TaleBlazer: A Platform for Creating
Multiplayer Location Based Games

by

Michael Paul Medlock-Walton
S.B., C.S. M.I.T., 2010,
S.B., C.M.S. M.I.T., 2010

Submitted to the Department of Electrical Engineering
and Computer Science
in Partial Fulfillment of the Requirements for the Degree of
Master of Engineering in Electrical Engineering and Computer Science
at the Massachusetts Institute of Technology
June 2012

© Massachusetts Institute of Technology 2012. All rights reserved.
TaleBlazer: A Platform for Creating Multiplayer Location Based Games

by

Michael Paul Medlock-Walton

Submitted to the

Department of Electrical Engineering and Computer Science

May 30, 2012

In Partial Fulfillment of the Requirements for the Degree of

Master of Engineering in Electrical Engineering and Computer Science

Abstract

TaleBlazer is a platform that allows users to create location-based games for their mobile device. Using a block-based programming language, game designers can control the behavior of agents that interact with the game players as they walk around in the world with their GPS enabled device. This thesis describes the design decisions and implementation of the repository server which holds stores the games, the mobile client used to play TaleBlazer games, the blocks programming language used to control the agents and players in the game, and the multiplayer server that allows players to interact within the same game world. The experience teaching a workshop with students who used TaleBlazer is described in addition to areas for future development of the project.

Thesis Supervisor: Eric Klopfer

Title: Director, MIT Teacher Education Program


Acknowledgements

I would like to thank Prof. Klopfer for giving me the opportunity to work on TaleBlazer and providing funding during my time at the STEP lab.

This thesis also would not have been possible without the staff members on the project, Judy Perry, Josh Sheldon, and Lisa Stump, and the UROPs who helped code and design TaleBlazer, Vineet Gopal, Sarah Lehmann, Emily Lin, Ruwen Liu, Christian Londoño, Emily Pitts, Yuzhi Zheng, and Hanna Zhu.

I would like to thank my parents for supporting me throughout my time at MIT, and for bringing me to where I am today.

Orit, through good times and bad times, we've gotten through the past few years together. Thanks for making sure it's always entertaining.

Finally, I would like to thank the friends that have been part of my MIT journey, especially those from Lambda Iota Zeta, SuMmErFuN, and C-entry, the support from Dean Staton and the OME, and Anne, Vera, and Linda in the Course VI office.

You all have made these years memorable.
# Contents

1 Introduction ................................................................................................... 7  
1.1 TaleBlazer .............................................................................................. 7  
1.2 Motivation ................................................................................................. 7  
1.3 Thesis Summary .................................................................................. 8  

2 Background ................................................................................................... 9  
2.1 Location-based storytelling and game creation platforms ......................... 9  
2.2 Block-based programming environments ................................................. 10  

3 TaleBlazer Mobile Application .................................................................. 14  
3.1 Game Selection Screens ....................................................................... 14  
3.2 Game Screen Tabs and Views ................................................................. 16  

4 Mobile Application Design ....................................................................... 18  
4.1 Saving Games ....................................................................................... 18  
4.2 Calculating Player’s Proximity to Agents ................................................ 20  

5 Block Design .............................................................................................. 25  
5.1 Features of blocks ................................................................................ 25  
5.2 Goals .................................................................................................... 28  
5.3 Colors .................................................................................................. 31  
5.4 Handling Parameters .......................................................................... 32  
5.5 Confusing Points for Game Designers .................................................... 34  
5.6 Un-implemented Features ..................................................................... 36  

6 Repository Server Design .......................................................................... 40  
6.1 Features of blocks ................................................................................ 40  
6.2 Game Code ........................................................................................... 42  
6.3 Media Files ........................................................................................... 46  
6.4 Upgrading games .................................................................................. 48
7 Multiplayer Server Design................................................................. 50
  7.1 Goals: Same Game World, Intermittent Connectivity ................. 50
  7.2 Approaches ............................................................................... 51
  7.3 Updating the game file ............................................................... 52
  7.4 Technical Goals ......................................................................... 54
  7.5 Security ..................................................................................... 60
  7.6 Un-implemented Features ......................................................... 62

8 Workshop ......................................................................................... 64

9 Future Work .................................................................................... 67
  9.1 HTML5 Emulator ....................................................................... 67
  9.2 Location Independent Games ..................................................... 67
  9.3 Data Visualization .................................................................... 68

10 Conclusion ....................................................................................... 69

References ......................................................................................... 70
List of Figures

2-1 ARIS Requirements Editor ................................................................. 10
3-1 Various screen shots of the TaleBlazer mobile application ................. 14
3-2 A screen shot of the map tab and the agent dashboard from inside of a TaleBlazer game ................................................................. 16
4-1 Scale of game units on a map ............................................................ 22
4-2 Game area changes collision area .................................................... 23
5-1 The scripts for Patient 0 in the Vaccination game .................................. 26
5-2 The scripts for the Sick Person in the Vaccination game ....................... 27
5-3 The scripts for the Healing Ruby in the Vaccination game .................... 28
5-4 Comparison of Scratch and TaleBlazer block colors ............................. 31
5-5 Comparison of the variety of colors in Scratch and TaleBlazer .............. 32
5-6 Comparison of dynamic drop-down lists and block factories .................. 33
5-7 For loop using a block factory ......................................................... 34
6-1 Original game code generating function ........................................... 43
6-2 New game code generating function .................................................. 44
6-3 Table of reordered numbers ............................................................. 45
7-1 How commands are propagated to and from the mobile clients and the repository server ................................................................. 55
7-2 Bumping the treasure chest shows dragon .......................................... 56
7-3 changeAgentVisibility function ....................................................... 56
7-4 handlePropagation function ............................................................. 57
7-5 updateFromUpdateObject function .................................................. 58
7-6 The Node.js processing model ......................................................... 60
7-7 Messages to enable security of TaleBlazer ......................................... 61
8-1 Screenshots from TaleBlazer tutorial ............................................... 65
Chapter 1

Introduction

In March 2012, according to comScore, more than 234 million Americans use mobile devices. [1] Users tend to use their devices for playing games, social networking, maps, and news. [2] People know how to use these different apps, but few can create their own content for mobile devices. TaleBlazer allows users to not only play location-based games on their phone, but also design them using a block-based programming language.

1.1 TaleBlazer

Users can begin using TaleBlazer in the online authoring environment where they are able to create their game by first choosing a map area where the game can be played, then placing agents on the map that the user can interact with. Each of the agents can contain scripts that would be triggered by events that will be fired during the course of the game. Game designers can share their projects online, where other users will be able to provide feedback on the design of the game they created. Players can then download the game to their mobile device, and walk around in the physical space of the game area to play the game by interacting with the virtual agents.

1.2 Motivation

Over the past 1.5 years, with input, feedback, and assistance from the TaleBlazer team, I have designed and programmed the mobile client, block set, repository server, and multiplayer server. Throughout the development of TaleBlazer, design decisions were made to provide the game designer with as
much control as possible while making it intuitive to create single and multiplayer games in the online editor. On the mobile client and the repository server, players should be able to easily find and play games on their iPhone or Android device. In the editor, designers should be able to use the same block set for single and multiplayer games, and have the blocks work as the designer expects in both cases, despite the technical limitations.

This thesis describes the previous work that provided inspiration for the project, the features of the current implementation of TaleBlazer, the design decisions that were made and the constraints that influenced them, the effects of the design decisions from a workshop, and recommendations on how to improve TaleBlazer in the future.

1.3 Thesis Summary

Chapter 2 describes existing location-based game creation environments and block-based programming languages. Chapter 3 will describe how the current TaleBlazer mobile application works. Chapters 4, 5, 6, and 7 will discuss design decisions that were made in the mobile client, the blocks, the repository server, and the multiplayer server, respectively. Chapter 8 will discuss findings from a 7-week workshop using TaleBlazer, Chapter 9 will look at future work, and Chapter 10 will provide concluding thoughts.
Chapter 2

Background

TaleBlazer combines the features of other mobile location-based storytelling and game creation environments with the simplicity of block-based programming environments to provide an easy way to create location-based games.

2.1 Location-based storytelling and game creation platforms

Several platforms like MITAR [3], 7Scenes [4], and ARIS [5][6] allow users to create their own location-based stories and games and play them through a mobile client.

2.1.1 MITAR

MITAR, the predecessor of TaleBlazer, contains two editing environments that are capable of creating location-based games for Windows mobile and Android devices. Similar to TaleBlazer, several icons on the map represent different objects or people, and the player can visit each of them and see the information they provide. MITAR provided two editing environments, Game Editor, which allows designers to have the most amount of functionality available, and Game Builder, which had less functionality, but was intended to be easy for students to use. TaleBlazer endeavors to have more functionality than MITAR's Game Editor while being easier to use than Game Builder.

2.1.2 7Scenes

7Scenes allows users to create location-based stories and games by integrating
videos, pictures, and sounds into tasks that the players can complete. 7Scenes also incorporates multiplayer functionality where players can use their mobile devices or computers to control the action that occurs in the game. After the games end, players can look at a history of all of the events that occurred in the game to reflect on their experience.

