TaleBlazer: Implementing a Multiplayer Server for Location-Based Augmented Reality Games

by

Sarah E. Lehmann

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

September 2013

Author ... [illegible] ....
Department of Electrical Engineering and Computer Science
August 23, 2013

Certified by. [illegible] ....
Professor Eric Klopfer
Director, MIT Teacher Education Program
Thesis Supervisor

Accepted by .... [illegible] ....
Professor Albert R. Meyer
Chairman, Masters of Engineering Thesis Committee

TaleBlazer: Implementing a Multiplayer Server for Location-Based Augmented Reality Games

by

Sarah E. Lehmann

Submitted to the Department of Electrical Engineering and Computer Science
on August 23, 2013, 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 engine that allows users to both design their own games as well as play them on mobile devices. This thesis explores the addition of a multiplayer option that would allow users to design and play games involving multiple players in a single game world. It details how such a system would be set up to use with the existing TaleBlazer code and provides some results from initial tests of this prototype.

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

Acknowledgments

First, I'd like to thank Eric Klopfer and Jason Haas for teaching the educational games class that opened my eyes to the work being done on educational games. I'd also like to thank Eric for providing all of the opportunities in this lab and guiding the teams towards success.

I'd like to thank Lisa Stump for her seemingly endless supply of patience as well as her constant drive to make TaleBlazer successful. I'd also like to thank Judy Perry for helping me set goals to make the most of my time and get the most out of my project.

Additionally, my project wouldn't be where it is today without the support of the UROPs (Stephanie Chang, Gerardo Gomez, Jordan Haines, Cristina Lozano, Kayla Meduna, Harry Rein, and Benji Xie) and fellow MEng Fidel Sosa. Also, to my replacement, Tanya Liu, who will be taking over the multiplayer server and taking it to loftier goals: best of luck.

I'd like to thank Patrick Winston and Rob Miller for giving me the opportunity to TA for their classes and thus allowing me to work towards my MEng. In addition, I'd like to thank all of my fellow TAs for their support and good company when times were rough.

Many thanks to my family and friends who helped and supported me through these past five years at MIT.

Finally, I'd like to thank Paul Medlock-Walton, who laid the scaffolding for this system and was a great source of knowledge, support, and good company over the course of the project.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

With mobile technologies becoming accessible to a wider range of people, applications utilizing tablets/phones are becoming more commonplace. One genre of application, augmented reality, consists of applications that feature ways of extending the player's environment via such things as visual overlays or location-triggered pop-ups. Given this wide availability, one natural extension of mobile augmented reality applications is to use them to support learning, allowing students to become more immersed in given scenarios. The Scheller Teacher Education Program (STEP) has done many projects in the field of educational mobile technologies. One such project, known as TaleBlazer, focuses on location-based augmented reality games. One of the current goals of TaleBlazer is to extend games to be multiplayer.

1.1 Motivations

Multiplayer games will enhance TaleBlazer, as they will allow players to have more shared experiences/interactions and further require players to work together in the context of the games. For this type of game to be feasible, there needs to first be a stable multiplayer server that can allow multiple players to play games similar to the already existing single player games. Once this stable server exists, there is a whole new realm of interesting multiplayer game mechanics that can be added to further expand the TaleBlazer design library.

1.2 Chapter Summary

Chapter 2 will provide background on the history of TaleBlazer, as well as details about its current state. Chapter 3 will give context for this new multiplayer functionality and provide guarantees that a successful multiplayer server should be able to uphold. Chapter 4 will detail the new user interface elements that aid players in playing multiplayer games. Chapter 5 will explain how the infrastructure for playing multiplayer games is set up and how information is passed between mobile devices. Chapter 6 will describe the different tests performed with multiplayer games. Chapter 7 will suggest new features that can be added to improve multiplayer games.

Chapter 2

Background

TaleBlazer is less of a standalone game and more of a powerful engine that allows both players to play games as well as designers to create their own games. TaleBlazer games are often created to enhance an experience at a particular location. For example, the TaleBlazer project is currently partnered with Old Sturbridge Village, a historical site in central Massachusetts. They have a historical economics game that visitors can play, which further teleports visitors into the early nineteenth century environment.

