TaleBlazer Multiplayer: Expanding Multiplayer Functionality for Meaningful Location-Based AR Games

by

Tanya X. Liu

S.B., Massachusetts Institute of Technology (2013)

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

Master of Engineering in Electrical Engineering and Computer Science

at the

MASSACHUSETTS INSTITUTE OF TECHNOLOGY

June 2014

Signature redacted

Author ....
Department of Electrical Engineering and Computer Science
May 23, 2014

Signature redacted

Certified by....... ....
Professor Eric Klopfer
Director, MIT Scheller Teacher Education Program
Thesis Supervisor

Signature redacted

Accepted by............. ....
Prof. Albert R. Meyer
Chairman, Masters of Engineering Thesis Committee

TaleBlazer Multiplayer: Expanding Multiplayer Functionality for Meaningful Location-Based AR Games

by

Tanya X. Liu

Submitted to the Department of Electrical Engineering and Computer Science
on May 23, 2014, in partial fulfillment of the
requirements for the degree of
Master of Engineering in Electrical Engineering and Computer Science

Abstract

TaleBlazer is a location-based, augmented reality game platform. Its web-based editor provides game designers with a method to create games regardless of programming experience. Users play these games while walking in the real world, using the mobile application as a gateway for interactions with the virtual world. This thesis extends previous work done on the TaleBlazer multiplayer game platform. It details the implementation of new multiplayer functionalities that allow meaningful interactions between players in multiplayer games.

Thesis Supervisor: Professor Eric Klopfer
Title: Director, MIT Scheller Teacher Education Program

Acknowledgments

First of all, I'd like to thank Eric Klopfer, Lisa Stump, and Judy Perry for allowing me to join the TaleBlazer developer team and to work on this project. I have learned so much over the past year, and it would not have happened without them.

I'd like to thank Judy Perry for being a guiding force throughout the entire project and helping me keep a timeline of work. I'd also like to thank her for all her help when it came to design decisions. I'd also like to thank Lisa Stump for being so helpful and patient when it came to all my questions about implementation and code structure. Without the both of them, this project would never have gotten off the ground.

I'd also like to thank the TaleBlazer Development team, including my fellow M.Engs Fidel Sosa, Cristina Lozano, and Stephanie Chang. It is because of them that the project was so enjoyable to work on, and I am forever grateful for their patience and helpfulness in answering my barrage of questions.

I'd like to thank my predecessor, Sarah Lehmann, for getting the multiplayer server chugging along so that I could have the opportunity to implement and expand the multiplayer features. I would especially like to thank Paul Medlock-Walton for all his wise words and many suggestions that greatly furthered development work.

I'd also like to thank Albert Meyer, my academic advisor. Without his advice and help during my undergrad (and graduate) years I would have been rather lost.

I'd like to thank my friends, whose support and cheer have kept me alive, kicking, and positive during stressful times.

Lastly, I'd like to thank my family, whom without their eternal support I never would have gotten the chance to be where I am today.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

TaleBlazer is a location-based, augmented reality game platform developed in the MIT Scheller Teacher Education Program (STEP) lab. Using TaleBlazer, game designers can develop their own educational games regardless of previous programming experience. These games are played on mobile devices, where players use the device GPS while walking outdoors to interact with the virtual game world. TaleBlazer is a mature single player platform that has been published for both Android and iOS phones. Organizations currently partnered with the project use TaleBlazer games as an unique learning experience for visitors. For example, the Old Sturbridge Village, a historical representation of New England, has created a game to teach visitors about historical economics. The games are a fun and engaging way for players to experience and learn something new.

Though TaleBlazer is successful in providing a single player an interactive virtual world, rapidly evolving mobile technology leads users to increasingly expect a level of connectedness with other users that TaleBlazer currently does not have. It thus follows that the TaleBlazer project would be greatly enhanced by the inclusion of multiplayer functionality.

1.1 Why TaleBlazer Multiplayer?

Adding multiplayer capability to TaleBlazer adds a social dimension to an educational experience. In a single player game, interactions are limited to the player and his or her own game world. The player gathers information on his own, and, for the most part, relies only on himself to accomplish a goal. Adding more players to the game world and allowing them to interact and cooperate introduces the concept of teamwork. Instead of individually working toward a goal, players now must learn both reliance on and accountability towards others. The implementation of multiplayer within TaleBlazer would allow for games that revolve around such qualities.

1.2 Thesis Summary

- Chapter 2: provides background on the history and implementation of TaleBlazer
- Chapter 3: discusses the goals and applications of multiplayer functionality, and also details the current state of implementation
- Chapter 4: explains the design decisions made as well as the implementation of the new multiplayer features
- Chapter 5: describes the various tests conducted
- Chapter 6: discusses features that can be implemented to extend multiplayer games
- Chapter 7: discusses contributions and conclusions

Chapter 2

TaleBlazer Background

As a location-based, augmented reality game platform, TaleBlazer enables the creation of education-based games that help users learn. The TaleBlazer project has many different components, and has several predecessors that have helped guide its development.

2.1 Similar Software

There have been previous platforms and software developed in the STEP lab that were similar in functionality to TaleBlazer. The experiences and ideas from these past ventures helped guide further development of TaleBlazer.

2.1.1 MITAR

MITAR was a predecessor to TaleBlazer. Users could create augmented reality games for handheld devices, and gameplay functioned similarly to TaleBlazer. Players would be able to interact with different icons on the map in order to gain information regarding a particular in-game scenario. An example MITAR game was "Environmental Detectives", where players ventured outdoors to uncover the source of a toxic spill. To do so, players interacted with virtual characters and simulated data collection and analysis [5]. Unlike TaleBlazer, MITAR did not use a block-based scripting language.

MITAR was more difficult to use, as the editor was an installed windows application, and the deployment of games to devices was cumbersome.

2.1.2 StarLogo TNG

StarLogo TNG is an extension of the previous StarLogo simulation and modeling software. It utilizes 3D graphics and sound to allow users to easily build games and simulations to help understand complex systems. Like TaleBlazer, StarLogo TNG uses a block-based programming interface to simplify the programming of such systems [8].

2.1.3 Previous AR Simulations/Games

The STEP lab previously hosted multiple relatively large scale augmented reality simulations and games. Two of main games that the STEP lab ran are:

- Mystery at the Museum
- Outbreak @ MIT

These two games would significantly help guide the development of TaleBlazer multiplayer.

Mystery at the Museum

Mystery at the Museum was the first large-scale indoor augmented reality simulation ran by the STEP lab [6]. It was hosted at the Museum of Science, and pairs of players were given pocket PCs to play with. Upon entering a room, players were shown what virtual characters and items were located in the room, and had the option of interacting with them. Different players had different roles, and each role would have access to different sets of information and actions. While the players were not in a completely shared world, the game had multiplayer characteristics, as players could interact with each other and send information via infrared data exchange. This was a very early iteration of an augmented reality multiplayer game, and helped generate ideas for multiplayer in TaleBlazer.

Outbreak @ MIT

Outbreak @ MIT was the first client-server based augmented reality game deployed by the STEP lab [7]. It utilized Wi-Fi connectivity to a server in order to maintain a shared world among all players. This meant that the actions that players took could directly influence the game experience of other players in the game. For example, resources in the O@MIT game world were shared between everyone and thus limited: once a player picked up an item, no other player could access it. Players had to work together in order to investigate and contain a potential outbreak on MIT campus. Like previous AR games, O@MIT was played on pocket PCs. The multiplayer aspects in this game heavily influenced multiplayer functionalities that were implemented in TaleBlazer.

2.2 TaleBlazer Game Structure

While TaleBlazer games are played in the context of the real world, players interface with the game by receiving information from their mobile phones. The mobile device uses its real-time GPS location to track the player in the game world. If a player's GPS location is close to an object's location, he or she will be able to interact with it. Most information is presented through object dashboards, which display the name, description, and other details about an object. There are many components that define a TaleBlazer game. These are defined by the game designer during creation, and fall into the following categories:

- Agents: Agents are the characters and items that the player encounters in game. These are the main objects that players will interact with, and are accessible on the map.

- Regions: Regions are the different maps that are available to the player. They are locked to specific GPS coordinates.

- Roles: A role is a player's defining characteristic. Players can have role-specific traits and actions and thus can have different game experiences. The player chooses his or her role at the beginning of every game.
- Traits: Traits are variables that are specified for objects in the game. Visibility and values of traits can be changed during the game.
- Actions: Actions appear as buttons on an object dashboard and allow players to interact with the game. Action scripts are defined in the editor, but the visibility of actions can be changed in game.

2.3 Current TaleBlazer Design

The TaleBlazer software is comprised of three main components: the web-based editor interface, used to create games, the servers, which are used both to store and host games, and the mobile application, used to play games. The next few sections provide more detail on these three parts.

2.3.1 Editor

The game editor is a web-based interface used to create TaleBlazer games. The editor provides a wide range of options that enrich the game experience. The user is able to select a location at which the game will be played and set game mechanics that will govern the gameplay. Mechanics used in the game can include the ability to visit different regions or the ability to interact with agents that are scattered around the game world. Game designers are also able to write different scripts that execute depending on a player's actions. Designers utilize a block-based scripting language to program game mechanics [9].

