Augmented Reality Games: Improved Data Layers

by

Joseph Michael Laurendi

Submitted to the Department of Electrical Engineering and Computer Science
in partial fulfillment of the requirements for the degree of Masters of Engineering in Computer Science and Engineering
at the
MASSACHUSETTS INSTITUTE OF TECHNOLOGY
June 2011

© Massachusetts Institute of Technology 2011. All rights reserved.
Augmented Reality Games: Improved Data Layers

by

Joseph Michael Laurendi

Submitted to the Department of Electrical Engineering and Computer Science on May 20, 2011, in partial fulfillment of the requirements for the degree of Masters of Engineering in Computer Science and Engineering

Abstract

The Scheller Teacher Education Program has been developing Augmented Reality (AR) games software for a number of years. In addition to allowing students to play AR games, the software allows students to create their own games based on a map, a set of characters, and a plot of their choosing. In an earlier version of the software, students were allowed to add “substances” (e.g. oil spills) to their games. The functionality for doing so did not give the student much control and was unintuitive to use. In this thesis, I propose and analyze a prototype of the design and implementation of a new, improved interface that allows students to add a generalized version of a “substance” to their AR games. This generalization of a “substance” is termed a “data layer”

Thesis Supervisor: Eric Klopfer
Title: Associate Professor
Acknowledgments

I would like to thank Eric Klopfer for his vision and ambition which has allowed the ARLogo project to take off in radical new dimensions. I would also like to thank Josh Sheldon for his undying support and ability to stay calm and composed under the most dire of circumstances as project manager. Finally, I would like to thank Lisa Stump for her unwavering commitment to the project, technical experience, and ability to help me stay on track with our ambitious timeline and keep my work in line with the broader goals of the project.
Contents

1 Introduction ........................................................................................................... 15
  1.1 Motivation ........................................................................................................... 16
  1.2 Summary of Thesis Content ............................................................................. 17

2 Background ........................................................................................................... 19
  2.1 Augmented Reality Games .............................................................................. 19
  2.2 Blocks-Based Programming ......................................................................... 24
      2.2.1 StarLogo ..................................................................................................... 25
  2.3 ARLogo ............................................................................................................ 26
  2.4 Substances (i.e. Data Layers 1.0) .................................................................... 27
      2.4.1 Substance Editor .......................................................................................... 27
      2.4.2 Testing Methods ......................................................................................... 29

3 Data Layer Editor .................................................................................................. 31
  3.1 Overview ........................................................................................................... 31
  3.2 Initial Prototype ............................................................................................... 35
      3.2.1 Creating and Initializing a Data Layer ..................................................... 35
      3.2.2 Specifying Behavior (aka the onTick Event) ......................................... 38
      3.2.3 Editing Properties of a Data Layer ......................................................... 40
      3.2.4 Emulating Data Layer .............................................................................. 40
      3.2.5 Toggle Map .................................................................................................. 42
  3.3 Analysis ............................................................................................................. 43
      3.3.1 Mosquitos in the Park .............................................................................. 43
## List of Figures

2-1 Screenshot of a Windows Mobile screen during the Environmental Detectives AR game. The player is currently in range to interview a character. Notice that the map is an overhead view of MIT which is where the game takes place. ........................................... 20

2-2 Example StarLogo code. Notice that the programming is organized in blocks. Further, the blocks are made to resemble biological enzymes which reinforces the science metaphor. ................................. 26

2-3 Here is a screen capture of what the oil spill editor looks like. The user has just picked two points which specify the oil spill. The oil spill is modeled as an ellipse and the two chosen points are the foci of this ellipse. .......................................................... 28

2-4 Here is a screenshot of all of the substances in a particular game (currently there is just one, “toxin1”) and the way the user can edit the properties of the substance including the test methods associated with it. .......................................................... 29

2-5 Shown here is a screenshot of the form a game designer sees when adding or editing a test method associated with a substance. ............... 30

3-1 Main screen of the data layer editor. ................................. 34

3-2 Screen of initializing a data layer. ................................. 35

3-3 Example of initializing a data layer from an image. The original image is on the left while the data layer is shown on the right. ............... 36
3-4 Example of initializing a data layer from points. The points input were located at (50,15) and (50,75) and have radii of length 10 and 20, respectively. ......................................................... 37

3-5 Example of initializing a data layer from a function. The function used is \( f(x, y) = 1/2 \cdot (x + y) \). ......................................................... 37

3-6 Example of an onTick event. ......................................................... 40

3-7 Screenshot of the edit preferences interface. ................................. 41

3-8 Example of emulating running the onTick event 10 times. ........... 42

3-9 Example of the game map being shown. ......................................... 42

3-10 Prototype of wizard dialog for creating a data layer. By employing a wizard process to create a data layer, the user will be guided through the process. Each step in the process should be easily manageable. Furthermore, it leaves less room for error as the path to creating a data layer is very constrained and clear. ................................. 51

3-11 The rest of the prototype of the wizard dialog for creating a data layer. This shows the options for creating a data layer “from points”. The “preview” feature is proposed to be added since it is very important for a user to feel comfortable with their choice. It also increases efficiency since the user wont have to go back through the entire wizard if he ended up not specifying the right points. ................................. 51

3-12 Prototype of the interface for creating a data layer “from points” using mouse click events. By highlighting the selected point, the state of the program is very clearly shown to the user. Also, having the editable properties right below the grid will make it very easy for the user to figure out how to change those properties and will also increase efficiency since there should be instant feedback. ................................. 52
3-13 Prototype of relatively minor changes to improve the UI of the front screen and to bring the "edit properties" section to the same page as the visualization of the data layer. By moving these properties to the same screen as the grid, as opposed to a different tab, this reduces the possible amount of frustration a user might encounter by trying to find where everything is located in our website.

3-14 Prototype of the interface to create, edit, and delete data layer items for a given data layer.

4-1 An example of what the main screen of the mobile client might look like.
List of Tables

3.1 Summary of suggested blocks to be given to specify the onTick event. 39
3.2 Summary of suggested blocks to be given to specify the onTick event. 41
B.1 Summary of results of our UI test of the initial prototype. . . . . . 71
Chapter 1

Introduction

Augmented Reality (AR) is a term that describes the use of a computer to alter the perceived reality of an environment. The spectrum of technologies that are considered to be AR ranges from sophisticated military simulators to relatively simple applications that run on phones. This thesis is concerned with the work that the AR group (MITAR) in the STEP lab at MIT works on. Their work is focuses on creating educational AR software that runs on smartphones. The STEP lab additionally creates software that allows students to design their own AR games. These games consist of a map of the desired region along with virtual characters and items placed at various positions in the game. The students playing the game are given a plot describing what the game is about and are then able to use their smartphones to explore the map, interview virtual characters, and so forth. The current project that MITAR is working on is called ARLogo. This is currently an internal name which may change. The work in this thesis pertains to one specific section of the ARLogo project, namely the topic of data layers.

MITAR has now already created multiple working versions of AR software that allows students to both design and play games. Up until recently there have been two major components to the software: the game editor and the game client. The game editor has traditionally been a Windows-based, desktop application that allows for the designing of games while the game client has been a mobile application running on certain smartphones allowing for the playing of games. The specific mobile platforms
supported by the mobile client has changed throughout the years from Windows Mobile to iPhone and then to Android. Over the past year, a third major component, a central server, has been added to the mix. The server allows for better integration of the game editor and game client and also increases the potential functionality immensely.

The ARLogo project looks to completely rebuild these three components from the ground up. When the server was initially added, a lot of the code for the game editor and game client was rushed in order to meet the deadline and could perhaps be written or organized in a better way. More importantly, the vision for the future of the project has taken ambitious new directions. Finally, having the game editor be a Windows desktop application only is very limiting. Making it web-based is far more ideal. These three factors have led to the realization that it would be much better to start the code base over from scratch and completely redo everything.