2.1.3 ARIS
ARIS allows users to add characters and items to a game and uses requirements put on those objects to control the action in a game. Developed at University of Wisconsin – Madison, the software features the ability to have quests, take pictures, scan QR codes, and an inventory. The requirements editor, as shown in Figure 2-1, provides the logic for the game, which is in many ways replaced in TaleBlazer using the block-based programming language in order to make process of creating the game logic easier.

![ARIS Requirements Editor](image)

**Figure 2-1:** ARIS Requirements Editor: The view used to add logic to ARIS games. The player must have the key to complete the quest.

2.2 Block-based programming environments
Four block-based programming language environments, Scratch [7], StarLogo
TNG [8], App Inventor [9], and Stencyl [10] informed the development of TaleBlazer.

2.2.1 Scratch

Scratch is a block-based programming language that allows kids to create games, stories, and animations and post them online. The blocks control how each of the individual sprites move and animate on the stage. Created by the Lifelong Kindergarten Group of the MIT Media Lab, it is the most popular of the block-based languages with more than 2.5 million projects online.

2.2.2 StarLogo TNG

StarLogo TNG allows game creators to use the blocks to control breeds of agents to create 3d games and simulations. StarLogo TNG is different from the other blocks languages in that multiple agents run the code generated by the same blocks simultaneously, allowing users to see the effects of a multi-agent simulations.

2.2.3 App Inventor

App Inventor allows game designers to use blocks to create Android applications. Users interact with the primitive Android controls, along with some built in components to control the screen on their Android device. Unlike TaleBlazer where it is possible to script the agents to perform some action, in App Inventor, designers program individual elements on the screen, such as labels, buttons, and images along with some higher-level components to interact with the users.

2.2.4 Stencyl

Stencyl allows anyone to create Flash and iOS games using a block-based language. Stencyl allows users to create behaviors that are can be attached to different actors in order to control the behavior objects in the game. By
decoupling the behaviors from the actors, the blocks that control the actions of agents are reusable, and also refer to "self" in order to get and set the properties of the particular actors. This approach is similar to the "me" that is included in many drop-downs TaleBlazer in order for the blocks to be easily copy and pasted across multiple agents.

2.2.5 Differences between the languages and TaleBlazer

Three differences between the block-based languages are how the blocks are used to control other agents, the runtime environment for the blocks, and the size of the block set.

2.2.5.1 Controlling other agents

TaleBlazer and Stencyl allow the designer to control the behavior of other agents by selecting the agent in a drop-down list on a block. Scratch does not allow users to directly control a different agent other than the currently selected one. Sprites in Scratch can subscribe to messages being sent by other sprites in order to react to commands sent from other sprites. In StarLogo TNG and App Inventor, when new breeds or components are created, a "My Blocks" palette appears with custom blocks that can be used to access the properties of the new object.

2.2.5.2 Runtime environments

TaleBlazer is the only language to run on a server, and the only block-based language to split the evaluation of the blocks over two machines. In multiplayer games, the mobile clients must communicate changes with the server, and the server runs certain stacks of blocks in order to maintain consistency in the game state. While StarLogo TNG only runs in a desktop application and App Inventor only runs on Android devices, Stencyl runs in Flash and iOS, Scratch runs in a desktop application and online in a Java applet, and TaleBlazer runs on Android and iOS devices.
2.2.5.3 Size of block set

The size of the block set for TaleBlazer and Scratch are similar with each containing 6 and 8 drawers respectively. The number of drawers and blocks in StarLogo TNG and App Inventor block increase with the My Blocks section, which adds a new set of blocks for every component. Stencyl has the largest block set by including categories of blocks inside each of the drawers. Minimizing the size of the block set while maximizing the functionality provided by the blocks was a goal throughout the development of TaleBlazer.
Chapter 3

TaleBlazer Mobile Application

The mobile client allows users to play TaleBlazer games on iOS and Android devices. The mobile client was built with Appcelerator’s Titanium, a cross-platform mobile framework that allows programmers to code in JavaScript to program for both iOS and Android. The mobile client has two parts, the game selection screens where players can find games, and the game screens where users play the games.

3.1 Game Selection Views

Figure 3-1: Various screen shots of the TaleBlazer mobile application. From the left, the home screen, the game code page, a game page, and the browse page.

When the user starts the TaleBlazer mobile application, they will see a screen like the first picture in Figure 3-1. If a player knows what game they want to play,
they can simply tap the "Enter Game Code" button, and type in the game code that uniquely identifies the game on the repository server to access the game's page on the mobile device, as shown in the second picture in Figure 3-1. For players who do not know what game they want to play, they can click on the "Find a Game" and they are presented with a list of games that are near their location. If a user has made a game that they want to play, he can tap the login button on the home page to sign in. After logging in, the user is sent back to the homepage, where he can hit the "Find a Game" button, and the browse page will have an additional tab that is populated with games that the user have made. The game page for single player games gives information about the game and a button to tap in order to download the game.

3.1.1 Missing feature - Browsing and finding games

The players’ experience browsing for games can be improved by providing more information about the games on the game list, adding the ability to search by username, and allowing players to easily find location independent games. Currently on the game list, the player can only see the name of the game, the user who created it, and the description. Adding the distance the user is from the starting the location and adding icons to indicate single or multiplayer games could be helpful to users that are finding games to play. While a player can login and see their own games, it is not possible to search for games made by another player. This can be fixed by adding another tab that contains a textbox and submit button at the top of the screen, and the search results in the table of games below.

Currently the new users to the TaleBlazer mobile application will see games near their location when they first tap the "Find a Game" button on the home screen. Most users will not have games near their location to play, making the application useless until they create their own games. However, if a list of games that were not location dependent were available, anyone who downloads the TaleBlazer
app will be able to play a TaleBlazer game without creating one.

3.2 Game Screen Tabs and Views

After the game is downloaded, the player is asked to select their role. Roles allow different players to have different traits and actions. When the user submits their choice for their role, he is entered into the game. The main game screen consists of several tabs at the top that the game designer can customize and order. The map tab, agent dashboard, and inventory tab will be described to provide the reader background regarding how the TaleBlazer mobile application works.

Figure 3-2: A screen shot of the map tab and the agent dashboard from inside of a TaleBlazer game.

3.2.1 Map tab

The map tab presents a map of the region that the player is in and the icons of the agents in their locations. There are three kinds of maps that the user can see, dynamic, static, and indoor. A dynamic map uses Google maps to show the location of the users and the agents. A static map uses an image provided by the
game designer for the map, but works the same way as a dynamic map, where the player can see their location and the location of the agents. An indoor map uses an image provided by the game designer for the map, like the static map, however the player cannot see their location on the map because the GPS is not used on indoor maps.

3.2.2 Agent Dashboard

When a player walks close to an agent in a region where GPS is used, or when a user taps on an agent icon in an indoor region, in many games, an agent dashboard will appear. The agent dashboard consists of the name, picture, description, action, and traits of the agent, as shown in Figure 3-2. The actions are buttons that will show the player text, video, or run a script when pressed. The game designer can programmatically control which actions are showing at any point in the game. In addition there are some pre-defined actions that are available to the game designer including, "pick up", "drop", and "give".

3.2.3 Inventory

When a player chooses the pick up action on the agent dashboard, the agent is added to the player's inventory. The inventory is a list of the agents that the user has picked up. The agents may have a "drop" action that would remove them from the inventory.
Chapter 4

Mobile Application Design

Both the game selection views and the game screen tabs have gone through design iterations, especially regarding the process of saving games and detecting the user's proximity to agents, and both parts still have room for improvement.

4.1 Saving Games

The goal in saving TaleBlazer games to the mobile device is to allow players to easily save, reset, and update the games that are made in the online editor. The process for saving games has changed over time from not saving the games to the file system, to saving multiple games, and allowing games to be reset. The ability to update games needs to be implemented.

4.1.1 Games not saved

When first creating the TaleBlazer mobile application, the games were stored in memory but were not saved to the file system. A call was made to the server to download the game file, and the JSON string was parsed and turned into a JavaScript object. Anytime a change was made inside of the game, the object was changed. The problem with this model was that the game was lost if the user ever exited the application.

4.1.2 Saving multiple games

In the next iteration of the game selection screens, the mobile client was able save multiple game files. Inside of the mobile device's file system, a file called
storedGameData.txt holds the file manifest. The file manifest is a JSON string that holds the meta-data about the games saved on the device, in addition to the next value of an increasing counter to give every game stored on the device a unique identifier. The game meta-data is used to populate the game page on the mobile device. Having the meta-data stored on the mobile device means that the player does not need to be connected to the Internet in order to see all of the information about a game they have on their phone. Every game is stored in a folder called "game#" where "#" is the unique identifier for the game defined in the file manifest. Inside of the folder, there is a file called "gameFile.txt" that holds the game file that was downloaded from the server. In addition to the game file, the media files that are extracted from the zip file on the server are in the "game#" folder.