Block-Based Programming

The game designer uses a blocks-based programming language to specify the game logic. The blocks are predefined and are easily understandable by a designer regardless

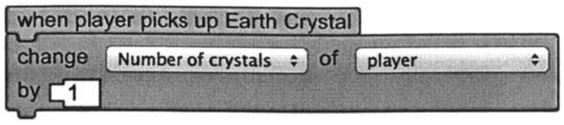


Figure 2-1: An example of a block script.

of programming experience. Each block has its own name and takes arguments. Figure 2-1 is an example of a script that a game designer could create for his or her game. This script will change the "number of crystals" trait of the player when the player picks up the crystal.

2.3.2 Servers

TaleBlazer uses two servers to store and power its games. One of the servers is a repository server. This server is what stores all the files necessary to play a game. When a game designer saves a game in the editor, the required pictures, videos, and resulting game file are stored in the repository server. When a player selects a particular game to play, the mobile will query this server for the relevant game files.

The second server is the multiplayer server. This server facilitates all communications between different devices, and ensures that the game worlds between players within the same game remain synced. The multiplayer server will be discussed in detail in a later section.

2.3.3 Mobile

The third and last part of TaleBlazer is the mobile application that players use to play TaleBlazer games. The mobile application is available on both Android and iOS systems, and is built using Titanium Appcelerator. Using Titanium allows developers to create the application for more than one platform. Upon installation of the mobile application, the player uses the mobile application to select and play a game.

The software on the mobile side of TaleBlazer runs and displays the games that are created in the editor. It downloads the predefined game file from the server and translates the blocks used by the designer into functions that will properly display

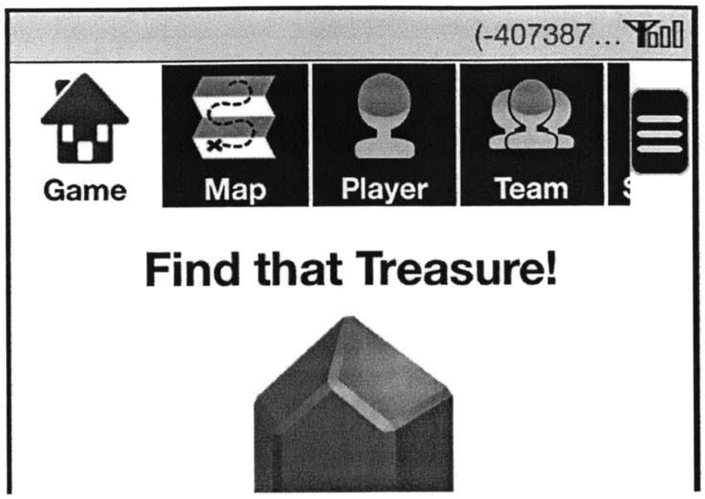


Figure 2-2: An example of tabs that can be shown on the mobile UI.

the game and its contents to the player.

The mobile interface contains a number of preset tabs, with which the player interacts with the game. The interface of each game can look slightly different, as the game designer selects the visible tabs during game creation. Two of the default tabs are the "Game" tab and the "Map" tab. The "Game" tab contains information about the game and allows the player to leave the game if necessary. The "Map" tab is the player's portal into the game world; this tab depicts the location of the player and various agents in the game world. There are a variety of tabs that can be displayed as part of a game, as can be seen in figure 2-2.

Chapter 3

TaleBlazer Multiplayer

The idea behind multiplayer games is that players play and interact in a shared world. In a single player TaleBlazer game, players are able to each experience their own versions of the game world without digitally affecting another player's game world. A multiplayer game, however, encourages players to communicate with one another, making the TaleBlazer experience much more social and interactive. Multiplayer functionality to enhance TaleBlazer games has long been in production.

3.1 Multiplayer Goals

TaleBlazer multiplayer should allow fluid player-to-player interaction. Players must be able to have a seamless game experience, where the actions of other players logically affect their own gameplay. This means that each player's game world should be constantly synchronized, despite intermittent connectivity.

In addition to being in the same game world, interactions between players should be meaningful and add an exciting element to the gameplay. Game designers should be able to allow players to interact with each other in order to further their game experiences. Ideally, a TaleBlazer multiplayer game should encourage players to be social, using teamwork to work with or against other players to achieve game goals.

Though the multiplayer portion of TaleBlazer should add a significant number of new functions, it should not deviate too much from the already familiar single player interface. Those who are familiar with TaleBlazer in the single player universe should easily be able to create or play a multiplayer game.

3.2 Possible TaleBlazer Multiplayer Game Types

There are an enormous variety of multiplayer game types, and each introduces many functionalities. It was necessary to narrow TaleBlazer's focus and decide what types of multiplayer interactions would best benefit TaleBlazer games. The development of TaleBlazer multiplayer focuses on three main multiplayer game types:

- Player vs. World
- Single Cooperative Team Play
- Multiple Competitive Team Play

Each type of multiplayer game gives an opportunity for game designers to develop different interactions between players.

3.2.1 Player vs. World

In a player vs. world type game, players interact with the game world simply to achieve their own, separate goals. The agents in the game world are shared and consistent across the worlds of all players, and player actions are capable of affecting the overall game world. An example of such a game would be players racing to pick up as many treasures as possible, where the player with the most treasures at the end is the winner. Previously implemented TaleBlazer multiplayer mechanics enabled the creation of simple versions of this type of game.

3.2.2 Single Cooperative Team

This type of game is completely cooperative. Players must help each other fulfill a common goal, each one bringing a part of the solution. In such a game, it is possible that players would each have a role, and as such would be able to affect the game world and agents in different ways. An example of this type would be a game where three players, a detective, a policeman, and a consultant, must work together to solve a murder mystery and apprehend the culprit.

3.2.3 Multiple Competitive Teams

Competitive team play consists of teams competing with each other to fulfill their own personal goals. Players are divided into teams, and work with the players in their team to win the game. A player would be able to easily differentiate between his or her teammates and other players and act accordingly. An example of such a game would be one where the first team to collect a certain number of items wins.

3.3 Previous Work

The previous work on TaleBlazer multiplayer was dedicated to the client/server architecture necessary to maintain a consistent, shared game world [4]. A consistent game world requires mediation by a central server. For the server to maintain a consistent game world for all players, all mobile devices must be able to communicate reliably with the server. This is a difficult task, as mobile devices by nature have intermittent connectivity problems.

TaleBlazer is capable of creating very basic multiplayer games where all players involved inhabit a shared game world despite connectivity problems. A shared game world means that each player sees the same game state and affects the same set of agents. Every game and its participating players are called "instances", and all multiplayer game instances are overseen by the multiplayer server. There can be multiple instances running at the same time, but players in a game instance can only interact with other players in the same instance. Players in two different instances of the same game will not be able to digitally affect the game worlds of other players. Like a single player game, each player is able to pick a role for himself and could explore the world on his own. Most importantly, game states of the players connected to the server are consistent. One action taken by a player is reflected in the worlds of all the others. For example, in a game where the goal is to pick up as many coins as possible, all players that are connected to the server see the same number of coins on their map. If one player picks up a particular coin, no other player can pick up that same coin. The coin disappears from the map of all players, not just the player who picked it up. If a player performs an action that reveals a number of new agents on the map, these agents likewise appear on the maps of all the different players.

3.3.1 Multiplayer Server

It is the responsibility of the multiplayer server to ensure that the game state seen by all players in a multiplayer game instance is consistent. The server maintains the information of all multiplayer games instances that are being played. All game files and changes to the game files are tracked by the server and propagated to the players' mobile phones.

The multiplayer server is written in Node.js, and properly facilitates communication between the players of a game instance. The server keeps track of the players in each game instance, differentiating between them using their TaleBlazer login ids and player ids [3]. As a result, the user was required to be logged in to the TaleBlazer app in order to play a multiplayer game. Because the TaleBlazer username was tied to the player id, however, a player could only join an instance once; he or she could not log in on multiple mobile devices and play the same game instance.

Because the player is walking around in the real world while playing a game, intermittent connectivity can be assumed. While the server does need to mediate actions between mobile devices to ensure synchronicity, it is detrimental to user experience for the mobile to wait for feedback from the server for every single event. Thus, the server need only mediate certain actions that would cause glaring inconsistencies in the synchronicity of the game world if carried out in the wrong order or by more than one player. Other actions can first be completed on the mobile and then updated in the game file by the server. The pick up action is an event that must be mediated by the server, as multiple players cannot pick up a single agent.

Advantages of Node.js

The implementation of the multiplayer server in node.js allows the server to handle multiple concurrent requests. An event loop in Node.js is single-threaded, meaning that other code cannot be executed in parallel [2]. At the same time, however, the server can still be listening for communication requests from different mobiles, as the backend of the server is still running even as an event loop is being executed. As a result, the server does not miss any requests while fulfilling a previous request, and there are not any concurrency issues while running an event loop. When the server receives a message from a mobile device, the server opens a queue for that particular game instance. Subsequent messages that are received will be stored in that queue, where they will eventually be processed and deleted.

3.3.2 Server/Mobile Interaction

The server and mobile communicate with structured messages [3]. Each message has a request number (or update number, in the case of the server). This number helps the server keep track of each mobile's game state, and enables the detection of asynchronous game states if a mobile's request number is not greater than the server's most recent request number. There are several types of messages that help the server and mobile figure out what actions need to be taken. It is through these messages that updates to the game file are made. The most common types of messages are as follows:

- Initialization Messages
- Update Messages
- Ping Messages

The Initialization Message

When the server first receives communication from a mobile device, it will receive an initialization message. This message tells the server details about the player and what game file and instance the player wishes to start or join. With the details from this message, the multiplayer server downloads the game file from the repository server and sends it to the mobile. The server then adds the player to that particular game instance.