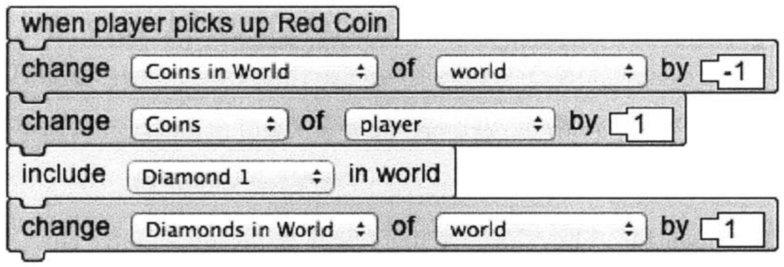
The software itself consists of three parts–the editor interface, where users create games, the mobile interface, where users play games after they are created, and the servers, which serve as both a repository for games and a mechanism for hosting multiplayer games.

2.0.1 Editor

This software allows the user to play the role of the designer and create his/her own games [7]. While the game itself is played on a smart phone/tablet, the designer defines the game via a web interface by selecting a location for the game to be played and adding agents to a virtual map of this location. From there, the designer can further define the agents and add more options to the game as detailed in Appendix A.

Blocks-Based Programming

In addition, using a set of programming blocks, the designer can specify how these defined options are affected through interactions with different parts of the game. These blocks allow a low barrier to entry for game creation and are used to define any dynamic aspects of the game (agents appearing/disappearing, traits changing, etc). For example, in Figure 2-1, picking up the Red Coin decreases the world trait, "Coins in World", by 1, increases the player trait, "Coins", by 1, makes Diamond 1 appear, and increases the world trait, "Diamonds in World" by 1. More examples of TaleBlazer blocks appear in Appendix A.



```
when player picks up Red Coin
  change Coins in World of world by -1
  change Coins of player by 1
  include Diamond 1 in world
  change Diamonds in World of world by 1
```

Figure 2-1: Example Blocks in the Editor Interface

2.0.2 Mobile

The mobile side of TaleBlazer contains all software that runs the user created games. Because the mobile code must be able to run any user created game (including all of the script blocks), it must be sufficiently general to support all possible game operations.

The mobile interface contains a number of tabs, only some of which are shown to the player (based upon which tabs are selected by the designer). The two basic tabs are the "Home" tab, which shows the player high level details about the game and the "Map" tab, which displays the game area as well as all visible agents and the location of the player.

2.0.3 Servers

In the software, there are two distinct sections of server code–one for the multiplayer server and one for the repository server. The multiplayer server, described at length in this paper, coordinates multiplayer games by propagating messages from/to the players of a game as players interact with the game world. In addition, there exists a repository server that stores all of the necessary files for games (pictures, videos, and the game files themselves).

2.1 Previous work

Within the STEP lab, there have been previous versions of similar software. Many of the functional components of TaleBlazer were derived in theory from components of previous iterations.

2.1.1 MITAR

Similar to TaleBlazer, MITAR also allowed users to play location-based, augmented reality simulations. However, game designers had a smaller set of game elements to choose from and had to create them without a blocks based programming language (early games had to be created directly in the MITAR-specific markup language). In one of the MITAR games, "Environmental Detectives", players tried to detect the source of a toxic spill by questioning virtual characters and taking virtual samples in the environment to see how far the spill had spread [2].

2.1.2 StarLogo

Also developed within the STEP lab, StarLogo was a tool for creating simulations to help understand systems, such as bird flocks and traffic jams [5]. The newer version, StarLogo TNG (The Next Generation), added 3D graphics and sound [6]. StarLogo utilized a similar blocks-based programming language to TaleBlazer and was one of its predecessors.