4.1.3 Reset Games
After implementing the ability to save games on the mobile device, there was a clear need to have the ability to reset games. In the previous implementation, to restart a previously played game from the beginning, the player would need to delete the game from the mobile device, and download a new copy of the game. This process could take a long time considering that all of the image and video files would need to be downloaded again. To fix this, when the game is downloaded, a copy of the game file is stored in a file called "gameFileReset.txt". Whenever the player decides they want to reset the game, the contents of "gameFile.txt", which changes as the game is played, is replaced with the content of "gameFileReset.txt", which remains unchanged from when the file was originally saved on the phone.

4.1.4 Updating games
While the current game selection pages are functional, there is more work to do regarding updating games. If a player has a game on their mobile device, and the
designer of the game has made a change to the game online, then the player should have the option to update to the latest version of the game. This would mean that the player's progress in the game would be lost, but they would be able to try out the latest version that is available. Right now, in order for this to happen, the user must delete the game from the mobile device and download the game again. Ideally, the player should just tap an "update" button on the game page to update a game. After confirming that the player wants to update the game and lose their current progress, the newest version of the game should be downloaded to their mobile device.

The simplest way to update a game would be to delete the old game folder, and re-download the new game folder. While this would work, it may not be efficient. If the game designer simply made a spelling change in the game file, it would be a waste of time to re-download all of the image and video files associated with the game. Given that the repository server knows which media files are associated with the old and the new version of the game, the server could send only the new media files, and instruct the mobile client to delete the media files that are no longer being used.

4.2 Calculating Player’s Proximity to Agents

The distance used to detect when a GPS triggered "bump" should occur between a player and an agent has changed by switching from using game coordinates to units that are more closely related to physical distance. Another change to this distance should be made to convert the current units to a standard unit, such as meters.

4.2.1 Detecting a bump using game units

The first implementation of TaleBlazer used game coordinates for the distance required for a player to bump an agent. In the editor, when a game designer
wants to build a game, he can decide where the game should be located. Designers can specify the area where the game will be played by dragging the corners of a rectangle that is overlaid on a Google Map. The game, regardless of the size of the rectangle, has a coordinate system that is 200 by 200. When the agents are placed on the map in the editor, the agents receive an x-y coordinate.

On the mobile device, when the player moves, their longitude and latitude is retrieved from the GPS and is converted into x-y coordinates. If the player's location is within 20 game units of an agent, the agent's GPS auto-bump will trigger. There were two problems with this approach, the default value did not scale well, and it was easier to bump agents from certain directions.

4.2.1.1 Scaling the default value
Setting a default value in game units for the distance that would allow the player to bump into an agent did not work well across games of different sizes. The coordinate system is always 200x200, regardless of the size of the game area. So in a small field, like Killian Court at MIT, a GPS bump radius of 20 game units would allow the agent to pop up when the player was approximately in the right location. However, in a game that spans all of MIT campus, 20 game units could have the player triggering an agent that is a building away as shown in Figure 4-1.
Figure 4-1: Scale of game units on a map. On the picture on the left, the distance from the building to the center of Killian Court is 100 game units, while the same distance is 40 game units in the picture on the left. The change occurs due to the size of the game area.

4.2.1.2 Bumping agents from different directions

In games where the game area is not square, it is easier to access an agent from particular directions. In a game where the game area is a square, the area that the player must enter to trigger the agent is a circle. However, if the lengths of the sides of the game area are not equal, the area the player must enter to trigger an agent becomes an oval. This is due to the fact that the game area is 200 x 200, regardless of whether the game area is not a square. Figure 4-2 shows that it is easier to trigger the agent if the player is walking north or south instead of east or west.
Figure 4-2: Game area changes collision area. On the picture on the left, the game area is a square, allowing the player within the yellow circle around the agent to trigger the agent. However, when the game area is rectangular, the player has to walk within the oval to trigger the agent.

4.2.2 Fixing Scaling Default

In an attempt to solve the scaling default problem, units that are proportional to real world distances are used. The goal was to have a unit where being 1 unit away from an agent in one game is roughly the same as being 1 unit away in another game, regardless of the size of the map. Since the collisions are still occurring in game units, it is not possible to use a real unit like "meters", because the radius around an agent is not circular for non-square game areas. Instead, the mobile client assumes that the square root of the area is the length of 200 game units, and that is used to calculate the collisions. This is how the mobile client currently calculates the distance needed for a player to bump an agent.

4.2.3 Change map coordinates

The way to fix both the scaling default problem and the problem with bumping
agents differently from some directions is to make the coordinates on the map scale to the size of the map. Right now the map is always 200x200, regardless of the fact that the map may not be a square. By allowing the coordinate system to change proportionally to the game area, standard units, like meters and feet, can be used by the designer to determine the size of the radius around the agent that would cause it to automatically bump the player.
Chapter 5

Block Design

The blocks in TaleBlazer allow the game designer to have programmatic control over the agents and traits in the game. By stacking them like LEGO bricks, designers can build up sequences of commands that are executed when certain events occur in the game, such as a player bumping into an agent or a player giving an agent to another player. Throughout the process of creating the block set, design decisions were made regarding the features that the blocks would control, the goals behind designing the blocks, the block colors, how parameters are handled, minimizing confusion for game designers, and features that need to be added to the blocks engine.

5.1 Features of blocks

The blocks allow the game designer to hide, show and change the location of agents, change the traits of agents, players, and teams, hide and show actions and traits, and perform math operations. Conditional blocks allow the game designer to test for different conditions in the game, including checking a player's inventory, the role or team of a player, the visibility or location of an agent, or the value of a trait, to create interesting game dynamics.

To demonstrate the features of the blocks, we will walk through the Vaccination Game created in TaleBlazer. In the game there is a vaccine, a needle, patient 0, a sick person and a healing ruby. The vaccine and the needle are items that the player needs to pick up, and use on patient 0 to cure him. The first stack of blocks in Figure 5-1 shows this interaction: when the agent bumps Patient 0, if
the inventory of the agent contains the vaccine and the needle, then the agent's vaccinate action is shown.

![Game Map Agents Player Teams Vaccination Game 2 Save Game](image)

**Figure 5-1:** The scripts for Patient 0 in the Vaccination game.

The "when the agent bumps Patient 0" block is a hat block, essentially an event handler. All stacks of blocks need a hat block to run, as the hat indicates when the stack should be run. The block that follows is an "if" block, which tests the boolean condition to the right of the "if", and if the result is true, runs the blocks in the socket next to "then". The shapes of the blocks help indicate where they will fit, and their data type. The boolean blocks like the "and" block and the "contains" block have circular notches that match the socket in the if block. Similarly the command block "show action" has notches above and below to indicate that it can connect with other command blocks.
When the vaccinate action is chosen, the vaccinated trait of Patient 0 is set to 1, and Patient 0 says, "Thanks for vaccinating me" to the player, as shown in the second stack of blocks in Figure 5-1. Traits are variables that are associated with the game, an agent, players, or teams. Traits can be agents, players, or values, which include numbers and strings. Patient 0 has a trait called "vaccinate" which is set to 0 by default in the editor. We see from the third stack in Figure 5-1 that the health of patient 0 goes decreases by 5 every minute that he is not vaccinated. When the patient is vaccinated, their health will not continue to decrease.

<table>
<thead>
<tr>
<th>Game</th>
<th>Map</th>
<th>Agents</th>
<th>Player</th>
<th>Teams</th>
</tr>
</thead>
</table>

![Image of vaccination game interface](image)

Figure 5-2: The scripts for the Sick Person in the Vaccination game.

In addition to helping vaccinate patient 0, the player must avoid the sick person who will chase them throughout the game. As shown in Figure 5-2, every 1 minute, the sick person will point toward the player and move 5 steps. If the player ever bumps into the sick person, the health of the player is decreased by...
and the sick person is moved to a random location on the map. The number socket on the "move <agent> to x: <number> y:<number>" can take an input of a typed number, or a block that is a number, like the "pick random" number block.

Lastly, in a multiplayer game, the user can give the healing ruby to another user. Multiplayer games contain a shared world where all players can see the same agents and are able to change the state of the game world. The game designer can use the blocks to control the interactions that happen when players interact with the agents, just like the single player game. When a player gives the ruby, the giver receives two health, and the receiver gets one health. By using the giver and receiver in the drop down on the trait blocks, game designers can control interactions between two players.

5.2 Goals

When creating the block set for the first version of TaleBlazer, the goals were to use the same block set for both single and multiplayer games, and to be intuitive for new users while maximizing functionality.

5.2.1 Same block set for Single and Multiplayer games

Creating a block set that works for both single and multiplayer games requires
adding blocks dependent on the player, but still make sense in a single player context, providing timing blocks that account for the limits of server communication, and making blocks that avoid infinite loops on the multiplayer server.

5.2.1.1 Player dependent blocks
Making blocks work in both a single and multiplayer game was mostly accomplished through adding a player input into blocks, which allows them to be player dependent in a multiplayer game. For a single player game, the player would refer to the only player in the game, while the player would refer to the particular player in that context in a multiplayer game. In multiplayer games, the game designers also have the opportunity to give agents to each other, and the player option in block drop-down lists gives the game designer the ability to create interesting dynamics with the other players. For example, giving the healing ruby in Figure 5-3 gives the giver 2 health points and the receiver 1 point.