Throughout this thesis, it is important to distinguish between the two types of users of ARLogo: those using the game editor and those using the game designer. As such, we define the two following terms:

- Game Designer - user building a game using the ARLogo game editor.
- Game Player - user playing a game using the ARLogo game client.

1.1 Motivation

The ARLogo project looks to be headed in an ambitious yet exciting direction. This thesis serves to lay the groundwork for one particularly important component of the ARLogo project: data layers. A data layer is an underlying feature of the game map which the game designer is able to specify and which the game player can not directly see but can instead sample and perform various other actions which effect the data layer. An example of this is an oil spill in nearby water that seeps into the ground.

In order to motivate the underlying need for data layers at all, one should consider possible scenarios that a game designer might want to build into a game. Here are a
few key examples:

- Mosquitos in the Park - Everyone at the local park has recently started to notice an increase in the number of mosquitos around. The goal here is to figure out how to effectively combat mosquitos.

- Oil Leakage - In the middle of the grasslands area of a town, an oil barrel accidentally spilled. The oil is now slowly spreading radially outward from the initial point of spillage. The town would like to be able to clean up the entire oil spill or at least important patches of it that might infect the water supply.

- Water Stream - In continuing with the oil leakage example, there are underground water streams in the town that are important to the town’s water supply. Finding where these streams are and how they flow is important.

In each of these games, the game designer would like for the game player to figure out some underlying property of the map through an indirect means. Then, presumably, the game player should have some way to effect this underlying property in order to obtain some goal. For example, with the mosquito example, the game player should be able to remove a standing water source or in the oil leakage example, the game player should be able to dig out dirt that has been contaminated.

In general, creating a mechanism for allowing all of these scenarios (and certainly more) to be created by the game designer is the exact problem that data layers seek out to solve.

The difficulty with implementing data layers lies in the balance of giving the game designer control over the data layers while maintaining a clean and easily usable interface. The main goal is to create an environment that naturally encourages learning. Hence, making both the editor and client usable is essential.

1.2 Summary of Thesis Content

Here is a quick summary of the chapters in this thesis:
• **Chapter 2** describes the previous work on AR games that the STEP lab has done.

• **Chapter 3** describes the data layer editor prototype that I built along with suggestions for future work.

• **Chapter 4** describes how data layers should be implemented in the smartphone software.

• **Chapter 5** concludes the thesis and gives directions for future work.
Chapter 2

Background

This section describes the background necessary to understand the foundations of my Masters thesis. In particular, the recent history of AR games and the fundamentals of blocks-based programming will be explored.

2.1 Augmented Reality Games

In the context of the STEP lab, Augmented Reality games are games that utilize the capabilities of GPS-enabled mobile devices to allow the game player to explore a virtual world that is built upon the actual geographical location of the game. Thus, an AR game is a game where the setting of the game is augmented through the use of technology. The types of AR games which can be made vary widely from single player to multiplayer and across all academic disciplines [10]. The key component that dictates the range of AR games that can be built and played is the specific devices and software used to play the AR games. The world of mobile devices has been advancing at phenomenal rates over the past five years. As such, the STEP lab has been able to continuously increase the sophistication of its software from version to version of its software. In the early days, the game client ran on Windows Mobile device which had relatively limited computing power and speed. Additionally, the mobile networks were nowhere near where they currently are these days with the advent of 3G and 4G networks. In other iterations of the AR software, iPhone and
Figure 2-1: Screenshot of a Windows Mobile screen during the Environmental Detectives AR game. The player is currently in range to interview a character. Notice that the map is an overhead view of MIT which is where the game takes place.

Android devices were also supported. Ideally, the AR software would not be limited to one specific device. Since we’re still in the early days of mobile devices, this has not been an easily realizable goal. Developing for multiple platforms has generally been a costly and timely task. Recent years have seen many innovations leading to easier cross-platform development. Additionally, recent years have certainly seen huge increases in the computing power of mobile devices. The divide between the power of mobile devices and the power of laptops is rapidly closing.

The AR software in the STEP lab has gone through a couple of different stages thus far. Initially the AR games were hard coded and run using the game client. From there, the need for the game editor became evident. So far there have been three major stages of the AR software:

- OutdoorAR - Initial iteration of AR software that included the game editor. This product lasted for about three years. The game client ran on Windows Mobile devices and the game editor ran on PC’s. As this product advanced, it began to support the creation of increasingly more complex AR games.

- MITAR - The major push in this iteration of the software was to get students
to design AR games. As such, the user interface required a lot more careful consideration. A lot of advanced features were axed. Notably, this iteration of the software included a central server which allowed students and teachers in a given classroom the ability to work together more easily.

- ARLogo - The current stage of project focused on increasing the range of games that students can create. This version of the project will be completely web-based and looks to incorporate blocks-based programming into the game editor heavily.

To understand the motivations for the history of the AR games going from OutdoorAR to MITAR and now on to ARLogo, we need to look back at the project. One of the first AR games developed (in the pre-OutdoorAR era) was called Environmental Detectives. The target audience was middle/high school students and the idea was that a toxic spill had taken place which was then up to the students to discern the cause as well as develop a remediation plan. Students were taken to the MIT Stata Center where they received a video briefing from the MIT President briefing them about the chemical spill and laying out the objectives of the game. The students were then equipped with mobile devices and let loose on MIT campus searching for clues to help them uncover the underlying cause of the spill and a way to approach resolving it. Their mobile devices gave them the ability to interview virtual characters as well as the ability to take readings of the chemical levels of the water at various locations.

Regarding the development of Environmental Detectives, there was a seven phase process which has been laid out in [9]:

1. Brainstorming new genres of educational software.
2. Designing the first instantiation.
3. Developing a first generation prototype.
4. Classroom field trials.
5. Classroom implementations.
6. Expanding into new contexts.

7. Customized dynamic events.

The main result of the first phase was to leverage mobile devices in a unique way by engaging the students with a narrative based game that took advantage of the actual location of where the game was taking place. The scenario in Environmental Detectives is an example of only one of a handful of possible different scenarios which can be made using the AR game editor. Other categories which these games have covered include history, chemistry, and health.

In phase four, the initial trials showed general interest and approval from the students. They were generally intrigued which is half the battle. The other half is to ensure that they are learning what they are supposed to. This was less successful as none of the students adequately assessed the cause of the toxin spill and there was no indication that they were learning the science involved in the game. One problem was that teams oftentimes reverted to approaching the game as a scavenger hunt trying to make it to each person on the map to interview. Instead, the goal is for the students to critically evaluate what they learn and allocate their time and resources wisely given the information they have.

In phase five, measures were taken to improve upon the inadequacies found in the previous phase. In addition to just general cleanup, which was required since the version of the software in phase four still had rough edges, information was laid out more effectively to allow students to be able to analyze the situation with more ease. Additionally, events were changed so that they were timed. In the first version you could visit any virtual character at any time. In this version, time dependencies could be specified which reduced the scavenger hunt qualities of the game. This version of the game saw more success with students being able to discern the cause of the spill. Their emersion in the science and ability to provide a reasonable future course of action were still lacking however.

During the sixth phase, the need for an ability for users to easily program or configure AR games themselves became apparent. Not only was it a natural extension
to allow for custom functionality but it also became apparent that translating a given game to a different location with a different audience required a number of adjustments regarding the terrain, the age group of the desired audience, and so forth.

Combined, the sixth and seventh phases exposed some of the desired features which led for a call for a revamped AR infrastructure and thus MITAR. In the seventh phase, the fact that many games boiled down to a scavenger hunt was revisited. To remedy this, three features were proposed:

- **Time dependence** - Characters should be able to change what they say over time. For example, at the beginning of the game they might not have useful information while at the end they could say that they have learned something over the course of the game.

- **Cascading events** - The game should allow for certain events to not occur until other events occur. Not only does this control the flow of the game but it also allows for a variable experience for different players if they perform actions in different orders.