2.1.3 Other STEP Lab Multiplayer Games

The STEP lab also previously tested two other multiplayer AR games. One game, Outbreak @ MIT, was a strictly indoor game that was played on Pocket PCs. Players had to work together to interview virtual agents and gather items to control the spread of an epidemic [3]. The other, POSIT (Public Opinions of Science using Information Technologies), was a game that revolved around a hypothetical policy question (ex. Should MIT build a level 4 biohazard laboratory nearby?). Players would assume different roles (ex. biologist, resident) which initially have varying degrees of support for this policy and they work to gather evidence to convince the other players and NPCs to sway their decision. Upon being swayed one way or the other, NPCs would change their tone/message to reflect this new position [8].

2.1.4 TaleBlazer

TaleBlazer itself is a descendent of MITAR and incorporates the ideas of other blocks-based platforms such as StarLogo. Currently, many functionalities (detailed in Appendix A) are implemented for single player games, both on the editor and in the mobile code. There also exists a defined model that is used to provide structure for the game files.

2.1.5 Multiplayer Server

There had been previous work done to start the multiplayer server prototype. The general protocol for sending messages was set up, as well as a server that could connect with mobile devices. Multiple players could be connected to a single game world, but the process to do so was unwieldy and very susceptible to human error. Also, due to intermittent connectivity, some messages were dropped and there was no system place to detect whether messages had reached their destination (and resend them if they hadn't). The game states of the players could very easily and quickly diverge, thus making any sort of real world multiplayer gameplay impossible.

Chapter 3

Intro to Multiplayer Games

In TaleBlazer single player games, each player has their own device and is looking at their own game world. Many people may be playing the same game, but nothing anyone does will affect anyone else's game. All players can be playing the same game at the same time and, assuming there's a coin in the game, each player can pick up their own version of the coin. But what if the game designer only wanted one coin for everybody and only the first person to pick up that coin would get it? For this, the game designer needs additional functionality and the ability to have many players in a single shared game world.

3.1 Possible TaleBlazer Multiplayer Games

In order to define multiplayer mode goals, there was an initial exploration of the possible use cases. This allowed for a more clearly defined vision moving forward.

3.1.1 Player vs. the World

This type of game involves many players interacting with a single game world without any clearly defined alliances. Therefore, all players should be viewing the same game world and their actions should affect everyone else's world. Though there may be competitive/cooperative elements, there is no other defined subset of other players that a player is said to be working with or against.

Example Games

- There are many agents in the world. Everyone tries to collect the agents, and the one with the most agents/highest score is considered the winner.
- There are many enemies in the world and different ways to defeat them. Whoever can defeat the most enemies wins.

3.1.2 Single Cooperative Team

Games like this are strictly cooperative–each player in the instance is working toward a common goal. The players may have different roles allowing them to play different parts in reaching this goal.

Example Games

- Players can have different roles which allow them to perform different actions on agents. For example, some players can open treasure chests and others can pick up coins.
- There could be a epidemic style game where agents start out randomly infected with a virus and the virus can spread through players making contact with infected agents (and players can get infected). Players would have to work together to combat and stop the spread of the virus.
- There is a certain agent that requires team members to coordinate actions. For example, to defeat a dragon, players would have to coordinate "taunt", "taunt", "stab".

3.1.3 Multiple Competitive Teams

Multiple Competitive Team games consist of multiple players on competing teams. Each team could be working towards the same goal, or different teams could have different goals (possibly goals with conflicting tasks). Note that Player vs. the World is essentially a subset of competitive teams–each player is merely a team of one.

Example Games

- There could be a quest-style game where players have different roles in their respective teams that aid them in completing these quests. Teams compete to finish the most quests/get the most points from quests.
- There could be an environmental disaster game where one team is working together to spread the disaster and another team is working to stop the disaster from spreading.

3.2 Multiplayer Goals/Guarantees

The multiplayer server keeps track of all "instances" of games being played. An instance consists of a game and all of the players currently playing that game in the shared game world.

The end goal of a multiplayer game is to give all participating players in an instance a shared game world. However, because TaleBlazer is run on mobile devices which aren't yet known for their consistent internet connections, the server will not be able to keep all players synchronized at all times. Instead, the system ensures that all players who can connect to the server will eventually see the same game world–though it makes no guarantees about the amount of time to possible synchrony.