5.2.1.2 Timing Blocks
Timing blocks that control events at the granularity of minutes instead of seconds allow the game designer to create time based events while not overloading the server and keeping a consistent block set across single and multiplayer games. In the current block set, there are two timing blocks, "every 1 minute" and "after 1 minute". Both blocks allow the game designer to change the "1" in the block into another integer. While allowing the user to control timing events to the granularity of seconds could provide different mechanics that are not currently available, having the server send updates to mobile clients every second would not provide a consistent game experience for all users. Since players may be in and out of Wi-Fi range, and given that there may be network delays, it is not possible to assume that all of the mobile clients would be able to receive updates every second.
5.2.1.3 Infinite Loops
The blocks were designed to make it difficult to create infinite loops on the multiplayer server. While other block-based environments run solely on a client device that the user controls, the blocks that are created in TaleBlazer can be run on the multiplayer server. Creating infinite loops on the server would prevent other games from running properly. To prevent infinite loops, in the first version of TaleBlazer's block set, functions cannot call other functions unless user intervention is required, in the case of a "jump" block for dialogs, where a user must make a selection for the game to continue. Of course it is possible to use these functions in ways that are not problematic, however it is not possible to use static analysis to determine if there will be an infinite loop. Considering that we want errors to be seen by the game designer as opposed to the player, and because a dynamic analysis would be required to find errors, it is ideal to build the block set in a way that avoids functions calling other functions. In the future, if infinite loops were to be added, then a recursion limit and possibly a time limit would need to be put on the execution of the blocks.

5.2.2 Intuitive while maximizing functionality
By providing the features that power users may want while balancing the size of the block set with the visibility of the blocks, the design of the block set intends to be intuitive for new users while maximizing the functionality.

The trait blocks demonstrate a design decision that was made when balancing the number and visibility of the blocks. There are three types of traits: values, agents, and players. In addition, it is possible for all of the traits to be variables or lists. Traits that are variables have blocks to set and get the values, and list traits have blocks to add an item to a list, delete an item from a list, check if a list contains an item, and get the length of the list. If each of the blocks were available for each data type, there would be 18 blocks in the traits drawer. Instead of creating trait blocks for each of the different types, the number of
blocks can be reduced to 6 by having the socket dynamically change to match the desired type. By reducing the number of blocks, it is not as obvious that different traits will have different sockets that are dependent on their type, however it provides the benefit of having a smaller block set.

5.3 Colors

The colors of the blocks were created to be similar to Scratch in order to have some uniformity between the block-based languages. While the colors in TaleBlazer are lighter due to the black text on the blocks, the blocks in the control, movement, operators, and looks drawers use roughly the same color blocks as Scratch's equivalent block categories.

![Figure 5-4: Comparison of Scratch and TaleBlazer block colors. The colors of the control, operators, and movement blocks are similar between Scratch and TaleBlazer.](image)
In addition, all of the block colors are distinct enough to have a one word common color name associated with them, specifically orange, green, yellow, purple, blue, and pink. Many other block-based programming languages rely on light and dark versions of the same color to distinguish block categories; Scratch has light and dark blue, and light and dark green. The goal in making the colors distinct is to have them be easier to refer by color name without being ambiguous. In TaleBlazer, if a designer is using a blue block, it is a movement block, while a blue block in Scratch could be from the motion or the sensing category.

<table>
<thead>
<tr>
<th>Scratch blocks</th>
<th>TaleBlazer blocks</th>
</tr>
</thead>
<tbody>
<tr>
<td>pen down</td>
<td>when game starts</td>
</tr>
<tr>
<td>set pen color to pick random 1 to 10</td>
<td>If</td>
</tr>
<tr>
<td>change pen shade by 5</td>
<td>player is a Wizard</td>
</tr>
<tr>
<td>move 10 steps</td>
<td>then</td>
</tr>
<tr>
<td>ask What's your name? and wait</td>
<td>show action Pick up</td>
</tr>
<tr>
<td>turn 60 degrees</td>
<td>set x of me (Wand) to 5</td>
</tr>
</tbody>
</table>
| set power of me (Wand) to | Figure 5-5: Comparison of the variety of colors in Scratch and TaleBlazer. In Scratch, the operators and the pen blocks are both green, and the movement and the sensing blocks are both blue. In TaleBlazer, each of the block drawers has a different color name associated with it to make discussing the blocks simpler.

5.4 Handling Parameters

Throughout the development of blocks, several designs were considered for handling the parameters of hat blocks. Different hat blocks have parameters that the designer should have access to inside of the stack of blocks. For example, the block "when player bumps <agent>“, there are two parameters to the hat block, the player and the agent. When the game designer is creating a script under the hat block, he should have access to the player's traits and the agent's traits and be able to modify them. The two ways of giving the game designer access to parameters that were considered are block factories, and dynamically
5.4.1 Block Factories

Block factories allow the designer to drag parameter blocks from inside of a hat block to access the values of the parameters inside of a stack. This approach was first used by the Scratch modification BYOB, renamed Snap, and is used in Scratch 2.0. The benefit to using block factories is that the idea of scope, having access to particular variables only under certain circumstances, becomes visible.

5.4.2 Dynamic Drop-down Lists

Dynamic drop-down lists allow the game designer to select a variable that is currently in scope from a list that is embedded in the blocks in the stack. This is the approach that is currently used in TaleBlazer. For example, in blocks in the stack under the "when player bumps <agent>" block, the designer can select "player" from the drop-down list. Additionally, blocks under the "when player gives <agent>" block, the "giver" and the "receiver" are selectable in the drop-down list as shown in Figure 5-6.

![Dynamic Drop-Down Lists](image1)

![Block Factories](image2)

**Figure 5-6:** Comparison of dynamic drop-down lists and block factories. The drop-down list on the left is how TaleBlazer currently allows users to select the parameters. The picture on the right shows how block factories could be used to give users access to parameters using blocks. The blocks on the right were generated using BYOB [11].

The problem with this approach is that it is not visually clear to the user which hat
blocks cause additional options to be populated to the drop down selections. The idea of scope is also not as salient because there is no visual feedback that certain drop-down selections would only work in certain stacks.

Figure 5-7: For loop using a block factory.

The for-each loop highlights a difference between the block factory and the drop-down list approach. Using the block factory, the loop could be implemented as shown in Figure 5-7. When the user wants to access the item, they could drag the block into the script in the appropriate location. Using the drop-down method, the user either must drag a block inside the for-each loop and discover that the for-each loop provides an additional option, or there should be a "parameter" block that the user can drop in, similar blocks in the "My Blocks" drawers in StarLogo TNG.

The main reason that the block factories were not tested in TaleBlazer is because the ScriptBlocks engine, which generates the blocks, did not support having block factories inside of hat blocks. In the future, if the ScriptBlocks engine is able to handle the block factories, using a combination of the factories and the drop-down lists could allow both the simplicity of the automatically populated drop-downs, while providing the clarity of the embedded block factories.

5.5 Confusing Points for Game Designers

The implementation of several features may be confusing to game designers who are beginning to use TaleBlazer, including how the action and trait visibility is
reset, showing the pick up and drop actions do not always work, and running scripts from asynchronous calls in multiplayer games.

5.5.1 Resetting Action and Trait Visibility

Resetting the visibility of the actions and traits works differently than the rest of the TaleBlazer platform, which may make it a point of confusion for users. Most of the changeable properties in the game including the value of the traits of the players, teams, agents and the location and the visibility of agents are changes that affect all players of the game. Also, there are some properties that are specific to the player, but remain permanent for the entire game, such as whether the player has visited an agent or not. However, the visibility of actions and traits are different from both models. Whenever a player bumps into an agent, the agent's actions and traits that are visible by default are the actions the designer set to be visible in the action and trait panel. The show and hide action/trait blocks under the "when player bumps <agent>" hat block allow the designer to dynamically change the visibility of the actions and traits. If the player decides to run a script action, then the hide and show action/trait blocks will change the current state of the visibility of the actions and traits. Finally, if the player leaves the agent, and returns, the scripts that hide and show the actions and the traits start from the initial visibility of the actions and traits, not the state the player left the actions and traits in. Changing the visibility of the actions and traits only affects one player, not all of the players in the game.

5.5.2 Showing Pick up and drop actions

Even though the pick up and drop actions remain in the list of actions to show, the action may not be valid depending on whether the agent is located in the inventory at the time. If an agent is located inside of the inventory, and the show action block is called with the action "pick up", the action will not appear, as it is not possible to pick up an object that is already in the player's inventory.
5.5.3 Running scripts from an asynchronous call

In multiplayer games, some scripts must run on the server due to synchronization that must happen between the mobile devices, however, that places some restrictions on how the blocks can be constructed. When picking up an agent in a multiplayer game, the server must be notified in order to avoid multiple clients believing the agent is in the inventory of multiple players. However, the mobile client has no guarantee when the server will respond to its request to pick up the agent. If the server responds to the pick up request a minute after it received the request, then executing certain blocks like "show video" and "say" might interrupt the gameplay. The current solution to the problem is to prevent blocks that cause an interface to appear, like the "say" block, from working in the "when agent is picked up" block in multiplayer games. This restriction does not apply in single player games, which could cause confusion among game designers.