- **Distinct roles** - Giving players the option of taking on different roles furthers the variable experience of students. It further introduces the possible aspect of communicating and bargaining with other players.

MITAR was the next version of the product which focused on making the game editor more accessible to students. As such, many of the advanced features were lost along the way (time constraints were also a factor in axing certain advanced features). As the project progressed, the goal of allowing students to use the game editor easily had clearly been accomplished. It soon became evident, however, that the types of games that were being built using the game editor software was limited to mainly just flat, static, scavenger hunt games. Game designers could place characters on the map and give them dialogs, but there wasn’t any way to have them change what they said based on dynamic game factors. While it was great to have students
building their own games, the STEP lab very quickly desired to revamp the product and create a game editor that allows the game designer to have much more control over the gameplay. In particular, the game designer should be able to program the characters on the map to move, have them say various things, and vary other factors in the game dynamically. One approach to this is to give the game designer a drop down menu of options for how to specify the dynamic behavior. However, this is very restrictive and cumbersome. Instead, allowing the game designer the ability to enter code dictating the dynamic behavior of the game would be a power solution to this problem. However, requiring that students write code using a specific language and development environment completely destroys the usability of the game editor. However, it turns that there’s exists blocks-based programming languages which solve the problem of programming languages being hard to pick up and understand.

2.2 Blocks-Based Programming

The goal of blocks-based programming languages is to create a programming language that is easy and painless for beginners to use. To do this, it eliminates many of the traditional complications with learning and using traditional programming languages/environments including:

- The overhead with having to install a potentially large number of software files, configuring them, and installing them properly. This process rarely works out easily since most programming language come fully equipped with a lot of functionality that beginners simply don’t need yet.

- Cryptic or unfamiliar syntax. For various reasons, programming languages have hard-to-understand syntax. They’re often complicated by the use of parentheses, curly brackets, etc. While all of these serve important functions and are often very worthwhile for an experienced developer, they unnecessarily complicate the amount of irrelevant things that a beginner needs to learn about.

- Frustrating and cryptic error messages. Debugging is a hard process even for
experienced programmers, much less beginners. Blocks-based programming languages look to cut out this painful process.

Blocks-based programming languages work by letting the user write programs by dragging and dropping building blocks into place. The user doesn’t have to worry about syntax errors since the programming language takes care of this for the user. The syntax is also generally very easy to understand. Finally, installation is just downloading and installing one program. Blocks-based programming languages have been shown to be very effective at helping students learn and use programming languages[4, 5].

2.2.1 StarLogo

One specific example of a block based, programming language is StarLogo which is also developed in the STEP lab. The StarLogo project has two primary focuses: the simulation of systems and the building/playing of games. As such, the three primary uses it has are introducing students to programming, helping students visualize complex systems (e.g. the diffusion of a gas across a membrane), and introducing students to the art and science of building games. Originally called StarLogo, the newest version has been termed StarLogo: The Next Generation or StarLogo TNG for short. The major goal of StarLogo TNG is to give students a more satisfying game building experience [8]. Specifically, the games it supports building are 3-D and allow characters to explore a scrolling landscape such as Mario. In fact, one of the first programs written by the StarLogo developers to exemplify StarLogo was a Mario based game.

StarLogo TNG has been proven to be a good, simple way to learn programming. Students and teachers responded positively to it and it has gained considerable popularity. One of the clear ways to make the process of learning how to program even easier is by removing the overhead of downloading software and instead having all the tools web based. Furthermore, network connectivity for the games would be ideal to allow users to interact with other users anywhere at any time.
2.3 ARLogo

The ARLogo project was originally conceived as a merger of both the AR and StarLogo projects. After much deliberation, it became clear that the two projects should remain separate. Each project has distinct enough goals and projected directions that it would be too hard and unnatural to combine the two completely. The name ARLogo, however, has still stuck as the name as the next version of the AR project. StarLogo has instead moved on to WebLogo which brings their project into the area of web-based applications. Nonetheless ARLogo is still heavily influenced by StarLogo and continues to remain in constant communication with them.

The major components of ARLogo are the editor software, the mobile software, and the game architecture. The editor software will be browser based to eliminate the installation process and allow it to be accessed anywhere. The mobile software will be focused toward Android devices. It mainly consists of the application which allows the user to play the game and communicate with the game server. (The game server is the core structure which will allow multiplayer games to be run simultaneously as
well as enable the ability to store centrally games, game states, user preferences, and so forth.)

2.4 Substances (i.e. Data Layers 1.0)

Data layers are a new concept allowing us to generalize the idea of the oil spills that we saw in Environmental Detectives. In the OutdoorAR project, oil spills were implemented using what was known as a “substance”. Game designers were allowed to specify basic properties of substances such as their shape, color, and how students playing the game could interact with them. In order to build our vision for better, more powerful data layers, it is important to fully understand the initial idea and implementation of substances, where they fell short, and where they succeeded. There are two components to analyze: the substance editor that the students building the game used to create and integrate substances into their games and the substance test method (i.e. sampling) mechanism which the game players used to help figure out where the substance was, its general shape across the map, and so forth.

2.4.1 Substance Editor

The substance editor is what the game designer interacts with in order to specify spills in their games and modify the properties of the spills. First, the game designer is able to create a substance as an arbitrary ellipse. Then she can choose the two foci of the ellipse and the AR software creates an ellipse from there. Once the ellipse is drawn, the game designer is able to drag the foci around in order to make the substance look like what they envision. This functionality is shown in figure 2-3. This allows for a relatively simple and completely constrained way for students to create the basic shape of the substance.

Once the shape of the substance has been specified, the game designer is able to specify a handful of specific properties:

- Substance Name - the name of the substance, e.g. Oil Spill or Pesticide.
Figure 2-3: Here is a screen capture of what the oil spill editor looks like. The user has just picked two points which specify the oil spill. The oil spill is modeled as an ellipse and the two chosen points are the foci of this ellipse.

- Acquisition Time - the amount of time between when a student playing the game samples the substance at a position and when the game returns the value of that sample.

- Display Color - the color that the substance shows up as.

- Accessible To - a list of all the game player roles that are able to sample this particular substance.

- Test Methods - ways that students playing the game can obtain samples of the substance at specific positions in the game. Test methods will be explained in more detail shortly.

The user interface for editing substances is shown in figure 2-4. There's not much the user can change beyond the size and shape of the substance. This provides the game designer with a very easy to use element to place into games. On the other hand, it very much constrains the creativity of the game. One of the major goals
of this thesis is to propose a way to greatly expand this notion of substances into something that the game designer has more control over. While doing so, we seek to keep the user interface as intuitive as possible.

![Substances screenshot](image)

Figure 2-4: Here is a screenshot of all of the substances in a particular game (currently there is just one, “toxin1”) and the way the user can edit the properties of the substance including the test methods associated with it.

### 2.4.2 Testing Methods

The other important component of the substances that game designers were able to add to AR games previously is the ability to add test methods. A test method is a way that a game player can sample the substance at a particular location. As seen in figure 2-4, there can be many test methods for each substance. Each test method has the following properties:

- Name - the name of the test method.

- Processing Time - the amount of time it takes for the game player to get a reading back when sampling.

- % - the maximum percentage error in the reading, i.e. when a player samples in an actual game, the reading can be in error by up to this percentage.
Accessible To - the particular roles that this testing method is accessible to (this is a subset of the roles associated with the substance).

The dialog a game designer sees when adding or editing a test method is shown in figure 2-5.

![Figure 2-5: Shown here is a screenshot of the form a game designer sees when adding or editing a test method associated with a substance.](image)
Chapter 3

Data Layer Editor

At the core of my thesis is a working software prototype of the data layer editor software.