Initialization messages are also used to detect if a player is trying to reconnect to the game. Thanks to the unique username associated with each player, the multiplayer server can determine if a player has lost connectivity and has already been in the game instance, and can reconnect him or her instead of instantiating a new instance.

Update Messages

Update messages are sent between the server and mobile when it is necessary to update the game file. Generally, the mobile device completes the action itself before sending the requested changes to the server for propagation. Certain actions, such as pick up, require the server to mediate the action. In this case, the mobile devices send requests to the server, and the server decides which device completes the action first based on a first-come-first-serve basis.

Rather than sending a new game file every time the game state is updated, the server sends the mobile the changes to the game state. These changes are first applied to the server's game file, and then are propagated to all of the players in the game. The mobile phone checks to see if the change was caused by itself so as to not repeat the same action twice.

The server and mobile devices also use update messages to ensure that each mobile device is up to date with all the changes. As previously stated, each message has an update number. If the mobile receives an update message from the server with an update number that is not consecutively after its most recent update number, it will know to request whatever range of updates it is missing. The server honors those requests and updates the mobile to the current game state.

Ping Messages

The purpose of ping messages is to ensure that each device is still properly connected to the server. If the mobile device has not heard from the multiplayer server within 15 seconds, it sends a ping message. If the mobile and server are still connected, the mobile will receive an acknowledgement message back from the server. Without a ping message, the mobile device would not be able to tell if inactivity from the server is due to lack of player activity or a connection failure. If a mobile device does not know that it is experiencing connection failure, it will continue to send messages to the server and update the local player's game world. This will drive the player's game world further out of sync from the global game world. The lack of acknowledgement from the server alerts the mobile device that it must attempt to reconnect to the multiplayer server as soon as possible.

Chapter 4

Multiplayer Game Mechanics

TaleBlazer's ability to support a variety of multiplayer games requires the design and development of new multiplayer game mechanics. At the commencement of this project, TaleBlazer only had the capability of making simple multiplayer games. In order to take advantage of the established shared world, player interactions must be extended, and new structures must be implemented. This thesis focuses on the ability to support team play within multiplayer games, choosing to implement and develop key components that would lead to a supporting infrastructure for the team object.

Motivations

The previous iteration of TaleBlazer multiplayer was sucessful on many fronts. It enabled the inclusion of multiple players in a game instance, and ensured that the game world remains consistent despite conflicting actions. It was, however, missing any significant in-game player interaction. Game designers were able to create games where players interacted by affecting the same set of agents, but players were unable to directly affect other players. Key aspects of a multiplayer game were unavailable to game designers; for example, players were unable to easily transfer items, and there was no concept of group competitive or cooperative play. In addition, game designers were only able to write scripts that affected the local mobile's player. In the editor, game designers must first create an object before it can be referenced in the script code. Because player objects are only created as players enter a game instance, they are not present at time of game creation and thus are not accessible in the editor. It is necessary to address these missing components in order to enable the creation of more interactive games. The addition of more player-to-player interaction allows designers to create games that focus more on teamwork.

To extend TaleBlazer multiplayer games in this manner, the following key functions were installed and stabilized:

- initial connection to a multiplayer game instance
- the give function
- the editor and mobile implementation of the "for each" block
- teams

The following sections describe each piece of functionality in more detail.

4.1 Starting the Multiplayer Game

When a user wants to play a multiplayer game, the game selection on the mobile is the same as a single player game. The player must input a game or instance code in the game code box, and the mobile will search and find the linked game. Unlike single player games, however, the mobile must communicate with the multiplayer server for every subsequent step. The multiplayer server is what downloads and provides the game file, and it is the multiplayer server that adds the player to the game instance. Before the player can enter a multiplayer game, he or she is taken to the role selection screen. In previous iterations of a TaleBlazer game, this is where the player would select his or her role in the game before joining the instance. Once the player selects his role, as can be seen in figure 4-1, he or she is entered into the game.

There were two main changes that had to be made to the initialization of multiplayer games in order to stabilize the addition of a player to a game instance. One was to redefine the method by which the player is identified and is entered into the

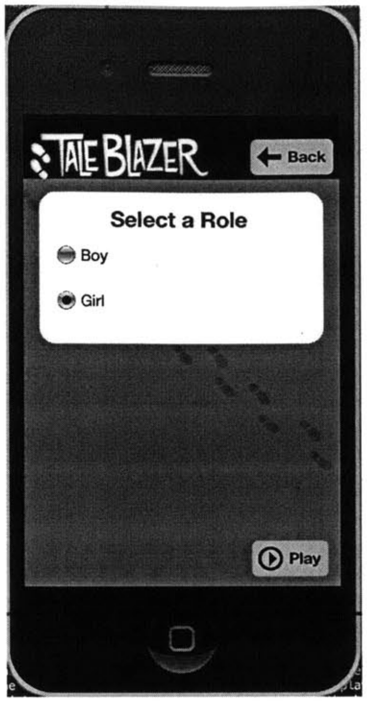


Figure 4-1: This is the role selection screen.

game. The other was to prevent the premature evaluation of code. Both will be further described in the next two subsections.

4.1.1 Server/In-Game Player Identification

Previously, it was necessary for the player to be signed in to his or her TaleBlazer account in order to play a multiplayer game. The mobile phone kept track of the current session, and uses the account username to add and identify the player within the game file. Because all TaleBlazer account usernames are unique, this was an effective way to keep track of the different players on mobile devices, as there could never be a repeat username.

The major drawback to this form of identification was that everyone who wanted to play a multiplayer game had to have a TaleBlazer account. The primary reason to have a TaleBlazer account is to create games on the editor. The majority of people who are just casually interested in playing a game or who are playing a game while visiting a particular location do not have a TaleBlazer account. Thus, in order to ensure that any user could play multiplayer games, it was necessary to come up with a new form of identification.

Usage of Device ID

Instead of trying to assign a unique identifier to each user, the server utilizes the installation-specific GUID (globally unique ID) as an identifier. Both Android and iOS-based phones have a unique, randomly generated ID that is consistent over a particular session. Android phones have a unique device ID that is tied to the device and is only changed in the event of a factory reset. The ID on iOS phones is instead tied to the particular installation of each application [1]. This ID is reset in the event of an application reinstallation. For the likely duration of a multiplayer game, however, the device ID would remain consistent.

When first connecting to the multiplayer server, the mobile phone uses its GUID as its user ID and its username. The server then uses this user ID to ensure that the player is not already in the specified game instance. Because this ID will always be unique between different phones, every player in the game will have a different identification.

In-Game Username

While the GUID of devices allows the server to keep track of the players in a given instance, it is not a human readable format. Thus, on the role selection screen, the player is asked to choose an in-game username. To prevent repetition, the user's choice is compared against all the other usernames of the players already in the instance before the player is allowed to join. This username identifies the player in the interface of the game; the players never actually see the identifying device IDs.

The player is only asked to provide an in-game username on his or her first entry into a game instance. When a player is attempting to join an instance, the server first checks to see if the mobile's device ID is already present within the instance's player list. If it is, it signifies that a player is trying to rejoin an instance he or she was previously in, and already has declared an in-game username. As a result, the player does not have to reselect a username and is instead taken directly to the game

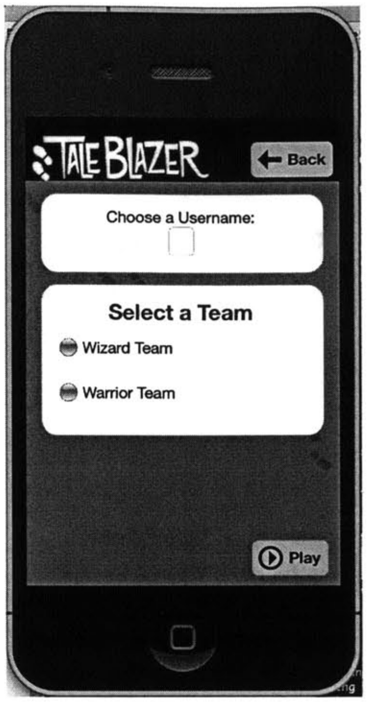


Figure 4-2: The player must first select a username before entering the game.

UI.

4.1.2 Queuing Update Messages

A mobile device begins receiving update messages as soon as the mobile device has established a connection. When receiving a message, the phone will parse the message to determine what process needs to be run. If the update request is foreign, meaning that the local player ID does not match the request player ID of the message, the phone will locally run the code. This occurs regardless of the current mobile's status.

This proves problematic when the role selection page is taken into account. The role selection page now contains multiple player-defined options that must be selected before the player can be fully assimilated into the game. After the player selects his or her player options, the mobile will fully launch the game, declaring and populating lists necessary for the game in the "launchGame" function. However, because these variables are only populated after the player leaves the role selection page and enters the game, it is possible to get an update request before properly entering the game. This will result in an error on the local mobile phone, as it will be unable to complete the update without the necessary structures.

Establishing a Message Queue

To prevent update messages from executing prematurely, the mobile phone should only run commands after the game has been launched. Any commands received from the server beforehand should be saved in order of occurrence. When connecting to the multiplayer server, the phone initializes a queue to store updates. If the mobile has not yet launched its game, the phone will stash the update into the queue instead of running the code. After the game is started, the phone will empty and run the commands that have been stored in the queue. This ensures that the updates will properly run in chronological order and that the local phone's game state will be synced with the world when the player enters the game.