5.6 Un-implemented Features

A combination of a lack of time and the lack of features in ScriptBlocks, the engine used to generate the blocks, has led to features that were designed to not be implemented in the current version of TaleBlazer. Features that would improve the current version of TaleBlazer include adding buttons to the blocks, adding stack dependent features in the blocks, and adding the ability to change images in the game using the blocks.

5.6.1 Buttons on blocks

If a future version of the blocks engine used by TaleBlazer allowed buttons on the blocks, functionality can be added to allow if/else-if/else blocks, blocks that ask multiple-choice questions, and provide formatted text throughout the game.
5.6.1.1 if/else-if/else blocks
Adding buttons to the blocks would allow else-if statements to be part of the language as opposed to nesting blocks. The current way to create an else-if statement is to nest an "if" block inside of an if-else block. Several else-if statements would cause the blocks to look unnecessarily nested. By adding a button between the "then" and "else" labels that a user can press, the user could create an "else-if" socket, allowing game designers to check for multiple cases while having cleaner code.

5.6.1.2 Multiple Choice Questions
If buttons were added to the blocks engine, the designer would be able to more easily ask multiple-choice questions to the players and enable branching dialogs. In the current implementation of TaleBlazer, the easiest way to create a multiple-choice question is to create multiple actions on an agent. This is not ideal due to the small amount of room available for the actions, and the large amount of work needed to get the questions set up.

Adding buttons to a multiple-choice block would allow the designer to easily add answers to the block. In addition, adding a button under each answer could open a socket connection that would allow users to programmatically control what happens after a specific answer is chosen.

5.6.1.3 Say blocks with rich text editor
A button in the say block would allow a rich text editor to pop-up and allow the designer to display formatted text within the execution of the blocks.

5.6.2 Stack Dependence
The current version of TaleBlazer does not dynamically change the blocks based on the stack the block is located in. Two places where this functionality could help designers are by providing a visual indicator where blocks will not work, and
to update the drop-down list based on the hat block.

5.6.2.1 Indicate Invalid Block Placement
Visually indicating what blocks will not work under certain stacks will minimize errors and help the designer to understand where certain blocks cannot be used. The show and hide action blocks are used to change the visibility of the action buttons on the agent, team, player, and game dashboard. However, these blocks do nothing when a dashboard is not present. A hat block is always run before a dashboard is shown, including "when player bumps <agent>" and "when player sees role tab", but some hat blocks do not refer to a dashboard being shown, like "when game starts". In order to indicate which stacks the hide and show action blocks will work with, as the user is dragging one of the blocks, the stacks where the block cannot be placed can be grayed out. This solution will make it more obvious to the game designer when certain blocks will not work based on the stack they are located in.

5.6.2.2 Stack dependent drop-downs
Allowing the designer to only choose drop-down arguments that are applicable in the stack would ensure that all of the drop-down selections were valid. Some blocks, like the trait blocks, allow the designer access to the properties of a player in the game. The player in the drop down, however, is only valid when there is a player to refer to. Under a "when game starts" block on a multiplayer game, there is no "player" to refer to. Additionally, blocks in a stack with the "when player gives <agent>" block should have access to the "giver" and the "receiver", but these options should not appear in the drop-downs of any other stack. By changing the options in the drop-downs based on the stacks, the designer will only have access to options that make sense within the context that the block is located in.

5.6.3 Changing Images Programmatically
Switching the image and map icon of an agent allows the game designer to change the appearance of an agent without needing to create an unnecessarily large number of agents. In the current version of TaleBlazer, in order to simulate changing the picture of an agent, the user would need to create two agents, and hide the one with the old picture while showing the other with the new picture. This cumbersome method does not work well when agents are inside of a player’s inventory, where hiding and showing an agent will not give the illusion that the agent’s picture has changed. To overcome this, an agent should have multiple pictures associated with it that the designer can choose from using the blocks. This is similar to the costumes in Scratch. A similar set of blocks can be used to change the map icon for the agent.
Chapter 6

Repository Server Design

The repository server stores the games that the users make using the TaleBlazer editor. Throughout the development of TaleBlazer I have created and improved on the different ways to publish games, access games using codes, saving the media files, and upgrading games.

6.1 Publishing Games

After a game designer finishes creating a TaleBlazer game, he will want to publish the game so that it can be playable on their mobile device. Over time, three different approaches have been taken to storing the published games on the server. Along with many design decisions that were made when creating TaleBlazer, when the initial approaches were created, they were made to be temporary fixes to immediate problems that would be revisited later when there was more time to devote to fixing the larger problem.

6.1.1 One Database Row per Game

When the server was first being set up, it was important to be able to simply save a game, and retrieve the saved game from the server. A table in the database called "games" was created to hold the game files. Inside the editor, the first time that the user clicks the save button, an entry would be added to the database, and the game would receive a unique gameld, the primary key for the table. Whenever the user would go to http://serverName/create/gameld the server would provide the editing environment for that saved game. When editing the saved game, if the designer hit the save button again, the game kept the same
gameld, and the new game file replaced the old game file that had previously been saved.

While this approach solved the immediate need for being able to save and restore games, it lacked the ability to distinguish games that the user wanted to save from games that the user would want to be playable on a mobile device. For example, if there was a logical error in the blocks, the designer would not want that version of the game to be downloadable on the phone, but he might want to save the game's current state so that he could work on the game later. Another problem with this implementation was that by replacing the game files and not saving the old versions, if the editor were to accidentally save an incorrect file, the work of the user would be lost since the old version of the game would not be retrievable.

6.1.2 Separating Saved and Published games

As the API calls to retrieve game data for the mobile client were being developed, having one row in the database for each game was not sufficient. In the second iteration of saving games to the database, a "save" button and a "publish" button were added to the editor. The user's profile page was split into two categories, one for published games and another for saved drafts. The published games would receive game codes and would be playable on mobile devices. The saved drafts were links to the editor so designers could continue changing the game they were making.

The original idea was that "saving" and "publishing" should be equivalent to saving and printing a document inside of a word processor. The designer would edit the game in the saved draft and then essentially "print" a game by publishing it. Like a printed page, the changes that the designer made in the saved draft after publishing had no effect on the saved draft.
This approach solved the problem of separating games that users wanted to save from the games they wanted to be released and available on mobile devices. In addition, every time the user clicks on the save button, the server saves the new game state in a new row in the database, without overriding the old row. However, this approach became untenable for two reasons: it was strange to have the published game and the saved draft in two different locations on the profile page, and the number of published games became very large and impossible to distinguish.

6.1.3 Showing latest published game

Before TaleBlazer needed to be demonstrated to clients of the lab, the way that the saved and published games were displayed on the designer's profile page was changed to address the problems with the previous version. The two sections "saved drafts" and "published games" were removed in favor of having only one section of games. The view of the games looked like the "saved drafts" section, with the exception of a link that allow the user to see the most recent published game. This design fixed two problems, 1) the user could now easily access the published version and the draft version of a game from the same location on the page, and 2) the number of published games was not large considering only the most recently published game was available.

One problem I have noticed with users of the TaleBlazer editor is a misunderstanding of what publishing and saving do. For example, if a user publishes a game, makes changes to the game, and then saves it, it is not obvious to some that the published game is not affected.

6.2 Game Code

The game codes provide a way for users to get access to a game that they want to play on their mobile device. After a game designer publishes a game on the
server, the published game receives a game code. The player can type in the
code on their mobile device to access the game so they can download and play
it. TaleBlazer has had two implementations of the game code which will be
described, and I would recommend one in the future that includes the usage of
QR Codes.

6.2.1 Predictable Game Codes

In order to quickly get the game code functionality working on the mobile devices
and the server, I temporarily created the function in Figure 6-1 to calculate a
game code. The function takes in a gameld, and converts it into a string. Every
digit is turned into a number, so a = 0, b = 1, c=2...j=9. Also, the letter "g" is put
before all game codes so I know it is a game code. So a game with gameld 21
would have a game code of gcb.

```php
function createGameCode($id){
    $gameCode = '';
    for($i=1;$i<strlen($id);$i++){
        $gameCode = $gameCode.chr(((str($id)/pow(10, strlen($id)-$i))%10)+97);
    }
    return 'g'.$gameCode;
}
```

Figure 6-1: Original game code generating function. This function creates game
codes that are predictable, but inefficient since only the letters a-j are used as
opposed to the entire alphabet.

There were two problems with these codes; first they did not use the entire
alphabet, which would eventually lead to codes with large numbers of characters.
The second, and more pressing problem is that the game codes were too
predictable. One of the requirements of the project is that users could create
games that were not available to the general public. Also, those non-publicly
available games need to be available without the user logging in. This method to
generate game codes would not be sufficient considering that a player could
easily browse both public and private TaleBlazer games by using successive
game codes "ga", "gb", "gc", "gd", etc.