3.1 Overview

The overall idea is for the student building the game (the game designer) to login to our website. From there they view their own games, other students’ games, etc. In particular, they are able to create and edit their own AR games online. The overall editor will allow the game designer to modify the game in a number of ways. Here we look at the functionality for creating and modifying data layers. This is the heart of my thesis work.

The prototype for the data layer editor was made using HTML5, JavaScript, jQuery, and PHP. To view the prototype one must use a browser which supports the HTML5 canvas tag (Chrome and Firefox are recommended). The code for the data layer editor was written with the intent that software developers would have to edit, modify, and integrate the current code into the larger ARLogo project. As such, an effort to make the code clean, commented, readable, and modular was made.

The choice of using HTML5[14, 12] required quite a bit of research and thought. The major requirement that ARLogo called for was the ability to display the grid, data layers, and programming blocks effectively. Standard HTML/JavaScript/CSS
is not suited to this purpose. The decision was thus between HTML5 and Adobe Flex[15]. The main tradeoff between the two is that HTML5 is likely to be the way of the future[13] while Adobe Flex is tried and proven. The question was really whether or not HTML5 and its associated tools would develop fast enough for the needs of ARLogo. One of the deciding factors was the rapid decline of Adobe Flex/Flash. The perhaps biggest hit to Adobe was dealt by Apple when they banned the use of Adobe Flash on iOS devices[1]. This directly affected us as we were looking to deploy our mobile client to as many devices as possible with Android and the iPhone being at the top of our list. HTML5 on the other hand looks promising. As of May 2010, 46% of people were using HTML5-compatible web browsers[11]. The largest portion of users without a compatible browser were running either IE6, IE7, or IE8. IE9, which is HTML5-compatible, was not out at the time that this poll was taken. Thus, it should be fair to say that the majority of users online use HTML5-compatible browsers.

While not incorporated into my prototype, one major decision of the ARLogo project is the use of Google's Closure JavaScript suite[6]. One of the weaknesses of using HTML/CSS/JavaScript is that JavaScript is not a very robust language. While libraries such as jQuery take on a lot of the pains with using traditional JavaScript, e.g. cross-browser compatibility and UI widgets, they do little to improve some of the more basic weaknesses of JavaScript as a language. JavaScript's major weaknesses are rooted in its origins. It was made for the simple requirements of the web from over a decade ago. AJAX was not around yet and the major use of JavaScript was simply for small animations, popup screens, etc. As the internet sees more and more large-scale web applications, the need for a robust programming environment is critical. That said, JavaScript is not completely unsalvageable (Google Closure does indeed attempt to salvage it). Besides the bad components, there are plenty of good features of the language. A good analysis of both sides of JavaScript can be found in [3]. Google Closure helps this cause by offering the following features which addresses many of the pain points of JavaScript[2]:

- Closure compiler - optimizes the JavaScript both in terms of the size of the code (it’s important to have compact JavaScript code to reduce page load times) and
runtime.

- Typing - maintaining large software systems is made even more difficult without strict type checking. JavaScript is a dynamically typed language but Closure adds type checking to make code more maintainable.

- Comprehensive library - this adds methods which take care of cross-browser compatibility and gives the programmer UI widgets.

- Templating - allows the developer to create reusable templates thus making code more manageable.

- JavaScript Style Checker - checks that the code adheres to common coding standards.

Since ARLogo will be a large-scale, complex web application, having all of Google Closure’s resources available will make the project much easier to maintain and build upon. While this is in general a good idea, it is particularly useful for the STEP lab as there is often a quick turnover of programmers as UROP’s/MEng’s come and go.

Now, getting on to my area of work on the project, data layers, there are a few important properties of a data layer:

- Grid size - 100 units × 100 units. The data layer is represented as a grid of cells each of which has a value corresponding to the intensity of the data layer at that point. The size of this grid is fixed for the purposes of the AR games. The data layer editor code can be easily changed to handle other sizes by changing a few parameters.

- Max value - This is set by the game designer (defaults to 100). In order to allow the data layer to model various scenarios more realistically, the game builder needs to be able to set the range of values the data layer contains.

- Min value - This is set by the game designer (defaults to 0).

The main screen of the data layer editor prototype is shown in figure 3-1.
Figure 3-1: Main screen of the data layer editor.
3.2 Initial Prototype

In the following subsections we walk through all of the functionality of the prototype with an emphasis on the motivation behind the features and how they were implemented. In the next section, we analyze the decisions made and lay out suggested changes that should be made to the prototype.

3.2.1 Creating and Initializing a Data Layer

The first thing that the game builder will want to do is create a data layer. He will also want to specify the starting state of the data. Making the initialization of data layers quick and easy is a UI challenge. We believe that allowing the game builder the ability to initialize a data layer from an images, from points, and from a function allow for a wide range of scenarios and build a solid foundation for future work. The screen for creating and initializing a data layer is shown in figure 3-2.
Figure 3-3: Example of initializing a data layer from an image. The original image is on the left while the data layer is shown on the right.

From Image

The first method of initializing a data layer is from an image. A data layer is represented as a grid of single values while a color image is a grid with three dimensions (red, green, and blue). In order to account for this, the server processes the image the game builder uploads by computing the grayscale of it and using the grayscale values to create the data layer. An example of this is shown in figure 3-3.

From Points

The second method of initializing a data layer is to specifying some number of circular points. The points are specified as a triple \((x, y, r)\) of \(x\)-coordinate, \(y\)-coordinate, and radius. The idea behind this method of initialization is that it gives the game builder a quick and simple way to specify a data layer. A lot of natural scenarios can be modeled decently well using these points as an initial state. In the current prototype the game builder needs to input the points manually, but if this method becomes very popular it would make sense to expand the functionality to more than just circles and to make the interface easier to use by allowing the game builder to simply draw shapes on the data grid. In figure 3-4 we see an example of a data layer which was initialized using circles at \((50, 15)\) and \((50, 75)\) with radii of 10 and 20, respectively.
Figure 3-4: Example of initializing a data layer from points. The points input were located at (50, 15) and (50, 75) and have radii of length 10 and 20, respectively.

Figure 3-5: Example of initializing a data layer from a function. The function used is $f(x, y) = \frac{1}{2} \cdot (x + y)$.

From Function

The final option implemented in the prototype is to allow the game designer to specify a function in terms of $x$ and $y$ to populate the initial values for the data layer. In its current implementation, this tends to be the hardest approach to achieve a specific result. Nonetheless, it’s a powerful method with a lot of room for improvement. Shown in figure 3-5 is an example of a data layer initialized using the function $f(x, y) = \frac{1}{2} \cdot (x + y)$. 
3.2.2 Specifying Behavior (aka the onTick Event)

Once the initial state of the data layer is specified, the next important task for the game designer is to define how the data layer changes over time. While we could give the game designer a small list of options that would cover a lot of scenarios, we feel that this is not scalable nor the most instructive way to approach this. Not only do we think it would be exciting to see what possible scenarios students concoct but by making the students create their own games at this level, we inject more educational value into the process. In particular, the game designer will be required to write the onTick function in code. The problem here is that only students that know the particular programming language we support can create games. Learning a programming language is not a small task and often ridden with many complications. To avoid many of these complications and to make allow as many students as possible to design games, we will use a block-based language for the coding.

The challenge with the block-based language is the question of what blocks should come standard. Clearly the designer should have access to basic blocks such as arithmetic operations, constants, logical statements, and loops. Beyond that, in figure 3.1 we specify a handful of prebuilt blocks that we believe the designer would find very useful.

The software prototype I created does not contain a block-based editor but rather has a simple text-based editor. This allows for demos of creating various onTick events. An example of an onTick function is shown in figure 3-6. The way that the onTick event works in the software prototypes is that the designer assumes she’s given \( x, y, grid, \) and \( grid\_copy \) variables. She specifies a function in terms of these and is able to modify the values in \( grid \) which will end up being the resulting value of the data layer after the onTick event has been run for all \( x \) and \( y \) combinations in the data layer.
Table 3.1: Summary of suggested blocks to be given to specify the onTick event.

