: Does it affect students' performance? 54:124 – 133.

Chandler, P. and Sweller, J. (1991). Cognitive load theory and the format of instruction. *Cognition and Instruction*, 8:293–332.

Colter, A. J. (2016). Evaluating and improving the usability of mit app inventor. Master's thesis, Massachusetts Institute of Technology, Electrical Engineering and Computer Science.

Daniels, H. (1996). *An Introduction to Vygotsky*. Routledge.

Du, E. (2015). Gameblox flexidor: Adding flexibility to blocks based programming environments. Master's thesis, Massachusetts Institute of Technology, Electrical Engineering and Computer Science.

Maiorana, F., Giordano, D., and Morelli, R. (2015). Quizly: A live coding assessment platform for app inventor. In *2015 IEEE Blocks and Beyond Workshop (Blocks and Beyond)*, pages 25–30.

Miller, G. A. (1956). The magical number seven, plus or minus two: Some limits on our capacity for processing information. *The Psychological Review*, 63:81–97.

MIT App Inventor (2017). About us. `http://appinventor.mit.edu/explore/about-us.html`,.

MIT Educational Studies Program (2017). Hssp. `https://esp.mit.edu/learn/HSSP/index.html`.

MIT Scheller Teacher Education Program (2014). Gameblox. `https://gameblox.org`.

Nielsen, J. (1993). *Usability Engineering*. Academic Press.

Pini, C. (2016). React + redux: Architecture overview.

Shneiderman, B. (2003). Promoting universal usability with multi-layer interface design. In *Proceedings of the 2003 Conference on Universal Usability*, CUU '03, pages 1–8, New York, NY, USA. ACM.

StarLogo (2010). How to edit subsets. `http://web.mit.edu/mitstep/starlogo-tng/learn/how-to-edit-subsets.html`.

Sweller, J. (1988). Cognitive load during problem solving: Effects on learning. *Cognitive Science*, 12:257–285.

Xie, B. Y. (2016). Progression of computational thinking skills demonstrated by app inventor users. Master's thesis, Massachusetts Institute of Technology, Electrical Engineering and Computer Science.

Youth Global Network (2018). Youth global network. `https://www.ygn.org.hk/en/home/`.

Youth Radio (2016). About youth radio. `https://youthradio.org/`.