6.2.2 Fixed Length, Unpredictable Game Codes

When the server was changed to the most recent method of publishing games, the game codes also changed to make the new codes a fixed length and unpredictable. The code in Figure 6-2 creates the new game codes.

```php
function getNewGameCode($id)
{
    $gcNum = bcmul($id, 260225119), 308915021);  
    $gcDivide = $gcNum;

    $gameCode = "";
    for($i=0; $i<6; $i++){
        $gameCode = $gameCode.chr(($gcDivide%26)+97);
        $gcDivide = $gcDivide / 26;
    }

    return 'g'.$gameCode;
}
```

**Figure 6-2:** New game code generating function. This function creates game codes that are unpredictable, and with a fixed length.

The goal for this game code function is to take the gameld and convert it into a game code with letters from "a" though "z". With the old game code function, the numbers that were converted into letters were in base 10, so we could only use 10 letters to represent the numbers. Now the code will use base 26 to use all of the possible letters. Numbers were not used in addition to the letters in the game codes because it can be difficult to switch between letters and numbers on mobile devices. The new function will now have a fixed length of 6, excluding the leading "g". This means that there are $26^6 = 308,915,776$ possible game codes.
Now that we have a way of creating fixed length game codes, we have to make the game codes unpredictable. Currently, the game code to gameld pairing would be, "gaaaaaa": 0, "gaaaaab" : 1, "gaaaaaz" : 25, "gaaaba": 26, etc, which is very predictable. To fix this problem, all of the numbers between 1 and 26^6 - 1 are arranged in an arbitrary order.

The code relies on relatively prime numbers and the modulus to arrange the numbers in an arbitrary order. To demonstrate, let's say I wanted to arrange the numbers between 1-6 in some arbitrary order. Given I have 6 elements, I must pick a number relatively prime to 7, we'll use the number 5. Relatively prime numbers do not share any factors other than 1. Lastly, we multiply the numbers 1-6 by 5, and use mod 7 to get the results you see in Figure 6-3. You will notice that all of the numbers from 1 to 6 are distinct, and in a different order.

<table>
<thead>
<tr>
<th>i</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
</tr>
</thead>
<tbody>
<tr>
<td>5 * i</td>
<td>5</td>
<td>3</td>
<td>1</td>
<td>6</td>
<td>4</td>
<td>2</td>
</tr>
</tbody>
</table>

**Figure 6-3**: Table of reordered numbers. This demonstrates how to reorder numbers between 1 and 6 mathematically. The numbers between 1 and 308915021 were reordered using this method to generate the game codes.

The function that calculates the game code uses this strategy for all numbers between 1 and 308915021. The larger number, 308915021, is one of the primes that is closest to 26^6 so that we generate as many game codes as possible. From Figure 6-2, 260225119 is a prime number, which by definition will be relatively prime to any other prime number.

It is possible that a user could figure out the formula used to calculate the game codes by converting the letters into numbers and doing the math, or by reading this thesis. A "game password" could be added for private games. This would require the user to type a password after typing in the correct game code. The
user would need to know the "game password" to download the game. Since this password would only be accessible and editable by the creator, it should be safe from anyone who happened to guess the game code of a private game.

### 6.2.3 QR Codes

While the current game code generator is sufficient for the current needs of the project, a QR code for each game would improve the usability of the TaleBlazer when accessing a game. It can be extremely cumbersome to type the game code using the onscreen keyboard on a mobile phone. Having the buttons so close together makes typing prone to errors, and since the codes are not words, the autocomplete feature that normally helps users improve accuracy on these small screens is useless. If QR codes were implemented in TaleBlazer, it should be in addition to the current text game codes, so that users who do not have the QR code can still easily access games. The QR codes would allow a player to simply click one button, hold up their phone to the game page on the website, which would contain the QR code, scan the code, and click a button to download the game.

### 6.3 Media Files

TaleBlazer games often have pictures and videos that are added to the game by the designer, and must be downloaded to the mobile clients. There are different approaches that can be used to accomplish this including, accessing the media files from the server as needed, putting the media files in a zip file, distributing the media files between several zip files, and offloading the videos to a video hosting website.

#### 6.3.1 Download media files as needed

An approach to handling the media files is for the mobile device to access them from the server as they are needed. So, if a player bumps into an agent in the
game, the mobile device would request from the server the picture of the agent. While this is the most straightforward approach, it is not sufficient for TaleBlazer because it requires a persistent Internet connection to access images.

6.3.2 Download zip file with all media files
The first approach that was implemented in TaleBlazer was to put all of the media files in one zip file. At first, this worked for the majority of games, however when games with many of videos were added, the zip files became large, with sizes over 30 megabytes. The mobile clients had trouble downloading files that large, so another approach became necessary.

6.3.3 Split zip file, download different parts
The current approach that is used on TaleBlazer is to put the media files in several zip files, and the mobile client downloads each of them. Downloading several smaller zip files allows the mobile clients to download games that have many media files. To obtain the files the mobile client first asks the server for a list of the zip files that need to be downloaded. After the server returns the list, the mobile clients requests, saves, and extracts each of the zip files.

6.3.4 Using a Video Hosting Website
In order to minimize the size of TaleBlazer downloads and also minimize the files stored on the lab's servers, videos could be hosted on a video hosting website like YouTube. While it is important that the clients of the lab are able to have games with downloadable videos, when TaleBlazer is released to the general public, hosting videos on the server may be too much for the servers to handle. Currently, when a video is uploaded to the server, it is converted into an mp4 with aac encoding for the audio so that it can be watched on both the iOS and Android devices. In addition to encoding the video, it is also compressed so that it takes up the smallest amount of space possible on the server while retaining a clear
picture on many mobile devices. The encoding is time and resource intensive on
the server, and probably better handled by YouTube. In addition it is undesirable
to hold many large video files on the servers at MIT. The disadvantage to using
YouTube is that the user must be connected to the Internet to watch the video.

6.4 Upgrading games
As new features are added to TaleBlazer, the game files and the mobile client
must be upgraded. Two approaches to handling the upgrading of games is to 1)
upgrade automatically to the latest version when the game is saved and 2)
upgrade all of the games on the server at once.

6.4.1 Upgrade when saving game
An approach that could be taken is to upgrade a game to the latest version
whenever it is saved. Using only this approach would lead to the fragmentation of
the TaleBlazer platform, as different games would only be available for different,
possibly unsupported versions of the software.

6.4.2 Upgrading all games at once
The approach that is mostly implemented and will be used in TaleBlazer is to
upgrade all of the games at once. This makes it possible for users to play any
game created with TaleBlazer using the newest version of the software. In
addition to updating the JavaScript file that generates the game file when a user
saves, changes to the games in the database, the mobile code, and the server
code must be made for the transition to newer versions of TaleBlazer to be
mostly seamless to the players and game designers.

6.4.3 Updating the database
All of the games in the database must be upgraded to the newest version. A
function in JavaScript was created to convert an old game to a new game.
NodeJS uses an HTTP GET to retrieve the games from the repository server, uses the function to upgrade the games, and then uses an HTTP POST to add the upgraded game to the repository server. The games are added as if they are the next saved or published version of the game, so the old version of the game is not lost. The old version is still saved so older clients can access the old version.

### 6.4.4 Updating server code

Even after all of the games in the database have been updated there is still a possibility that a designer will be attempting to create a game using an old version of the JavaScript file, which would lead to a old version of the game file being added to the database. To fix this problem, whenever a game file is about to be saved to the database, the server must check the version of the game file. If the version is not the most up-to-date, then the JavaScript function must be called to update the file before it is inserted in the database.

### 6.4.5 Updating the mobile code

Supporting the new game file also requires changes to the mobile code. First, when a new version of the mobile client is installed, it must update all of the game files that are stored on the phone. Also, the API calls used to get playable games from the server need to include the mobile version number so that the server will only return games that had playable game files for a compatible version of the mobile client.
Chapter 7

Multiplayer Server Design

The multiplayer server uses Node.js to allow mobile clients to communicate with each other so that multiple players can play the same game together. The design of the multiplayer server is the backbone behind the unique functionality of providing an easy way to create games where players can interact with each other. The implementation novel as it allows designers to use blocks to create scripts that run remotely on the multiplayer server to affect changes on the mobile clients playing the game.

Understanding the development of the multiplayer server requires knowing the goals for multiplayer gameplay, the different approaches that could have been used, design decisions regarding where to update the game file first, the technical goals, and the security of the server.

7.1 Goals: Same Game World, Intermittent Connectivity

The two primary goals in creating the multiplayer server are to allow players to play in the same game world together, and that it is possible to play when there is intermittent connectivity with the server that synchronizes the communication between the mobile clients. Players in the same game world should be able to see changes in the game world caused by other players, for example, if a player picks up an object, the object should disappear from the ground for all of the other players in the game.

Working with intermittent connectivity requires the players to continue playing the
game even if there is not a Wi-Fi connection to the server for a short period of
time. For example, if a user needs to go into a tunnel and come back out to get to
an agent in the game, the game should be able to handle receiving updates after
the user returns from being inside the tunnel.

In addition, the game world must remain consistent to some degree, even without
constant communication to the server. For example, if two players go in a tunnel
at the same time, and both pick up a key located inside of the tunnel, when they
both leave the tunnel, they should not both have a key in their hand because the
single key should not be cloned. Managing user expectations while providing a
predictable world for the game designer was a guiding force in the design of the
multiplayer server.