<table>
<thead>
<tr>
<th>Name</th>
<th>Input(s)</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>diffuse</td>
<td>pctLost</td>
<td>Each cell transfers an equal amount to each of its 8 adjacent neighbor cells. The input pctLost specifies the total amount transferred.</td>
</tr>
<tr>
<td>diffuse4</td>
<td>pctLost</td>
<td>The same as diffuse but each cell only transfer to the 4 cells directly touching it.</td>
</tr>
<tr>
<td>diffuseP</td>
<td>pctLost, p</td>
<td>The same as diffuse but each cell transfers its value to each of the 8 adjacent cells with probability p.</td>
</tr>
<tr>
<td>diffuse4P</td>
<td>pctLost, p</td>
<td>The same as diffuse4 but each cell transfers its value to each of the 4 cell directly touching it with probability p.</td>
</tr>
<tr>
<td>diffuseTowardPoint</td>
<td>pctLost, x, y</td>
<td>Similar to diffuse but cells only transfer value to cells that are in the direction of the point ((x, y)).</td>
</tr>
<tr>
<td>diffuseTowardPoint4,</td>
<td>-</td>
<td>These all are similar to diffuseTowardPoint with the corresponding twists described at the beginning of this table.</td>
</tr>
<tr>
<td>diffuseTowardPointP,</td>
<td></td>
<td></td>
</tr>
<tr>
<td>diffuseTowardPoint4P</td>
<td></td>
<td></td>
</tr>
<tr>
<td>moveTowardPoint</td>
<td>x, y</td>
<td>Each cell transfers its value to the cell in the direction of point ((x, y)).</td>
</tr>
<tr>
<td>moveTowardPoint4,</td>
<td>-</td>
<td>These all are similar to moveTowardPoint with the corresponding twists described at the beginning of this table.</td>
</tr>
<tr>
<td>moveTowardPointP,</td>
<td></td>
<td></td>
</tr>
<tr>
<td>moveTowardPoint4P,</td>
<td></td>
<td></td>
</tr>
<tr>
<td>nsum</td>
<td>none</td>
<td>Each cell sets its value to the sum of the the values of all 8 cells adjacent to it.</td>
</tr>
<tr>
<td>nsum4, nsumP, nsum4P</td>
<td>-</td>
<td>These are all similar to nsum with the corresponding twists described at the beginning of this table.</td>
</tr>
</tbody>
</table>
3.2.3 Editing Properties of a Data Layer

There are a few basic properties of the data layer that the designer should be able to specify. These properties are mostly an issue of aesthetics when considering the experience of the player. The properties allowed to be changed in the software prototype are listed in figure 3.2 and a screenshot of the interface for changing these properties is shown in figure 3-7.

3.2.4 Emulating Data Layer

An important feature for the designer is being able to run the onTick event a number of times to see how the data layer actually evolves over time. The software prototype contains functionality for this. The designer can run either one step at a time “run a while” which will run 20 steps at a time showing each step as it goes. An example of an initial data layer and its result after 10 time units is shown in figure 3-8. The data layer was created from the point (50, 50, 10) and was provided with the onTick function shown in figure 3-6.
Table 3.2: Summary of suggested blocks to be given to specify the onTick event.

<table>
<thead>
<tr>
<th>Property</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Name</td>
<td>The name of the data layer.</td>
</tr>
<tr>
<td>Min Data Value</td>
<td>The minimum value that a cell in the data layer can be assigned.</td>
</tr>
<tr>
<td>Max Data Value</td>
<td>The maximum value that a cell in the data layer can be assigned.</td>
</tr>
<tr>
<td>Color</td>
<td>The color that a cell with the max data value gets colored with. Cells with the min data value are colored with white and all other cells are colored using a linear scaling of colors between white and the max data value color.</td>
</tr>
</tbody>
</table>

Figure 3-7: Screenshot of the edit preferences interface.
3.2.5 Toggle Map

One additional feature that comes in handy for the designer is being able to see what the data layer looks like when it is overlaid onto the game map. It’s important that the important sections of the data layer actually line up with their intended places on the grid. To achieve this, the ability to toggle the game map on and off is included in the software prototype. An example of the game map being toggled on is shown in figure 3-9.
3.3 Analysis

My thesis project serves a two-fold purpose. First, it lays a foundation for the data layer editor portion of ARLogo with a lot of the key features in place. The second purpose addresses the issue that since the goals of the ARLogo project are lofty, it’s hard to discern how well we are actually able to create the experience that we envision in software. The prototype of the data layer editor gives a glimpse of what is actually possible and provides more information about the exact directions that would actually be best to pursue.

In order to get some indication of how well this prototype will work for our intended purposes, we should consider the three examples of games one might want to build posed in section 1.1: mosquitos in the park, oil leakage, and a water stream.

3.3.1 Mosquitos in the Park

In order to model this, one straightforward approach would be to have two data layers: a mosquito layer and a standing water layer.

The standing water layer can be specified using the “from points” method since standing water usually forms in pools which are approximately circular.

The mosquito layer can be specified in a number of ways. Let’s just say they’re spread out randomly across the map to start which we can initialize using the “from function” method.

The onTick event for the water layer could just be that it stays the same. It’s a bit trickier for the mosquito layer where the change in values should depend on the standing water data layer. In the current version of the prototype this is not possible, i.e. it’s not possible to access other data layer values. This should be changed since this is clearly a useful example. If we’re given the ability to access other data layers, then the mosquitos onTick could be to move toward the pools of standing water.

In the next section, we’ll see that one of the features that should be added to the AR games are this notion of “data layer items”. In this game, the game designer might want to add bug spray which would repel mosquitos, a mop to clean up standing
water, or a bug zapper which would kill mosquitoes. Each of these would be easily modeled using data layer items.

Thus, with just the addition of being able to access other data layers in the same game, this scenario should be easily attainable. Even without being able to access other data layers, this game could be made by hardcoding the positions of where the standing water puddles are in the onTick function for the mosquitoes. This is clearly not ideal, however.

### 3.3.2 Oil Leakage

In the oil leakage example, the only data layer that needs to be specified is an oil data layer which can be initialized using the “from points” functionality since the oil barrels are relatively compact and contained in a small area.

In order to specify the onTick event, having the oil diffuse away from the barrel which they started at should be sufficient to model this.

One data layer item that might be useful is a shovel truck which would pick up the barrel and move it someplace safe.

Thus, this scenario seems to be pretty simple and easily taken care of by my prototype.

### 3.3.3 Water Stream

In the water stream example, let’s say we have a single source of water and a single sink. The water flows from the source to the sink. The only data layer we need in this example is one for the water.

We can again use the “from points” method to make the initial data layer be a large pool of water located where the source is. Then the onTick event could just have the water move toward the sink point.
3.3.4 Conclusion

Each of these three scenarios was easily covered by the prototype with only a small number of adjustments which were already on the list to be added. Under this metric, the proposed model and proposed blocks seems to perform well in terms of whether or not it's expressive enough.

All of the specified blocks in table 3.1 might not be necessary. In these cases we only used the moveToPoint block and the diffuse block. Perhaps the nsum block is not completely essential.

Additionally, the "from points" initialization method seems to be pretty expressive at least to give decent approximations to good starting points. However, the "from image" method could also easily be used for any of these three scenarios to provide a more precise starting arrangement. The "from function" method was used once, to specify a random initial arrangement. Other instances where "from function" might make sense are when the grid needs to start with some uniform or relatively uniform distribution which is easier to express as a mathematical function. This doesn't seem like it would be an extremely common feature but it still serves a useful purpose.

Thus, this model seems to serve a solid foundation on which a good assortment of games can be built just as long as the ability to access other data layers in the same game is implemented.

3.4 Future Work

Based on the analysis done in the previous section and careful thought, this section provides a handful of ideas for future work.