4.2 The Give Action

One of the key aspects of a multiplayer game is the ability to interact with other players. The give action allows a player to exchange an agent with a different player and introduces new strategies into multiplayer games. An example game where this action is useful is a team versus team game where players on a team are each a different role. If there exists an agent with an action that can only be performed by a player of a certain role, the give action makes it possible for a random player to pick it up without having to worry about his or her role. He or she could then give the agent to the appropriate team member and the team would then have access to its full functionality.

There were two main parts to the implementation of the give action: the communication between server and mobile during the action and the presentation to the players within the mobile interface.

4.2.1 Server Communication

In order to ensure that giving agents behaved like expected, it was necessary to decide how much the server needed to moderate the action. The give action ideally behaves similarly to the pick up action: the agent is moved from a location to a player's inventory. Unlike the pick up action, however, there is no room for potential conflict between the actions of different players. The agent is already located in one particular player's inventory. Thus, only that player can affect the agent at any point in time. Thus, there is no need for the multiplayer server to mediate the invocation of the give action.

Checking Mobile Connectivity

Because of the intermittent connectivity of mobile devices, it is possible that either the giver or the intended receiver will lose connection to the server in the middle of the give process. It was necessary to decide how the server handles this situation. One possible solution is for the server to determine the connection of both players before it carries out the give action. Figure 4-3 demonstrates the workflow for this option. After the giver selects to which player he or she wishes to transfer the agent, the mobile attempts to contact the server to propagate the give action. There are two points of failure here: either the giver's local mobile or the receiver's mobile is disconnected from the server. In either case, the mobile would retry the connection multiple times before declaring the action failure and informing the player. This method prevents sending an agent to a disconnected player, but has the drawback of a feedback wait time. While this organization does not detrimentally affect the experience of a temporarily disconnected receiver, as he will not see any feedback from the give action, it forces the giver to wait for feedback before he or she can progress with the game.

Minimal Mediation

While it is necessary to ensure that the game world remains synced and that agents are not spontaneously lost, it is also necessary to provide a seamless game

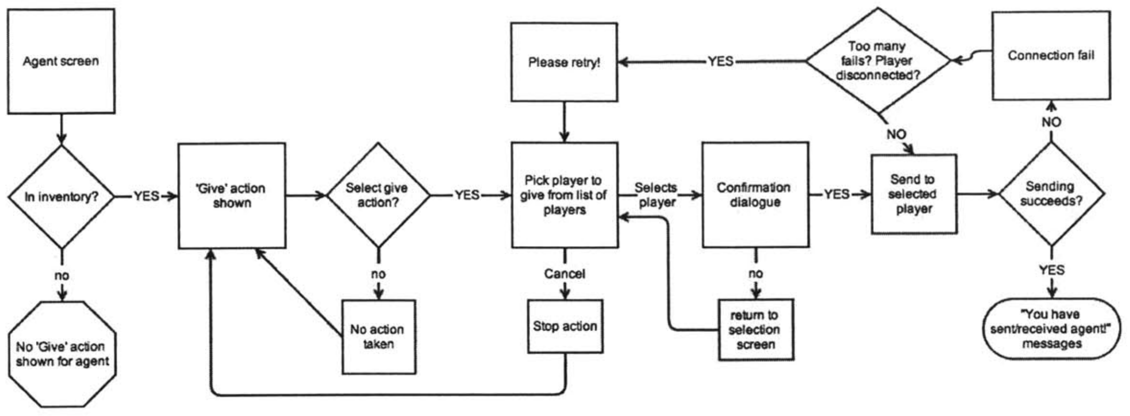


Figure 4-3: This is the workflow for a connection check during the give action.

experience to the player. It is inadvisable for a player to be waiting for feedback from the server before the game can progress. Thus, this iteration of the give action opts for minimal mediation by the server. The agent that is being given always reaches the inventory of the receiver regardless of the receiver's status. This makes the giver's experience feel fluid and uninterrupted. When the giver selects the player that he or she wishes to give the agent to, the local mobile will try to send the server a request to move the agent to that player's inventory. The server will update the game file, moving the agent. If both players are still connected to the game, then the give action is immediately successful and they both receive a notification of agent transfer.

In the event of a temporary disconnection from the giver's mobile phone, the request for a give action will be propagated to the server after reconnection. The agent will no longer be available to the giver, but the receiver will not be able to see the agent in his or her inventory. When the giving device reconnects and updates the server, the server will propagate the give function's resulting location to the other mobile phones, and the receiver will be notified of the agent transfer. While the giver's local world will be temporarily out of sync from the game world, the asynchronous time it takes for the mobile to reconnect will not negatively affect the player's experience.

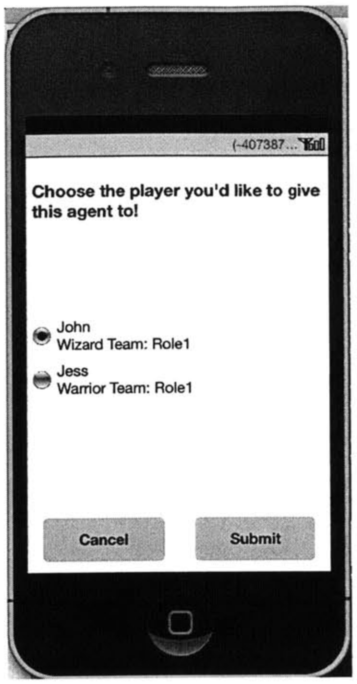
The temporary disconnection of the receiver's mobile phone is handled in a similar manner. The giving device will send an update message to the server containing the give command. The server will then propagate the message to all mobile devices with the exception of the receiver's disconnected device. The receiving device will receive the propagated update upon reconnecting with the server, and the player will then receive a notification about the transfer.

The drawback to this current method is the case where a player has disconnected and left the game for good. Because of the way multiplayer games are set up, players are not removed from the player list when they disconnect or leave a game. Thus, the server cannot determine whether a player is temporarily or permanently disconnected. If a giver were to select a permanently disconnected player as the receiver, the agent would be inaccessible for the rest of the game. This problem should be solved in future iterations of the give action.

4.2.2 Mobile UI

The interface of the give action must allow the player to easily and intuitively select another player to receive the agent. Because the give action is an action that is attributed to an agent, it is treated the same way as other agent actions and is displayed as a button on the agent dashboard. When the user taps the give button, a player selection screen is presented to the user. As displayed in figure 4-4a, the user sees the different selection options as radio buttons detailing the player in-game usernames and roles. In the event that the multiplayer game has teams, the players' teams will also show up in the selection options. The user can leave the player selection screen at any time, canceling the give action. The agent is not given to the selected player until the user clicks submit. The player options will only list players other than the user, as it does not make sense to allow the user to self-give agents. There are two other special cases that the player selection page has to take into account: if a new player has joined the game but has not fully set player traits such as role or team, and if there are currently no other players in the game.

If a player has joined the game, he will be added to the game file's player list. He is then accessible as an object within the game file. However, he will not have any defined traits until after he selects them on the role selection page. As a result, any

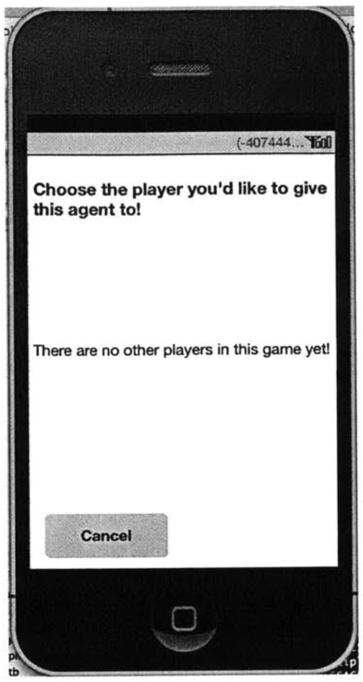

(-407387...

Choose the player you'd like to give this agent to!

John
Wizard Team: Role1

Jess
Warrior Team: Role1

(a) Options to pick receiving player


(-407444...

Choose the player you'd like to give this agent to!

There are no other players in this game yet!

(b) No players are in game warning

Figure 4-4: Screenshots of the player selection screen

attempt to access his traits will cause an error. In addition, it does not make sense for a player who is still deciding his username and role to receive notification that he has received an agent. Thus, the radio options on the player selection screen will not display any player that has not been fully defined within the game.

It is possible for the first player in a multiplayer game to have access to an agent with the give action before any other players join the game. There will then be no players to populate the player selection screen. In this case, the screen will notify the user that there are no other players in the game instance, and will not allow the user to proceed with the give action. An example of this case can be seen in figure 4-4b.

4.3 "For Each" Block on Editor

In the previous iterations of the TaleBlazer editor, there was no way for the game designer to reference players. Unlike other objects, player objects are created dynamically during game runtime. Game designers can only reference objects that are already created at design time. Thus, only the local player was an option for block commands– all other players objects were created after game design and the designer could not access them. For example, in the event that a gate agent required every player to have an item type before a new agent could appear, there is no way for one player to bump the gate and have the gate check the inventories of all players. Instead, the gate agent would have to have a trait keeping track of how many players meet the requirement. Each player in the game would have to bump the gate agent, and each player's mobile would change the gate's trait. Only after the last player bumped the gate and set the gate's trait to the desired amount could the new agent show up. The "for each" block would enable a check of all players' inventories when a single player bumps into the gate agent, as it allows the inventory check script to walk through all players.