7.2 Approaches

There are three approaches that were considered when building the multiplayer
server: a pull approach where the mobile clients would poll a central server, a
peer-to-peer approach where clients would communicate directly with each other,
and a push approach where a central server would send updates to the mobile
clients.

7.2.1 Pull

The pull approach would allow mobile clients to send updates to a central server
and poll the server for changes. A benefit of this approach is that it could use the
already existing PHP backend to support the API calls. The main problem with
this approach is that it is too slow when propagating the updates from the mobile
clients.

7.2.2 Peer-to-Peer

The peer-to-peer approach would allow the mobile clients to communicate
directly to each other. When joining a game, the mobile clients would register with a central server that would maintain a list of IP addresses and ports of the mobile devices that are playing the game. When a player made a change to the game state, the mobile clients would be responsible for sending the appropriate update messages to the other mobile clients. The benefit to this approach is that a central server is not burdened with transferring all of the game state changes between all of the mobile devices playing TaleBlazer games.

A technical complication with this approach is network address translation (NAT). Mobile devices behind NATs, which change the IP address information in the messages sent from the device, often cannot accept incoming connections. It is possible to use an intermediary server that is not behind a NAT to assist in communicating with devices behind a NAT.

7.2.3 Push
The push approach, which is currently implemented in TaleBlazer, allows the mobile clients to send updates to a central server, and the server is responsible for saving and forwarding those updates to the mobile clients. This publish-subscribe pattern was implemented using TCP sockets in Node.js. This approach allows information to be sent directly to the clients, eliminating the delays from the pull approach, and does not require incoming connections to be established with the mobile clients, removing the problem that the peer to peer approach has with NATs.

7.3 Updating the game file
Designing the multiplayer server, requires deciding where the game file is first updated after a change occurs, on the server, on the mobile devices, or on a combination of both.
7.3.1 Updating on the server first

One approach to updating the game files is for the mobile clients to send any change that occurs to the server, have the server arbitrate between the different request and save the game file, and send the changes in the game state to the mobile clients. Having the server handle the request to the mobile devices ensures that all of the mobile devices have the same game state and do not get out of sync. The problem with this approach is that it fails to meet the intermittent connectivity requirement. Playing a multiplayer game would always require having a connection to the server to take any action.

7.3.2 Updating on the mobile client first

Another approach would be to update the mobile clients first, and pass the changes to the server. This approach would allow the mobile clients to take actions anywhere, fixing the intermittent connectivity problem of the previous approach. The problem with this approach is that it can lead to inconsistent data. For example, the key that is cloned by two users picking it up in the tunnel would be possible.

7.3.3 Combine updating on the server and the mobile client

The approach that is currently used on TaleBlazer is to update the mobile client first in most situations, and to update the server first when synchronization is important. Allowing the mobile devices to update the game files first in most situations brings the benefit of being able to play the games despite intermittent connectivity. In the current version of TaleBlazer, anytime an agent is picked up, the server must update the game file first. This makes it possible to avoid the problem of having two users picking up a key in a tunnel, and both leaving the tunnel with a key, even though there was only supposed to be one key in the game. Due to the ordering commands, it is still possible to get inconsistent state with this approach. An approach to fix this is to periodically send the entire game...
7.4 Technical Goals

Given that updates to the game file will happen on the server and the mobile client, the maximizing code sharing between the mobile client and the server and managing concurrent requests to the server were technical goals for the development of the multiplayer server.

7.4.1 Maximizing Code Sharing

The code to update the game file is shared by the mobile device and the multiplayer server. The benefit of using Titanium for mobile development and Node.js for the multiplayer server is that it is possible to use the same JavaScript code for both platforms.

There are four files that are shared between the mobile device and the multiplayer server, arlogo_model.js, blockEvaluator.js, updateState.js, and mobileUpdate.js. arlogo_model.js has all of the helper methods to access the game file. This file is also used in the game editor on the website.

blockEvaluator.js is responsible for evaluating the stacks of blocks that the game designer created whenever the action that triggers those blocks is invoked. Functions inside of updateState.js are run whenever the game model needs to be updated. mobileUpdate.js contains functions that update the user interface of the mobile in response to changes in the game state. The functions in mobileUpdate.js do not do anything on the multiplayer server.
Figure 7-1: How commands are propagated to and from the mobile clients and the repository server. Messages start from the mobile client at the top, sent to the multiplayer server in the middle, and end at the other mobile clients playing the same game instance.

In order to demonstrate how all of the files connect on the server and the mobile, the steps required to trigger an agent to be visible on all mobile devices playing a game will be described.
In this demonstration game, there will be treasure chest on the map. When the player runs into the chest, a dragon will appear, as shown in the blocks in Figure 7-2. On the mobile client, when the player runs into the chest, blockEvaluator.js is run to evaluate the blocks. A function in blockEvaluator.js calls the changeAgentVisibility function in updateState.js, shown in Figure 7-3. The parameters of changeAgentVisibility include the agentId, the unique identifier of the dragon agent, the visibility, which is true since the dragon is becoming visible, and a propagate variable, which is always true when the function is being called from blockEvaluator.js. The propagate variable is used as an indicator of whether the function should send a message to the server or the other mobile clients about the update to the game file.

```javascript
tb.blocks.updateState.changeAgentVisibility = function(agentId, visibility, propagate){
    tb.blocks.context.game.getAgentFromID(agentId).visible = visibility;
    tb.blocks.mobileUpdate.updateAgentLocation(agentId);
    tb.blocks.updateState.saveGame();
    var paramObject = {
        "agentId":agentId, "visibility":visibility
    };
    tb.blocks.updateState.handlePropagation("changeAgentVisibility", paramObject, propagate);
}
```

**Figure 7-3:** changeAgentVisibility function. This function is in updateState.js and is called on both the mobile clients and the repository server to hide and show agents.
The function changeAgentVisibility then sets visibility of the agent in the game model to true in line 3. The function updateAgentLocation in mobileUpdate.js is called to make the agent appear on the map on the mobile device, and the game file is saved in the file system of the mobile device using the saveGame function.

The next line of changeAgentVisibility creates an object that has the parameters of the function. This will be used in order to communicate what needs to be changed to the game server. Lastly, the handlePropagation function is called with the name of the function, "changeAgentVisibility", as the first parameter, the parameters for the changeAgentVisibility function as the second parameter, and the propagation variable as the third.

```javascript
1 if(!tb.blocks.context.game.multiplayer){return;}
2  
3 if(propagote && tb.platform == "mobile"){
4     paramObject.propagate = true;
5     tb.game.server.updateServerGameState(methodName,paramObject);
6 } else if(propagote && tb.platform == "server"){
7     paramObject.propagate = false;
8     var excludeIds = [];
9     if(tb.blocks.updateState.serverOriginPlayerId != null){
10        excludeIds = [tb.blocks.updateState.serverOriginPlayerId];
11     } 
12     tb.blocks.updateState.serverUpdateObjects.push(
13        tb.blocks.updateState.createUpdateServerObject(methodName,paramObject,excludeIds))
14 }
15}
```

**Figure 7-4:** handlePropagation function. This function is run on both the mobile client and the multiplayer server in order to determine whether a message should be sent to the from the server to the mobile device, or vice versa.

The handlePropagation function is responsible for passing the message to the server or mobile device if necessary. The function first checks that the game is a multiplayer game. If the game is a single player game, the function will stop evaluating after the first line. The next line checks to see if the propagation variable is true, and if this code is being run on the mobile client. Considering that both are true, it sets the propagation variable to be true, and sends the name of the function and the parameters to updateServerGameState so the message can
be sent to the server. The propagation variable is set to true because when the message is evaluated on the server, the message should be propagated to all of the mobile clients.

On the multiplayer server, after the message has been correctly routed and the correct game file is opened, the server calls the function updateFromUpdateObject that is located in updateState.js. updateFromUpdateObject takes the update object, and using the method name and parameters calls the correct function in updateState.js in order to update the game model. In this case, "changeAgentVisibility" is selected, causing the changeAgentVisibility function to be called on the server with the same parameters that were used when it was called on the mobile device.

```javascript
1. tb.blocks.updateState.updateFromUpdateObject = function(updateObject){
2.   tb.blocks.updateState.serverOriginPlayerId = updateObject.playerId;
3.   var params = updateObject.parameters;
4.   switch(updateObject.method){
5.     ...
6.     case "changeAgentVisibility":
7.       tb.blocks.updateState.changeAgentVisibility(
8.         params.agentId, params.visibility, params.propagate);
9.       break;
10.   ...
11. }
12. }
```

Figure 7-5: updateFromUpdateObject function. The mobile client and the multiplayer server run this function in order to send an incoming message to the correct function in updateState.js.