In general, the desired functions that should be at the hands of the game designer should be kept in mind. These include:

- Creating data layers.

- Initializing the data layer values. This can be a complex operation for the game designer.
• Setting/editing basic properties of data layers, e.g. name, color, min/max data value.

• Setting the onTick behavior. This is the most important and technically challenging part of creating the editor interface. Challenges include:
  – Creating the blocks interface.
  – Giving the game designer a good set of blocks to start with.
  – Letting the game designer create reusable modules.
  – Letting various game designers share modules.

• Being able to get an indication of what game players will see. Being able to emulate the onTick event is a start, but something else that would be useful is for the game designer to be able to emulate actually playing a game and seeing what happens without actually going through the usual process of doing so.

• Adding ways for game players to affect the data layer.

3.4.1 Creating and Initializing a Data Layer

Initializing a data layer is one of the two main challenges in balancing the usability of our software while giving the designer more power. There are clear directions to proceed in to improve both the usability of the current design as well as the power of the designer.

From Image

Initializing from an image works as expected and there’s no clear way or even reason to try and improve upon it.

From Points

Initializing from points has a lot of room for expansion. First, it would improve the usability to show the designer a preview of the points before she has to commit to
those points. The initial approach of inputting a list of points was used for sheer simplicity on the coding side. From a UI perspective, there is a lot to be gained.

Going even further, making the interface completely interactive could provide an even better experience for the designer. That is, allowing the designer to click, drag, and drop directly on the map to create and edit shapes would increase the power and usability of this feature.

Another extension would be to allow for different kinds of shapes to be drawn as part of the standard tools given to the designer. Support for polygons would be the most useful to start with.

Continuing down this path of extending the functionality of our editor would bring us closer and closer to the realm of having a full-on browser-based image editing system. There’s nothing necessarily wrong with this except that designers are able to use pre-existing image editing programs to create pictures which they could then upload using our upload feature. Thus, expending a lot of energy increasing the feature set of the data layer editor in this direction should be low on the priority list.

**From Function**

The relation between entering in a function and it showing up as the data layer is rather mystic in the current prototype. There are a number of things that can improve this:

- Labeling the axes of the data grid as $x$-axis and $y$-axis.

- Making $(0,0)$ be at the bottom-left instead of the top-left as it currently is and having the $x$-axis span horizontally while the $y$-axis would span vertically.

- Allowing the designer to preview what a function looks like on the grid before committing to it.

All of these would help to increase the association of our editor with concepts that the user should be familiar with.
3.4.2 Specifying Behavior (aka the onTick event)

There is clearly a lot of work left here with creating and integrating the block-based programming interface. The question of which blocks should be given to the designer will be answered iteratively. The current set of proposed blocks should allow for quite a bit of creativity. After seeing how designers use the blocks and using their feedback to see what they really would like to make, blocks can be added or removed.

Additionally, allowing the game designer to access other data layers (instead of just the current one) is important for creating complex games.

3.4.3 Emulating Data Layer

Emulating the data layer works basically as expected. The only major addition would be to allow the designer to go back in time steps. A straightforward approach would be to keep the state of the data layer at all time steps in memory which would cause a memory nightmare. Instead, a better approach would be to keep in memory only a few of the intermediate states of the data layer along with how many onTick events occurred to get to those states. Then in order to go back in time steps, you can run the onTick event on the closest saved grid state until you get to the one you want.

3.4.4 Data Layer Items

Finally, one essential area of future work is to implement the feature allowing the designer to add what we will call “data layer items” to his games. A data layer item is something that the player will have access to that effects the data layer. For example, if the data layer in question is an oil spill, the designer might want to add a pump that players can use to reduce the amount of oil in an area. Or maybe the designer wants to allow the game to have adversaries that can place additional oil barrels on the map thus increasing the amount of oil in the data layer.

In either case, the designer must be able to specify the following with regards to a data layer item:

- Name of item.
• How many time units the item is active for.

• The net effect of the item each time step.

• Which players in the game can pickup or use the item.

In keeping consistent with the rest of the editor interface, it seems most natural to allow the designer to specify the net effect of the item each time step as an additional onTick which gets run after the usual onTick event. This cuts down on the number of different interfaces the designer needs to learn in order to create a game.

While the notion of having items that the game players have to pick up seems natural and makes sense, there is also the possibility that the game designer will want to give the game player ways to perform actions on the data layer without having to obtain some item. This notion unfortunately does not fit in line with the notion of data layer items. This can be accommodated in a few straightforward ways:

• Allow items to be have a property specifying what “type” of object they are, e.g. inventory objects and ubiquitous objects. The inventory objects would be the ones that need to be picked up while the ubiquitous ones could be defined to be usable anywhere and are not stored in a user’s inventory.

• Create a separate notion of a “data layer action” which would just allow for actions that game players can perform anywhere. This could be added in addition to data layer items or instead of them depending on what seems most ideal.

• Have just data layer actions and no items. The actions could then be made to incorporate the items as well although this would make the term “data layer action” feel a bit unnatural.

3.4.5 Paper Prototypes

To give a better picture of the intended suggestions for future work, paper prototypes are shown in the figures below. These prototypes have not yet been created in code.
The main goal with these prototypes is to make the data layer editor as easily usable as possible. Beginners should not have to struggle endlessly to accomplish even the most basic of tasks. If a user is unsure of something, there should be a way for him/her to test out what it does quickly, easily, and without fear of irreversibly messing up the state of their game. When possible, the user should not have to navigate away to various screens. All of these ideas come out of basic UI theory. The most important fact of UI is that no matter what, you must user test. Even if a design makes sense in theory, the user might not see it the same way. In Appendix B, a sample user test script is given which can be used as-is or built upon. The captions below the figures describe why certain choices were made.
Create New Layer
- From FWMFU
- From RM IMOP
- From Image

Specify Points
- From List
- Pick Manually

Specify Function
- From Points
- From Function
- From Image

Upload Image
- From Points
- From Function
- From Image

Figure 3-10: Prototype of wizard dialog for creating a data layer. By employing a wizard process to create a data layer, the user will be guided through the process. Each step in the process should be easily manageable. Furthermore, it leaves less room for error as the path to creating a data layer is very constrained and clear.

Input List of Points

Points:

Submit Preview

Figure 3-11: The rest of the prototype of the wizard dialog for creating a data layer. This shows the options for creating a data layer “from points”. The “preview” feature is proposed to be added since it is very important for a user to feel comfortable with their choice. It also increases efficiency since the user wont have to go back through the entire wizard if he ended up not specifying the right points.
Figure 3-12: Prototype of the interface for creating a data layer "from points" using mouse click events. By highlighting the selected point, the state of the program is very clearly shown to the user. Also, having the editable properties right below the grid will make it very easy for the user to figure out how to change those properties and will also increase efficiency since there should be instant feedback.
Figure 3-13: Prototype of relatively minor changes to improve the UI of the front screen and to bring the “edit properties” section to the same page as the visualization of the data layer. By moving these properties to the same screen as the grid, as opposed to a different tab, this reduces the possible amount of frustration a user might encounter by trying to find where everything is located in our website.
Figure 3-14: Prototype of the interface to create, edit, and delete data layer items for a given data layer.
Chapter 4

Data Layers in the Game Player

Client

While it's important for the game designer to be able to easily create powerful data layers, it's critical that these data layers are actually integrated smoothly into AR games when they are played on the game client. In this section, I address this issue and give my ideas on how to implement the game playing engine at a user interface level.

4.1 Overview

The software client that the player uses to play AR games runs on mobile devices – both Android devices and the iPhone will be supported. In both cases, the mobile application will be built using HTML, JavaScript, and CSS. Then it will be compiled into a native application using Titanium Appcelerator[7].