There was also no method that allowed the designer to affect multiple objects with the same set of blocks; it was necessary to have the same set of block commands attributed to every object that they were meant to affect. For multiplayer games, it is useful to have the ability to act on multiple players with one action. With the addition of a "for each" block in the editor, game designers can write a script to walk through all of the players in a game. Inside this script, game designers could change the traits of all players that matched a specific criteria. For example, if a player with the pirate role scattered treasure from the pirate treasure chest, the game designer could use the "for each" loop to walk through all players, and check if the player is a pirate role. If so, then the game designer could increase the trait "treasure" by one for those players. The "for each" loop allows one player's action to directly affect another player's gameplay.This creates a game world where one player's actions has significantly more effect on the rest of the game world.

4.3.1 Block Design

The idea behind the "for each" block is to allow the designer access to a list of objects on which his or her commands can iterate over. To properly convey the purpose of the block, there are two things that must be made clear to the game designer: the

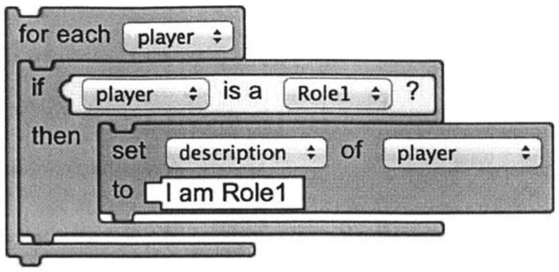


Figure 4-5: This is an example of the "for each" block with the predefined variable dropdown.

type of list that the block would be iterating over, and a reference back to the current object that is being acted on.

Predefined Variable Dropdown

The predefined variable design, as depicted in figure 4-5, does not allow for any customization by the user. There is only one dropdown containing the three different types of objects that the player can iterate over (i.e. agent, player, team). This dropdown selection also acts as the variable that represents the object being acted on. The variable will populate the dropdowns of any blocks that are run within the "for each" block. An example of this block usage is shown below.

As can be seen, the <player> selection in the "for each" block serves as a consistent representation of the object being acted on by the subsequent blocks. The "for each" block loops on a list of players, and each player object is referred to simply as "player" in the subsequent blocks. The "if" block will check if the current player is of role Role1, and, if so, change its description to "I am Role1" using the "set trait" block.

The main problem of this design is that it does not allow game designers to easily differentiate between different "for each" block variables, as they are forced to choose between three predefined variable names. With this design, nested "for each" blocks that iterate over the same object type are no longer possible. For example, if a designer wanted to compare all players with each other, he would need to nest a "for each" block within another "for each" block. Both blocks would have <player> as the argument. As shown in figure 4-6, it is confusing for the game designer to

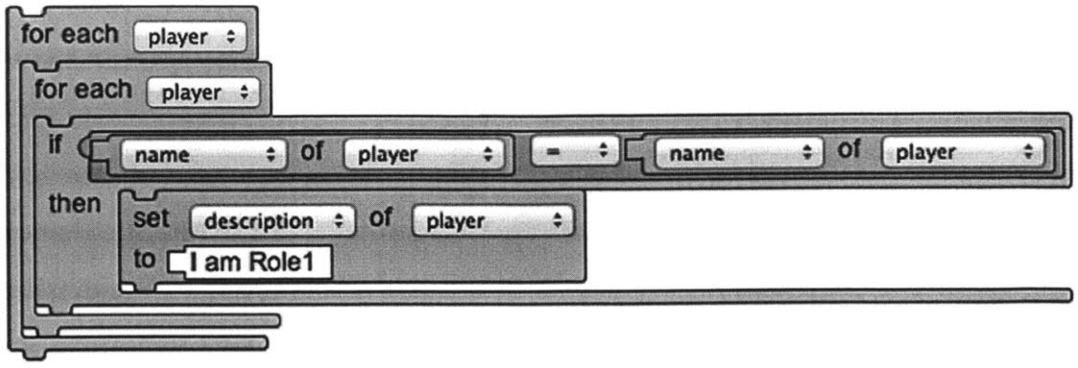

for each player
for each player
if name of player = name of player
then set description of player
to I am Role1

Figure 4-6: In this example, both "for each" blocks have the variable "player". This makes it impossible to tell which "player" variable in subsequent blocks refers to which "for each" block.

differentiate between <player> variables.

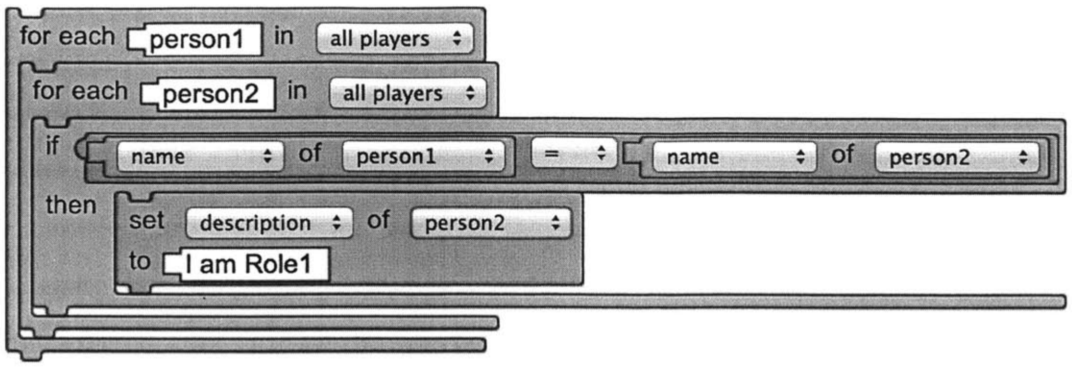
Player-Defined Variable

To allow easy differentiation between variables, the "for each" block design used in the editor has an input field that allows the designer to assign his or her own variable to the block. The variable input field contains a default string to make it clear to game designers that the field is a text field and not a slot for a block. The dropdown that follows in the block depicts the three different types of lists that the player can iterate through (i.e. all agents, all players, all teams). These two argument fields enable the block to keep track of what type of object the variable is referencing.

Because the variables used in each "for each" block is different, nesting blocks is no longer an issue. After the two variables are declared in the two "for each" blocks, both variables will appear as options in the dropdowns of blocks that are nested inside the "for each" blocks. Figure 4-7 is an example of nested "for each" blocks. Here, it is simple to tell which variable belongs to which "for each" block.

4.3.2 Block Dropdown Population

When nested within "for each" blocks, certain children block types should have the "for each" variable in their option dropdown lists. A list of all blocks able to have the variable option in their dropdowns is located in the Appendix. To understand how the "for each" block implements this behavior, it is necessary to have a basic



for each person1 in all players
for each person2 in all players
if name of person1 = name of person2
then
set description of person2
to I am Role1

Figure 4-7: An example of the nested "for each" block.

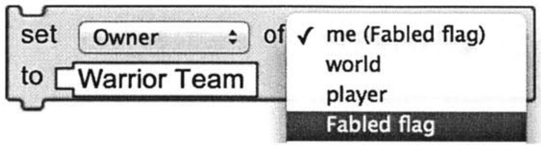


Figure 4-8: An example dropdown with both agent types and string types.

understanding of how TaleBlazer blocks populate their option dropdown menus.

Block Dropdown Structure

When defining a block for use in the editor, it is necessary to declare what types of objects can populate that block's option dropdown list. Each option in the list is composed of two parts: the name string, which is the string displayed in the dropdown, and the value, which is hidden from the user. The value of the option is either the name string itself or a numerical ID referring to a specific object in the game file. Figure 4-8, for example, depicts a "set trait of entity" block. The options in this block's dropdown consist of <me (Fabled flag)>, <world>, <player>, and <Fabled flag>. The strings visible to the game designer (e.g. "player", "Fabled flag", etc.) are the options' name strings. The values of these options are all different, and depend on the type of the option. While the <player> option is referring to type player, the value of the <player> option will be the string "player", as player is not yet an object in the game file. The value of the <Fabled flag> option, however, will be a number. This number is the object ID that is associated with the Fabled flag agent in the game file. The Fabled flag agent has already been defined in the game file at game creation, and thus has an object ID.

The dropdown is populated based on the types that are allowed to be included in the block dropdown list. If, for example, one of the allowed types of a block were the agent object, then the dropdown would be populated with all the agents in the game. Each of the agents would have an option in the dropdown, where the option name string was the name of the agent, and the option value was the ID associated with the agent object. If the dropdown type were instead of type player, the option name string would be "player", but the option value would be the string "player" rather than an ID. Both cases are handled differently during block evaluation.

Variable Option Population

The "for each" block introduces a new option type that can be included in block dropdown lists. The variable option type is used to display the appropriate "for each" variable to the user and to keep track of which "for each" block's variable the current block is referring to. The name string of the option will be the variable name that the user inputted in the parent "for each" block. The value of the option is a string that is composed of three parts: the term "var" to distinguish it as a variable, a letter to distinguish the type of object the variable refers to, and the variable name.

Before the variable option can be created, it is necessary to see which parent "for each" block the variable type is referring to. To do this, a block must look through its parent blocks to find the "for each" block it is nested in. However, due to the structure of TaleBlazer's scriptblocks, each block only records its direct ancestor. Thus, the block must iterate up its parents until it sees a relevant "for each" block. Once it finds this parent, the block will use the "for each" block's variable and type arguments to determine the name and type of the variable. With this information, the block will create the variable dropdown option. The process is repeated for any other "for each"blocks that the block is nested in. An example of this can be seen in figure 4-9. In this example, the parent "for each" block's variable name is <person>, and its type is player. The "set trait" block is using both these arguments to create the variable dropdown option. The variable option name string is "person" (the parent

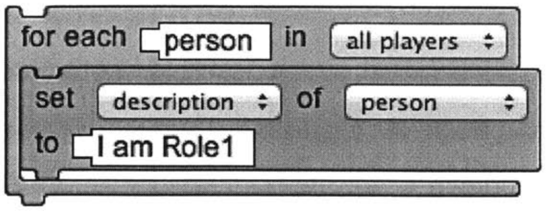


Figure 4-9: An example of the dropdown option of a block nested within a "for each" block.