When changeAgentVisibility is called, the dragon's visibility is updated in the model on the server, and the handlePropagation function is called. This time, the boolean test in the "else if" is true because the propagate variable was true, and the platform is the server. Now, the propagate variable is set to false because the mobile clients that are going to receive the message that the dragon is not visible
should not propagate the message because all of the mobile clients and the server have already received the message. Then, the handlePropagation function creates an array called excludeIds that has the playerId of the device where the message originated. This is done so that the message does not return to that device again. Lastly, an updateServerObject is created using the method name, the parameters, and the excludeIds, and is pushed onto an array of these updateServerObjects, which the server processes to send updates to the mobile clients.

After receiving the update from the server, the function updateFromUpdateObject in updateState.js is called on the other mobile clients. Again, updateFromUpdateObject causes the function "changeAgentVisibility" to be called. This causes the model to be updated on the mobile clients, and the call to "updateAgentLocation" in mobileUpdate.js causes the dragon to appear on the map of the mobile devices. Since the propagate variable is false, the message is no longer passed on.

By having four shared JavaScript files, all access to the model and when to propagate messages is decided in code that is shared between the mobile client and the server, allowing for code reuse.

7.4.2 Managing concurrent requests
The multiplayer server is set up to handle requests that come in while other requests are being processed. Since event loop in Node.js is a single thread, there are no threads that have to be synchronized for the server to manage the connections properly. However, the server does need to efficiently handle multiple requests coming in at the same time.
Figure 7-6: The Node.js processing model. The diagram shows that a single threaded event loop receives the client connections, and that Node.js send the long running task to non-blocking workers. This architecture allows the game queues, which are only updated in the event loop, to not be concerned about concurrency issues since the loop is not multi-threaded. The diagram is from Aaron Stannard’s blog. [12]

When a message comes from the mobile client, a queue for the game instance is created. Accessing the game file from the file system takes time, and while the server is accessing the file it is possible for updates to come in from the mobile clients. This queue holds all of the updates from the mobile clients that come in while the multiplayer server is saving or opening a game file. After the game file is opened, all of the messages in the queue are processed, and the queue is emptied.

7.5 Security

Communications with the multiplayer server can be secured using RSA public key encryption for all messages going between the server and the mobile client.
When the mobile client originally logs in, it should generate a public and a private key. The public key should be sent to the repository server along with the user's credentials, as shown in Figure 7-7. When the mobile device initiates a connection with the multiplayer server, it should encrypt messages using the server's public key. This will allow only the server to read the messages. Messages should contain a sessionId for the player, so the multiplayer server can request the public key of the user and encrypt messages to the mobile device.

**Figure 7-7:** Messages to enable security of TaleBlazer. This diagram shows what messages are sent between the mobile client, the repository server, and the multiplayer server in order to have secure communications in the game.

The userIds that are sent to and from the multiplayer server should be replaced with sessionIds to increase the security of the application. The userIds are the unique identifier associated with the user in the database. If a malicious player is
has guessed the password of a player, even if the real player changed their password, the server could not de-authenticate the malicious user since they are communicating with the server using the unique identifier for the user. By using a sessionId, the server can revoke permissions from any logged in user by invalidating a sessionId.

The security measures were built in to TaleBlazer, but were subsequently removed due to efficiency. The time to generate the public and private keys on the mobile device took more than 15 seconds at times, which was not practical for general use. A more efficient method should be found for generating the keys.

7.6 Un-implemented Features

7.6.1 Teams
Teams allow multiple players to have shared traits and inventory, enabling different game mechanics in multiplayer games. Grouping players together in the current implementation of TaleBlazer is cumbersome, but possible. By allowing teams to be created, game designers can easily allow players to work together to achieve a goal together.

7.6.2 Drones
Drones allow multiple clones of the same agent to be used in a game, which allows game designers to create playable games without knowing the number of players in the game. In multiplayer games, the game designer will not always know how many players will be in the game. However, in the current implementation of TaleBlazer, the game designer can only include a pre-defined number of agents inside of the game. For example, if a player is required to have a key to enter a door, but five players joined the game, and the designer only
created three keys, then two players would not be able to continue the game. A drone is a template agent that could be cloned. Allowing dynamic cloning of agents would allow the game designer to create a key for every player in the game, regardless of the number of players that joined.
Chapter 8

Workshop

During the development of TaleBlazer, I held workshops for a group of 12 middle school students for 1.5 hours on Saturdays for 7 weeks. The challenge with the workshops is that TaleBlazer had a very limited feature set and a large number of bugs when the workshops began. As the workshop progressed, the students found bugs and requested features in the software that were added over time.

After a small introduction to the class and the editor, the first class began with the students playing the MIT hack game, which allowed them to understand the capabilities of the platform. After the game, the students walked through the game they had played to see how to make their own. In the following classes, I demonstrated a new concept or feature every class, and encouraged students to explore it while they were building their own games.

However, in the second class, when the students were beginning to create their own TaleBlazer applications, many were confused about how to build a game. Also, as the students were asking questions, I was forced to answer the same question multiple times, even after explaining it to the entire class.

In the week between two classes I created a mini-tutorial that leads students step by step to create a scavenger hunt, the simplest game to make in TaleBlazer, and builds to using more advanced functionality in the platform. This allowed the students to learn the interface of TaleBlazer at their own pace, and allowed me, as the instructor, to spend more time answering questions that only specific students had, as opposed to questions the entire class had.
Parts of this tutorial, which are available on the TaleBlazer website, can provide a scaffolding for any user who is beginning with TaleBlazer. It begins with how to get started in the editor, continues with instructions for finding the game on the mobile device, and ends with optional ways to make the hunt more interesting as users explore the features of TaleBlazer.

**Scavenger Hunt Tutorial**

(Descriptive text about the tutorial's contents)

**Step 1: Creating a Scavenger Hunt**

1. Create a new game
2. Name the game
3. Map tab
4. Find your address
5. Move the game map
6. Lock the map

**Step 2: Playing the Scavenger Hunt**

1. Go to game page
2. Tap the Game Code button
3. Enter Game Code
4. Download the game
5. Tap the Map Tab
6. Walk to an agent

![Diagram of the steps](image)

**Figure 8-1:** Screenshots from TaleBlazer tutorial. The tutorial teaches users how to create a Scavenger hunt and expand on it to learn different features within TaleBlazer.
Reactions from the class, besides finding bugs in the software, mostly included the desire for TaleBlazer to do more. The students recognized the limits that the block-based approach put on the ability to build apps, but also appreciated the simplicity as it allowed them to make games quickly after they knew what they were doing. Common feature request included allowing player’s to have traits, like score, and for players to have an inventory. While the inventory was implemented before the last class, the player traits were not.
Chapter 9

Future Work

TaleBlazer could improve by providing more feedback quickly to game designers about problems with their game, exploring location independent game designs that TaleBlazer supports, and providing the tools to allow players and programmers collect data about TaleBlazer games.

9.1 HTML5 Emulator

While TaleBlazer lacks the immediate feedback that other block-based programming languages provide, an emulator can shorten the time between when the designer creates and tests a game. In Scratch, StarLogo TNG, and App Inventor, designers can quickly test the changes they have made to a program. In the current version of TaleBlazer, the user must download the game and its assets again to test it. Titanium 2.0 supports the mobile web in addition to Android and iOS. [12] By embedding the Titanium mobile client in the editor, game designers will be able to test out their game as they are making it in the editor, decreasing the time it will take for game designers to find mistakes in their programs. In addition, this will allow users who do not have a smart phone to play TaleBlazer games from their computer.

9.2 Location Independent Games

Even though a lot of focus throughout the development of TaleBlazer has been on creating location-based games, there is research to be done into location independent game mechanics that work well with the platform. The mechanics behind many interactive fiction games, including the ability to have an inventory,
players and objects in the world having traits, and the ability to go room to room, are supported in a using TaleBlazer. Allowing multiple players to interact in the interactive fiction's game world is similar to Multi-User Dungeons (MUDs) without the text-based input. Games can also be created based on passing objects from person to person, a similar dynamic to many card games.

9.3 Data Visualization

Further research should be done into ways of visualizing data generated from games, and ways allowing programmers, and non-programmers to interact with the data. As players walk around with their mobile devices playing a multiplayer TaleBlazer game, the server processes every update, and applies them to the game model. Research into what information from the game is useful, and how to expose that information could lead into interesting applications that expand on TaleBlazer. There can be visualizations that work for any game, including a playback of all of the events that occurred, similar to 7Scenes. Game specific visualizations could be made, including a graph displaying the results to a question that players answered during the game, or a customized team scoreboard based on different traits in the game. An API could be developed to allow the TaleBlazer community to generate their own game specific visualization from the game instance data. In addition, programs could be created to allow the data from the game instances to be imported into programs like Scratch [13] so novice programmers can have the opportunity to create their own data visualizations.
Chapter 10

Conclusion

TaleBlazer allows users to create single and multiplayer location-based games on iOS and Android devices using a block-based programming language. Throughout the development of TaleBlazer, design decisions focused on making it easy to create games for beginners while not limiting the potential for advanced users. Multiplayer and single player games use the same block set for creating games. In multiplayer games, the blocks that the designers stacks together are executed both on the multiplayer server and the mobile client in order to provide a more consistent game world in spite of intermittent connectivity. TaleBlazer will help players to not only explore the environment around them through playing games, but also turn users into creators of stories and games for their own mobile devices.
References