Titanium Appcelerator provides a library of JavaScript functions and UI widgets. In addition to making it easy to create an application that looks good on the mobile device, Titanium also provides API's for accessing core functions of the devices such as GPS, local storage, etc. In order for all of this to work, the application cannot simply be viewed as a website. It instead needs to be compiled into a native application. Titanium takes care of this process and outputs a program that can be installed on
the mobile devices. A small number of additional setup and configuration steps need to be taken for each platform that the application is intended to run on. Most, if not all, of the application code should be able to be reused across different devices.

The choice of using Titanium was not a particularly easy one. Making a mobile application that runs on many different devices is not a simple process. The straightforward approach is to program an application for each of the devices separately. This suffers from too many development environments, programming languages, paradigms, conventions, and available features of the device. Of course, this happens to be a well-known problem that mobile developers face. A novel approach to this has been to create a way to compile a given program into code that will natively run on multiple devices. There are a number of companies out there that offer products for this exact purpose: Titanium, Phonegap, and Rhomobile are the top competitors. Some of the major reasons for choosing Titanium over the others are as follows:

- Great support, although it doesn’t come free.
- Supports all the functionality (e.g. GPS) that we desire.
- Titanium seems to be well-funded and poised to be around for a while.
- Solid documentation and a good, easy-to-navigate website.

It should also be noted that our lab tested out both Titanium and Phonegap extensively and found Titanium easier to get started and going with. One of the main downsides of Rhomobile is that it uses Ruby which few people in the lab and at MIT are experienced with. The other two use HTML/CSS/JavaScript which is far more well-known. The added benefit to using this set of technologies is that both our editor and mobile client will be written using the same languages.

The whole game will be run on mobile clients that communicate back to a central server. Considering the setup of the server and mobile clients, the key considerations with regards to data layers are the following:
• The client has GPS access which allows for the position on the map to be known. This is nothing new in the realm of AR.

• It will be possible for games to be played by multiple players using different clients. As such, allowing for such things as sharing sample information will be possible.

• Due to the variability in signal strength, communication to the server is not always available. Thus anything relying on split-second communication between the server and the client at all times is not always possible. Signal outages can occur on the order of 10 minutes if, for example, a game player walks into a parking garage with no signal.

• Offloading intensive calculations to the server is a possibility, if necessary.

As the game players progress throughout the game, taking samples and performing actions using data layer items should be integrated into the games with a focus on being intuitive and consistent with the rest of the game client interface.

4.1.1 Mobile Client

All of the specifics of the mobile client are not necessary to be discussed here. The basics, however, provide a solid background to understand the issues regarding how interacting with data layers should be implemented on the mobile client. To give an idea of what one screen of the mobile client might look like, an example is shown in figure 4-1.

When first starting the mobile client, the major functionality needed by the game player are as follows:

• View available games.

• Load a specific game.

Once a specific game has been loaded, there are a number of important aspects of the game that need to be supported:
Figure 4-1: An example of what the main screen of the mobile client might look like.

- View game map - The game map is one of the core aspects of the game. It's what the game player will see most of the time and dictates a lot about the game. In addition to the image of the geographical map, the map should also display important information such as the locations of characters.

- Move around/explore - When game players moves around in the actual world, the mobile client should update their position on the game map and any location-based events should trigger when applicable.

- Talk to characters - Game players should be able to interact with game characters when applicable.

- Sample data layers / view samples - Game players should be able to sample the data layer and view samples that he/she has taken along with samples that other game players on his/her team have taken.

- Use data layer items - Once a player picks up a data layer item, he/she should be able to use it.
Again, there are certainly more features than just these but these form a good starting point.

4.1.2 Data Layers

There are three main aspects to consider regarding data layers on the mobile client: taking samples, viewing samples that have already been taken, and using data layer items to effect the data layer.

Taking Samples

Taking samples should be a fairly simple process. From the main screen, the game player should be able to click on a button to get to the “take a sample” screen. There should then be an intuitive way to pick what data layer (since there might be multiple ones defined for a game) to sample from and then a button to take the sample.

Viewing Samples

Viewing the samples is not as straightforward as it might sound. The grid is 100 x 100 and the data layer will get updated around once a minute. That’s a lot of data and deciding on how the “samples” work dictates a lot about how usable the “viewing samples” interface will be for the game player. There are a couple of options to consider:

- Single Location, Single Time - For each sample, only show the value of the data layer at the location the data layer is sampled and only at the time it takes place.

- Single Location, All Times - For each sample, only show the value of the data layer at the location the data layer is sampled, but show its value for the entire duration of the game.

- All Locations, Single Time - For each sample, show the value of the data layer for the entire grid, but only show it for the time at which the sample was taken.
• All Locations, All Times - Upon sampling, show the entire data layer at all times.

One can consider extending these such that the game designer has the option to specify some radius (or region) around the sampling location that gets revealed for each sample and for some variable amount of time before and after when the sample occurs. We stick to these four scenarios for simplicity.

The first scenario will probably not be very fun for the game player. Getting only one value of the data layer at one time step when there are $100 \times 100 = 10,000$ locations over a span of at least, say, 10 time units gives the game player very limited information.

On the flip side, the last scenario defeats the purpose of sampling and too easily gives the game player all of the information.

Thus, one of the other two options is preferable (or a mix of the two).

In either case, we will have some number of samples that not only span a two dimensional space, but also span time. Viewing the samples in space is easy to accomplish – just superimpose them on the game map. Viewing them over time is not as straightforward. I propose that some mechanism to play through time should be implemented. One way to do this is with a slider that the game player can move back and forth to visualize the movement of data through time.

It might also make sense for a player to be able to view the samples that other people, e.g. on the same team, have taken. One reason for why this might be useful is to increase the number of samples that one has the ability to see. There are some easy ways to add the ability to see other people’s samples:

• Have a list of checkboxes for each of the other teammates. This would control which samples are being shown.

• Have a couple of tabs, e.g. “my samples” and “everyone’s samples”.

• Show other people’s samples using different colors. This can be done in addition to the previous two ideas.
Another idea that would add even more complexity and power to the AR games would be allowing the game designer to create intricate situations where teammates acting in accord can increase their sampling power more than just the sum of their two sampling ability. For example, it would be interesting to have a scenario where a giant rock can only be moved if at least three players are there.

**Using Data Layer Items**

Data layer items should show up on the map. Once a game player gets close enough to the item, he should be able to pick the item up and then use it whenever either once or multiple times, depending on how the game designer configured it. The button to use a data layer item should be located in one of the menus and when an item is used, this should be recorded and should show up somehow when the game player goes through and views the samples.
Chapter 5

Conclusion and Future Work

In this thesis, all of the relevant topics regarding data layers in the ARLogo project have been discussing and delved into in detail. Much of the functionality for the data layers has been prototyped in code and a lot of the suggested future directions of the work has been paper prototyped. The entire ARLogo project is still, at the time of the writing of this thesis, in its initial development stages. As such, a lot of details regarding the project are subject to change. Keeping this in mind, things that are discussed in this thesis might not be applicable anymore or might be irrelevant at some point in the future. Nonetheless, this thesis serves as a proof of concept of this improved notion and implementation of data layers. It also serves to direct the future work done on ARLogo, especially regarding data layers.

There is plenty of future work to be done. The ARLogo project in itself is a young and thriving project currently being developed in full force. There is much to be done in every aspect of the project – the game editor, mobile client, and central server. Many relatively short-term improvements to my prototype of the data layer have been talked about in chapter 3 of this thesis.

One very important consideration for the future of the project is the specific blocks that are given to the game designer. In particular, the number of blocks, the scope of what the blocks do, and the way that they are displayed to the game designer are important considerations. Not only should the game designer have the power to make complex and interesting games, but the interface for doing so should be intuitive and
usable. Creating a great user experience is not an easy task and requires ample user testing and the ability to incorporate feedback. Thus, part of the future work on this project should be to see how students are actually using the software and iterating.