"for each" block's variable name) and the option value is "var:p:person", where "p" is the letter representing the player type.

4.3.3 Block Evaluation

Block evaluation is done on the mobile device. When the "for each" block is evaluated, it first checks to see what type of object it is iterating. This is achieved by accessing the type argument of the block, which stores agent, player, or team. Based on the selected type of the block, the mobile will pull a list of all objects of that type in the game file. For example, if a "for each" block has a type argument <player>, the mobile will declare a list of all players within the game. This list is what the mobile iterates over.

It will then determine the variable name using the "for each" block argument. The device maintains a global dictionary of all of "for each" variables. Each dictionary key is a variable string composed of three parts: the "var" string, a letter based on variable type, and the variable name. This dictionary key definition is identical to the previously mentioned dropdown option value, and serves as the link relating the parent "for each" block to its children blocks.

Each time the mobile iterates through the object list, it will assign the current object's ID to the "for each" block's variable key in the dictionary. This dynamic key-value assignment is how the mobile tracks which object the code is acting on. The mobile code will then evaluate the blocks that are nested within the "for each" block. If any of the children blocks have the "for each" block's variable for an argument, the block can simply look up the variable key in the dictionary to discover the current object ID. It will then be able to run its code using the appropriate object ID. This process will be repeated until the mobile is finished iterating through the object list.

4.3.4 Applications to Single Player Games

While the "for each" block was created out of multiplayer necessity, it is a useful block for single player games as well. Because single player games do not have the concept of multiple players or teams, those options are hidden from the block's type dropdown when the game is single player. The only option that appears in the type drop down of the single player version of the block is the "all agents" option. This allows game designers to loop through all agents in the game.

4.4 Teams

Teams are an integral part of the multiplayer game world. With team-based gameplay, players are more inclined to socialize and work together, whether for the purpose of achieving a common goal or simply to beat the other players. The addition of teams in TaleBlazer opens up the opportunity to create an entire new class of games that benefit from collaborative and competitive aspects.

There are several components to the team implementation:

- team structure in the game model
- configuration of teams in the editor
- mobile user interface

The design and implementation of these team components will be discussed in the following sections.

4.4.1 Team Structure

The team object has a variety of attributes. Like agent and player objects, teams have traits and actions attributes. Another important team attribute is the various members on the team. Teams will record the players that have joined the team using a list. This list stores numerical player IDs in order to keep track of team members.

Joining a Team

Players choose a team on the role and username selection screen when they join a game instance. The team cannot be changed once the player has chosen. When a player enters the game, his or her player ID is added to the chosen team's player list. Similarly, the player object's team is set to the team's ID.

Propagation Across Mobiles

Being on a team means that multiple players will share and be able to view information about the team. This information should be kept constant across all devices. Thus, changes in traits should be propagated to all the players. For example, a team score should always be constant across the devices of all team members, regardless of who caused a change in score. If a team member picks up an agent, the team inventory should be updated and all players should then see that agent within the inventory.

4.4.2 Teams in the Editor UI

To have games with teams, the game designer must be able to declare and customize teams on the editor. This process must be consistent with the rest of the editor so that users already familiar with making TaleBlazer games do not have to learn something new. Thus, the editor must be able to handle two different states: a multiplayer game with no teams, and a game with teams.

No Teams

A newly created multiplayer game defaults to having no teams. This is essentially a player versus the world game, where players each act on their own to fulfill their own goals. This type of game is akin to having an infinite amount of teams, each with one player. Since having a large number of one-player teams in a game is both impractical and unreasonable, the game will instead have no teams.

If a game has no teams, it is unnecessary to show the blocks and the editor space, as there would be no teams to modify. Thus, as can be seen in figure 4-10a, the editor will hide the detail view, and instead warn the user that there are no teams in the game. This view will be shown anytime there are no games in the editor, including the case where the user deletes all created teams.

Teams in Game

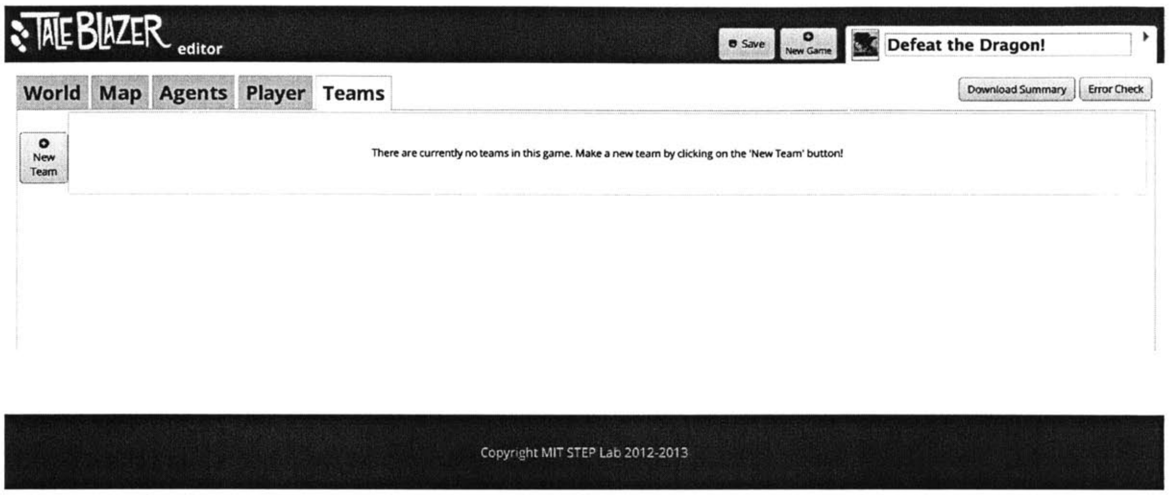
If there are teams in a game, then the team dashboard on the editor looks and functions the same as the role dashboard. Here users will be able to customize their teams, adding traits, actions, and scripts using the provided block drawers. Game designers should be able to add teams to a game at anytime. If there are no teams in the game, the rest of the editor dashboard will be loaded when a team is added.

Modifying Mobile Tabs

Logically, the team tab on the mobile should only be shown if a particular player has been assigned to a team. This can only occur if the multiplayer game has teams. Thus, whenever it is loaded, the mobile tab configuration widget checks to see if there are teams in the game. If there are no teams in the game, the option to show the team tab on the mobile will not be available to the game designer on the UI. The presence of the team tab on the mobile will be marked false even if it was previously true. This situation can occur if the designer had previously created teams and wanted to show the team tab on the mobile, but later deleted all teams. The UI option to show the team tab will appear when the designer adds teams to the game, but the team tab will not automatically be included in the mobile. This mobile tab functionality is consistent with the rest of the game, as most mobile tabs are not shown on default.

4.4.3 Team Blocks

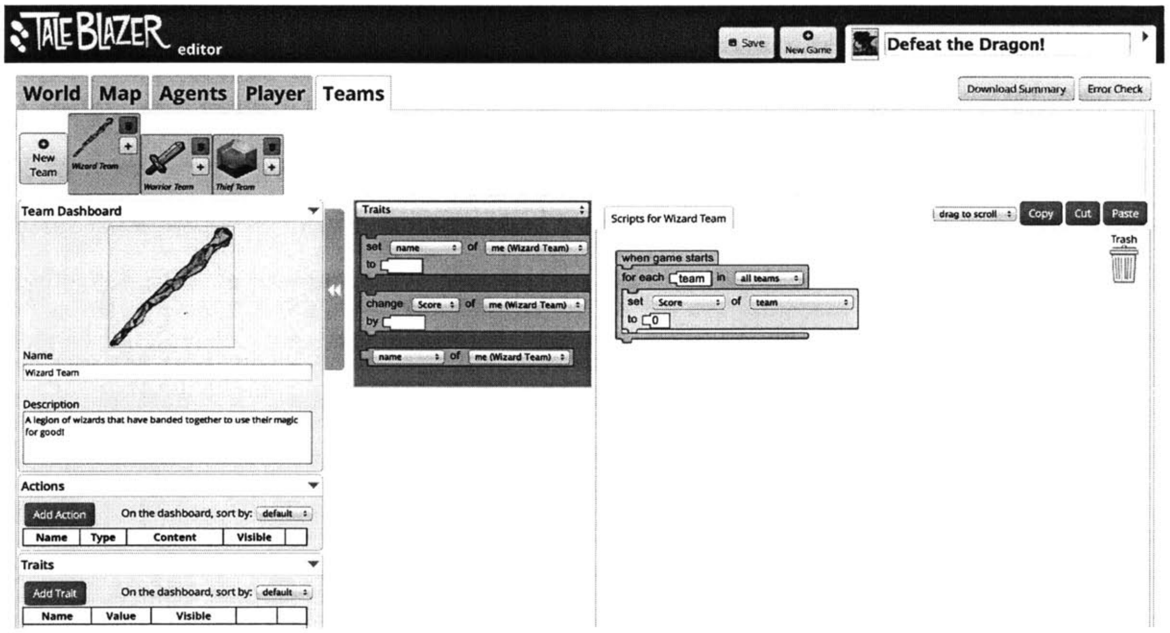
With the inclusion of teams in the editor, it became necessary for there to be blocks that utilize the team object. The "for each" block allows game designers to iterate



There are currently no teams in this game. Make a new team by clicking on the 'New Team' button!