Likewise, the functionality for game players viewing data layers on the mobile client will certainly get a constant stream of feedback as it gets tested and used. Iterating on the current design of this will certainly be an important task for the future.

Another consideration for future work is the security of the system with respect to running the blocks that the game designer specifies. This should be done in a sandbox environment. Creating a good sandbox, however, is not a trivial problem. While security is not an immediate concern, it should certainly receive attention at some point and will certainly need to be considering if and when the project scales up.
Appendix A

Data Layer Object

The important part of my code that needs to be carried forward is a DataLayer class which was written in JavaScript. This section contains a specification of that class. The DataLayer class takes as inputs three arguments: name, numHorUnits, and numVertUnits. It's unlikely ARLogo will stray from having a different grid size than 100 x 100 but in case they do want to change this, the DataLayer class is designed to be able to do so easily. If either of numHorUnits or numVertUnits is not specified, the constructor of the DataLayer class defaults the grid size back to 100 x 100. The name input to the DataLayer class is simply just the name associated with the DataLayer. If no name is specified, name is defaulted to “Untitled Data Layer” and the game designer would be able to change the name later.

There a number of member properties defined in the DataLayer class as follows:

- grid - Array of arrays representing the underlying data layer values.

- grid_copy - Copy of grid that is used when simulating an onTick event. The values in the new data layer depend on the old data layer. We need to have a copy of the old data layer otherwise we get incorrect results when calling the onTick function.

- name - This stores the name of the data layer.

- onTick - Code that specifies what happens to the data layer after each time
- **id** - ID of the DataLayer object. This is not currently used but is important when the DataLayer code gets integrated into the entire ARLogo project.

- **minDataValue** - Specifies the minimum value that can be present in the data layer. This defaults to 0.

- **maxDataValue** - Specifies the maximum value that can be present in the data layer. This defaults to 100.

- **color** - Hex value representation (e.g. white is #FFFFFF) of the color that is used to represent the data layer. The data layer will portray locations with a value equal to maxDataValue with the color color and a 1.0 alpha value (i.e. no transparency). Locations with minDataValue are also represented with the color color and a 0.0 alpha (i.e. fully transparent). Values in between these two have an alpha value that's scaled linearly and are represented by the color color.

- **numHorUnits** - Number of horizontal grid units in the data layer.

- **numVertUnits** - Number of vertical grid units in the data layer.

- **horUnitSize** - Number of pixels in one horizontal grid unit. This value is currently hardcoded to 6 and cannot be changed by the game designer.

- **vertUnitSize** - Number of pixels in one vertical grid unit. This value is currently hardcoded to 6 and cannot be changed by the game designer.

- **displayWidth** - Horizontal size of the grid in pixels. This equals horUnitSize $\times$ numHorUnits.

- **displayHeight** - Vertical size of the grid in pixels. This equals vertUnitSize $\times$ numVertUnits.

- **canvas** - Variable specifying the HTML5 canvas element that this instance of the DataLayer should be drawn to.
Additionally, a number of methods are defined in the DataLayer class as follows:

- **runOneTick** - Runs the onTick function one time.

- **clearGrid** - Clears the data layer from the associated canvas.

- **importFromImage** - Initializes a data layer given an image as input. The input to this function is assumed to be a 2-dimensional array of HSV values which is given to us from the server after a user uploads or specifies an image.

- **importFromPoints** - Given an input points which is an array of points in the form \([x_1, y_1, r_1], [x_2, y_2, r_2], \ldots\], this initializes a data layer with a circle centered at \((x_i, y_i)\) of radius \(r_i\) for each \(i\) in the input array. These circles are solid circles with the value maxDataValue everywhere in or on the circle.

- **importFromFunction** - Given a function, as a string, this initializes the data layer based on it. The function should take in as arguments \(x\) and \(y\) and return a value corresponding to the value of the data layer at the points \((x, y)\).

- **nsum** - Underlying code for the nsum code block. Sets each value in the data layer to the sum of the 8 squares around it and subtracts that value from those surrounding squares.

- **nsum4** - Same as nsum but it only acts on the 4 directly neighboring squares.

- **diffuse** - Underlying code for the diffuse code block. Takes as input percent which is the amount that each square transfers at each step. At each step, a total of percent percentage of the each value in the data layer is spread among all 8 of its adjacent neighbor squares.

- **diffuse4** - Same as diffuse but it only acts on the 4 directly neighboring squares.

- **setCanvas** - Set’s the canvas property of the current data layer. Takes the canvas as an argument.
- **drawGridToCanvas** - Draws the x-y grid to the canvas.

- **drawDataLayerToCanvas** - Draws the current data layer to the canvas.

- **clearCanvas** - Clears the canvas of all drawing.
Appendix B

User Test Study

B.1 Test User Prompt

We first gave the users an introduction to AR games and what they were about to embark on. Then we asked them to perform various tasks. We watched them perform these tasks and asked for feedback at the end.

B.1.1 Introductory Prompt

This is the prompt that we gave users:

Hello.

We would like you to test out our software today. In particular, we are concerned with how intuitive it is, how easy it is to use, and how relevant the interface is to the tasks we give you. This is an early prototype of our software so there are bound to be many shortcomings on our part. We are not testing how adept you are at figuring out our interface. Any and every problem you encounter when using our software is valuable for us to know as it points to a very specific place in which we can improve the usability of our software. Please understand that this is not a test of your intelligence! While we would be very pleased to see you succeed in all of the tasks we ask you to complete, we’re looking to find the sore spots in
our UI. So please share with us as much as possible about your experience with our software.

One important note is that the software we are having you test is just one component in a much larger system which has yet to be built. As such, you will need to know what the rest of the software will do for everything to make sense. The overall software allows users to create what are called Augmented Reality games. The user starts off by choosing a map [show them an example of a map], designating characters, a plot, and so forth. Once the user has completed making the game, it can be played by a group of other users in a real life environment. The component of the game building process that we’re having you try out today is what we call data layers. Data layers allow the game designer to add additional data to their games. For example, consider a game where a couple of barrels of gasoline have been accidentally buried in the ground. The gasoline then starts to leak from these barrels contaminating the ground around it. We could model the spread of this oil leak using a data layer. The data layer would be a grid superimposed on the game that gives the amount of oil at each grid unit on the map. In addition to this, there needs to be some way to model the spread of the oil between time stamps. At the beginning, there would just be the giant amounts of oil focused at the points where they were buried. But after each time step, the oil will spread to surrounding cells.

Does everything make sense so far?

B.1.2 Tasks

These are the tasks that we asked the test users to perform:

- Remember the game with the leaky barrels of oil that we described to you previously? Please try to create an initial setup of the data layer using our software.
Table B.1: Summary of results of our UI test of the initial prototype.

<table>
<thead>
<tr>
<th>User</th>
<th>Course No.</th>
<th>Task No.</th>
<th>Feedback</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>6</td>
<td>1</td>
<td>how to create a new data layer was not completely intuitive</td>
</tr>
<tr>
<td>A</td>
<td>6</td>
<td>1</td>
<td>the “from function” option was very hard to understand</td>
</tr>
<tr>
<td>A</td>
<td>6</td>
<td>1</td>
<td>the “from points” option didn’t make sense to me at first</td>
</tr>
<tr>
<td>A</td>
<td>6</td>
<td>2</td>
<td>would like to have feedback when clicking the “submit” button for the “edit behavior” tab</td>
</tr>
<tr>
<td>A</td>
<td>6</td>
<td>3</td>
<td>testing was straightforward</td>
</tr>
</tbody>
</table>

- Now try to create the function that specifies how the oil spreads at each step.
- Test to see that this seems to give you what you want.

### B.2 Results

Due to time and resource constraints, we only conducted this study on a few current MIT students. None of them had any particular prior knowledge about our AR games except for perhaps a broad overview. Here’s a table of our observations and feedback.
Bibliography