Copyright MIT STEP Lab 2012-2013

(a) Team tab when there are no teams



Name: Wizard Team

Description: A legion of wizards that have banded together to use their magic for good!

Actions

Add Action On the dashboard, sort by: default

Name	Type	Content	Visible

Traits

Add Trait On the dashboard, sort by: default

Name	Value	Visible

```
set name of me (Wizard Team) to
change Score of me (Wizard Team) by
name of me (Wizard Team)
```

```
when game starts
  for each team in all teams
    set Score of team to 0
```

(b) Team tab with team added

Figure 4-10: Screenshots of the team tab on the editor

over team objects. Aside from the "for each" block, there are two blocks that deal with teams:

- the "player on team" block
- the "is agent in specific inventory" block

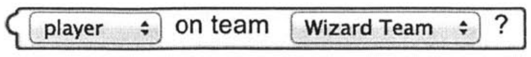
Both blocks help game designers build games that effectively use teams.

"Player on Team" Block

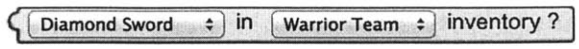
Since players now have a team attribute, it is useful to be able to figure out the team of a specific player. The "player on team" block has two dropdowns (figure 4-11a). The first contains all possible player objects. Here, the only possible objects are the local player and the potential player variable. The second dropdown contains all team instances in the game, including the team variable. On evaluation, this block will simply check if the selected player ID is stored within the player list of the select team, returning true if so, and false if not.

"Agent in Specific Inventory" Block

While there already exists an "is agent in inventory" block, this block only checks the local player's inventory. In previous iterations of TaleBlazer games, this block was sufficient, as the local player was the only object with the concept of an inventory. However, in multiplayer games, especially with the addition of teams, there are now multiple objects that have an inventory. In a multiplayer game with teams, all player objects and all team objects had an inventory. Thus, it was necessary to make a block that allowed the user to specify which object inventory to look in. As can be seen in figure 4-11b, this new block has two dropdowns. The first dropdown decides what agent to check for, while the other lists all objects in the game that could possibly have inventories to search through. The second dropdown is populated with the player and various team arguments.



(a) "Is player on team" block



(b) "Is agent in specific inventory" block

Figure 4-11: Examples of the new team blocks

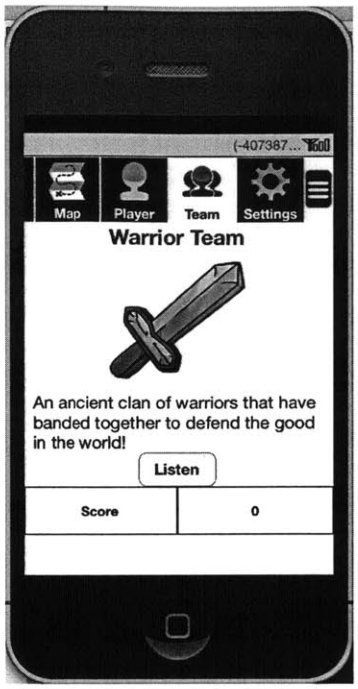
4.4.4 Teams in the Mobile UI

Because there is a lot of information about teams that must be displayed for the player, the mobile interface must be clear and easy to navigate. It is necessary to balance player convenience with current interface consistency. We evaluated two ways to organize team information: integrating team information alongside the existing similar player information or consolidating it to a single location.

Integrated Team Information

One method to display team information was to integrate it with the corresponding player information. Since a player is a part of a team, team traits are equivalent to player traits, and information about agents in the team inventory should be viewed similarly to those in the player inventory. For example, on the player dashboard, the table containing player traits would be followed by another table that contains team traits. The inventory tab would no longer contain only the local player's items; it would contain all the items picked up by the team by order of player.

Though this makes most bits of team information easy to locate, it is not consistent with the rest of the application interface. In the current interface, information that is located on a particular dashboard is only about that dashboard's object. Including information about the team object on a dashboard about the player breaks consistency. In addition, the integration of team information into the player dashboard would create large interface differences between games with teams and games without. This could potentially be jarring for users who are used to player TaleBlazer



Warrior Team

An ancient clan of warriors that have banded together to defend the good in the world!

Listen

Score	0

Figure 4-12: An example of the team mobile tab

games but are new to games with teams.

Isolated Team Tab

In the current iteration, team information is presented to the player in its own tab. This team tab contains the team dashboard, and is based on the preexisting player dashboard. As can be seen in figure 4-12, this dashboard displays all the information pertinent to a team, such as traits and actions. The team information was all consolidated into one tab in order to be consistent with the rest of the application. Though it might be easier for the player to see team traits along with player traits or the team inventory as along with the player inventory, it does not make logical sense with regard to the rest of the application.

Team Representation on Map

On the "Map" tab, the location of the local player is represented by a yellow dot. In previous multiplayer games, the other players in the game were represented by a fuzzy blue dot. With the division of players by team, it is useful to be able to

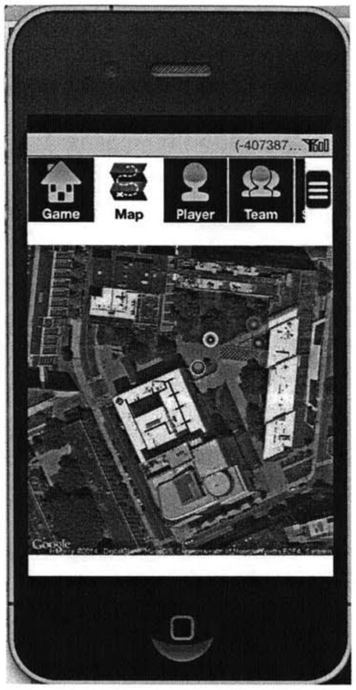


Figure 4-13: An example of the icons depicting players in the game.

distinguish the locations of team members from all other players. Thus, members of opposing teams have differently colored icons from members on the local player's team. In the current iteration, these icons are magenta.

Chapter 5

Play Testing

This chapter details the many testing steps performed on the various functionalities implemented for multiplayer. Testing was used to ensure the stability of the new implementations as well as to gather feedback about design decisions made and how they affected user experience. Both types of testing served to guide and refine the development of the multiplayer functionalities.

5.1 Mobile Functionality Playtests

There were a number of tests ran throughout the development of the different functionalities. The primary test method for each multiplayer function is detailed in the following subsections.

5.1.1 TaleBlazer Team Give Test: Hot Potato

Hot Potato was a game created for the purpose of testing the newly implemented give action. It was the first test to also be held on a larger scale, involving 8-9 people, each with their own devices. The purpose of the hot potato game was simply to ensure that the give action would work as players expected. There were several requirements for the Hot Potato test to be successful:

- Neither the server nor any of the mobile devices should crash

- Multiple, simultaneous give actions should be successful
- Receiving players should receive the correct agent
- The give action should behave as expected when players temporarily disconnect from the server
- Notifications should appear when appropriate and display the correct information

The Hot Potato test was run two different times. The first run unearthed several errors.

In the first runthrough of the Hot Potato test, the biggest and most significant error that occurred was that many of the mobile phones crashed right after connecting to the multiplayer server. The error was caused when the mobile tried to handle update messages before the game interface was initialized. This would occur while the player was on the role selection page. This test was the driving factor behind the implementation of the update queue discussed in Section 4.1.2, and helped solve most of the connection errors regarding premature code evaluation.

After fixing the errors found in the first test run, the second run confirmed that the give action indeed behaved as the players expected.

5.1.2 "For Each" Block Test: Defeat the Dragon!

Defeat the Dragon was a game designed specifically to test the functionality of the "for each" editor block. It was necessary to see that each type of object list would be correctly iterated over and that nested block actions would be correctly handled. The test specifically looked for the "for each" block to cause these behaviors:

- Agents meeting a specific requirement should become visible on the map
- The location shift of all non-creature agents when a particular agent was bumped
- The setting of traits based on the traits of all players

- The movement of all players to the "end game" region when the game was beaten

This test was run with multiple groups of 3 different players, each with his or her own mobile device. While the "for loop" block worked as expected for most requirements, it at first could not change the regions of all players. A later run of the test game with multiple members of the TaleBlazer team demonstrated the fixed ability to change the region of all players based on a single player's action.

5.1.3 Team Functionality Testing

There were two parts to the team functionality testing. It was necessary to ensure that the editor interface behaved as expected by the game designers. It was also necessary to ensure that the mobile implementation of teams behaved as expected. Separate measures were required to test each.

Editor Interface Testing

The testing of the editor interface required making multiple different types of multiplayer games. In testing the team functionality of the editor, proper behavior under the following requirements was expected:

- The team tab would only appear for multiplayer games
- The team tab would properly handle the transition between no teams in game and teams in game
- The mobile tab customization would only have the team option if there are teams in the game
- The new team blocks showed the correct options in their dropdowns

Testing of the interface was done by a single person. The editor never showed any significant problems with team functionality.

Mobile Functionality Testing

The mobile functionality testing involved the creation and running of many different multiplayer games. These games primarily focused on ensuring that the propagation of trait changes was logical and consistent among different scripts programmed into the editor. They also heavily tested the implementation of the new team blocks, making sure that they returned true under the proper circumstances. Testing also made sure that team information was properly and logically displayed for the user.

5.2 User Feedback Playtests

In addition to functionality testing, the multiplayer functionality on both the editor and mobile-sides underwent testing for feedback from potential users. The purpose of these tests was to ensure that the layout and implementation of the functionalities were intuitive for the end user to understand and use. In the test, the tester was shown an example game in the editor and was walked through how all the scripts were used. He was then shown the same game on the mobile, and was able to play around in the game world. The tester was then given the opportunity to create his own game in the editor and the subsequently play it on the mobile. The focus was to see how he felt about the layout of the editor and the presentation of the mechanics on the mobile device.

A discussion with a local Massachusetts teacher produced significant and useful feedback. Feedback regarding the current implementation was positive. A short explanation of each of the features was enough for him to understand, and few questions were required. Much of the other feedback received included recommendations for new features that could be added to expand TaleBlazer multiplayer. These suggestions for future work are covered in the next chapter.

Chapter 6

Future Work

The purpose of this thesis was to expand the functionality available for TaleBlazer multiplayer games. Though a solid framework has been laid out, there are still a large number of improvement and functionalities that can be implemented in the future in order to create multiplayer games with more interesting mechanics and player interaction.

6.1 Device Identification

While the usage of device ID is a useful way to recognized mobile devices, TaleBlazer is not allowed to collect any data that can potentially identify a user. In the future, this manner of identification should be modified.

6.2 Implementation of Idle Players

There is a possibility that players will disconnect from a game instance and not return. Because of the way a multiplayer game instance is stored, players that have disconnected are not removed from the player list. This can cause confusion, as players still inside the game instance can perform actions that target a player (e.g. the give action). A potential solution to this issue would be allowing the server to track the most recent time that a player has been active in the game. Players that have exceeded a reasonable time of inactivity could be labeled as idle. The other players would then be able to tell which players were idle and could take action accordingly.

6.3 Modifications to "For Each" Block

Though the "for each" block successfully allows game designers to walk through lists of objects, there are some fixes that can be made to make the block more versatile.

6.3.1 Unique Variables

Due to the current way that variables are stored and tracked, it is essential that all "for each" block variables are globally unique. In the future, there should either be a warning when repeat variable names are used or a way to track identical variable names. This will allow the game designer to use whatever variable names he or she wishes instead of having to keep track of previously used variables.

6.3.2 Visibility Propagation

Future iterations of TaleBlazer should consider the implementation of a game designer's ability to propagate the visibility of traits and actions. Blocks in the "Looks" drawer of the editor should contain an argument that allows in game scripts to change the visibility settings of other players. For example, a game designer should be able to decide that one player's action shows a team action to all players in the team.

6.4 Location-Based Mechanics

While TaleBlazer is an AR location-based game, there are no location-based mechanics that can be used to effectively extend multiplayer games. The following subsections describe two mechanics that would enable the creation of more interactive games.

6.4.1 Location-Based Give Action

Currently in this implementation of TaleBlazer, the give action between two players can be carried out despite player location. An interesting mechanic that could be toggled by the game designer is to require close proximity between players before the give action is allowed. This would force players to come together physically in order to exchange items and adds the dimension of relative location into multiplayer games.

6.4.2 Team Zones

An interesting addition to team-based gameplay would be the concept of team zones. These zones would simply be predefined subareas in the map, and would essentially function as "home bases" for a team. In the team zone, team members could potentially be allowed to access certain actions or could be immune from certain negative effects, while other opposing players would be put at a disadvantage. The implementation of this mechanic would first require the ability to divide the map into various subareas.

6.5 Direct Player-to-Player Interaction

While there are grounds for player interaction and ways for players to influence the game worlds and experiences of other players, it would be interesting for there to be a more direct method of player interaction. Such methods would allow for games that force players to be more social.

6.5.1 Messaging

One way to make player-to-player interaction more interesting is to have players be able to directly broadcast messages to each other. This would enable players to communicate even if they are not physically next to each other. Such communication would allow for new strategies and cooperation between players. For example, a player that discovers the location of a clue code agent could message the location to the player with access to the clue code. Instead of the player needing to take time and physically deliver the desired message, the mobile phone could receive the message in an instant.

6.5.2 Players Bumping Players

An interesting mechanic would be if players could bump into other players like how players bump into agents. If the local player were to come into contact with a different player in the game map, the player's dashboard would display on the local mobile, and the local player would be able to see information about the other player. This would enable the different players to interact with each other in the game world, and players would be able to perform actions to directly influence other players.

An example game would be a "freeze tag" type of game. The game would consist of two different teams. Once the local player bumped another player, he or she could either freeze or unfreeze the player based on team membership and player traits.

6.6 Teams Modifications

Though the framework for team support in TaleBlazer multiplayer games has been set, there are features that can be added to extend the team capability.

6.6.1 Team Tab Additions

In the future, the team tab should also display the members on the team as well as the team inventory. The tab, however, should not grow to be cluttered, as the player should be able to easily find this information. Since the player does not necessarily need to see the members or inventory of his or her team every time he or she frequents the team tab, it is possible to create buttons that would call up these lists in new screens. This would serve to provide the necessary team information and to maintain the uncluttered and orderly team dashboard.

6.6.2 Customization of Player Icons

In future iterations of TaleBlazer, icons representing different team members should be customizable by the game designer. It should also be possible for game designers to toggle the inclusion of distinguishing icons, as it is possible for a game designer to want a game in which membership to a team is anonymous.

6.6.3 Methods of Choosing a Team

A game creator might want to vary the way that membership to a team is assigned. For example, he or she might want the players randomly assigned to teams when joining the game in order to vary the combinations of players. In the future, it should be possible for the game creator to decide how players are assigned to teams. This would be an interesting addition to the customizability of teams.

6.6.4 Dynamic Teams

Currently, the only way for a multiplayer game to have teams is for the game designer to create and define them in the editor. Thus, the number of teams cannot be changed, and players must choose a team from the predetermined set of teams. While this is a fine design for small-scale games, it can potentially run into problems with a large-scale game. If a game designer is not sure how large his or her game might be, he or she might not want to set a fixed number of teams.

One way to combat this is to develop the ability to create dynamic teams that can be defined and formed as more and more players join a game. Dynamic teams can have requirements that, once fulfilled, mark the team as filled and open the creation of a new team. Such requirements can include a maximum number of players or a maximum number of each type of role that players could choose from. The formation of new teams as players populate the game ensures that the game remains scalable regardless of how many players enter the game world.

Chapter 7

Contributions and Conclusion

7.1 Contributions

With help and contributions from the TaleBlazer team, this thesis has helped create new functionalities to expand TaleBlazer multiplayer games. It redefined the way that players were identified within multiplayer games and introduced the give action to increase player-to-player interaction and to make games more social. The implementation of the "for each" block gave game developers an easy way to affect multiple objects and players at once. Lastly, it developed support for team functionality, allowing for the creation of entirely new multiplayer games. These implementations have become a foundation on which future work can build to further expand the multiplayer experience.

7.2 Conclusion

The multiplayer functionality of TaleBlazer is a rapidly expanding area with a large space in which to grow. The multiplayer mechanics introduced and implemented for this thesis have helped introduce entirely new and different player interactions within TaleBlazer multiplayer games. While basic multiplayer games with interesting player interaction can now be created within TaleBlazer, it is the author's hope that the functionality implemented in this thesis will lead to more complex mechanics that allow for even more interactive and meaningful multiplayer TaleBlazer games.

Appendix A

Tables

Table A.1: Blocks that can have an option of type variable in their dropdowns

Block Definition	Valid Option Type
Game Drawer	
Include Agent From World	Agent
Exclude Agent From World	Agent
If Agent in World	Agent
If Player is a [Role]	Player
If Player is on Team	Player Team
Looks Drawer	
Show Action of Entity	Agent Team
Hide Action of Entity	Agent Team
Show Trait of Entity	Agent Team
Hide Trait of Entity	Agent Team
Movement Drawer	
Move Entity to Region	Agent Player
If Entity in Region	Agent Player
Move Agent to Location	Agent
Set X/Y of Agent	Agent
Change X/Y of Agent	Agent
X/Y of Entity	Agent Player
If Agent in Specific Inventory	Agent Player Team
Traits Drawer	
Set Trait of Entity	Agent Player Team
Change Trait of Entity	Agent Player Team
Trait of Entity	Agent Player Team

Bibliography

[1] Appcelerator documentation. http://docs.appcelerator.com/titanium/3.0/api/Titanium.Platform.

[2] Adron Hall. Understanding the node.js event loop. http://strongloop.com/strongblog/node-js-event-loop/.

[3] Sarah E. Lehmann. Taleblazer: Implementing a multiplayer server for location-based augmented reality games. Master's thesis, Massachusetts Institute of Technology, September 2013.

[4] Michael Paul Medlock-Walton. Taleblazer: A platform for creating multiplayer location based games. Master's thesis, Massachusetts Institute of Technology, June 2012.

[5] MIT Scheller Teacher Education Program. Mitar. http://education.mit.edu/projects/mitar-games.

[6] MIT Scheller Teacher Education Program. Mystery at the museum. http://education.mit.edu/ar/matm.html.

[7] MIT Scheller Teacher Education Program. Outbreak @ mit. http://education.mit.edu/ar/oatmit.html.

[8] MIT Scheller Teacher Education Program. Starlogo tng. http://education.mit.edu/projects/starlogo-tng.

[9] MIT Scheller Teacher Education Program. Taleblazer. http://taleblazer.org.

[10] Scratch. scratch.mit.edu.