However, the intermittent connectivity should not hinder the game's progress. Therefore, most elements of the game are not mediated by the server unless it is crucial. This allows for players to make most changes to their local game world instantaneously and have them integrated into the server later.

The one crucial action the server must mediate is agent pickup. Because agents are often very integral to games, the server must be able to guarantee that no agents are ever duplicated. That is, once a player picks up an agent, no other player can also pick that agent up – thus ensuring that only one copy of each agent can exist in the world.

Eventually, multiplayer will be able to replicate and extend upon all single player elements and blocks(as listed in Appendix A).

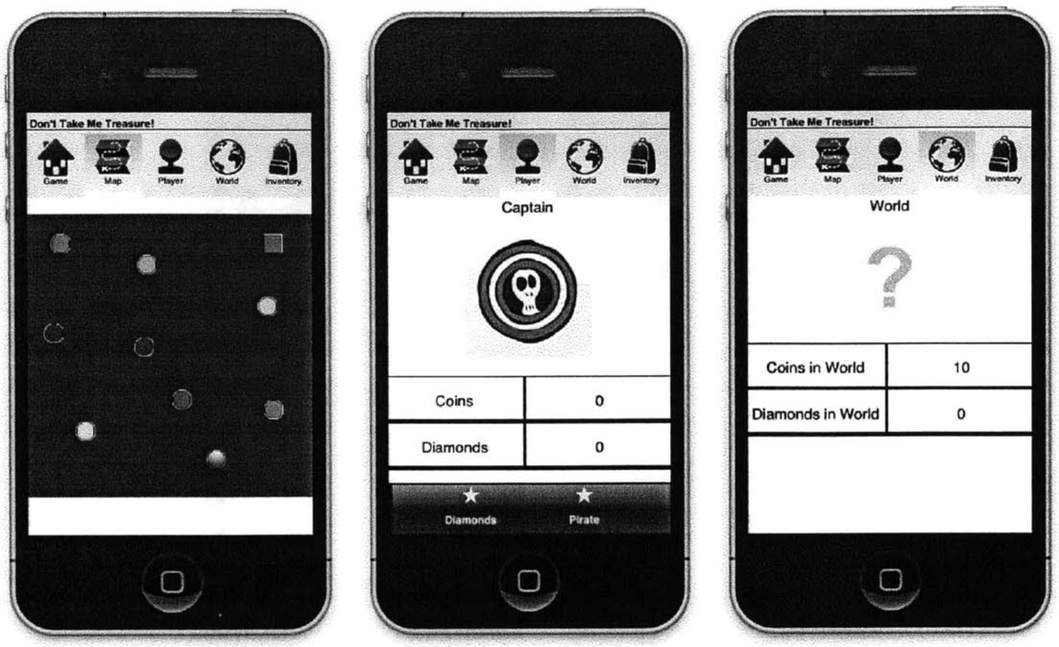
3.3 Example Multiplayer Game: "Don't Take Me Treasure"

Created for a formal play test in summer 2013, this game serves both as a demo of the multiplayer features present after this project as well as a test of those features. In addition, it highlights certain features that distinguish multiplayer games from single player games. The game is played on an indoor region and can thus be played anywhere. The premise of this game is that the players are a group of treasure seekers guided by their captain (the person administering the game/test). There are roles defined for each type of player.

The captain starts up the instance and instructs each of their treasure seekers to connect to that instance. When they enter the instance, they can all confirm that the map tab has 10 objects, the player tab reads that the player has picked up no coins and diamonds, and the world tab says there are 10 coins in the world (Figure 3-1).

The captain can then instruct the treasure seekers to pick up the coins, either one at a time (with one player trying to pick up each coin) or with all players trying to pick up a single coin. This demos/tests both successful pick up of agents by a single person and the consequential disappearing of those picked up agents from other players' maps. In addition, after all of the coins have been picked up, players can confirm that they got the expected coins by looking at the values in the player/world tab, as well as agents in their inventory. This ensures that traits have also been properly propagated.

Following that controlled test, the captain can select to add more agents to the world via actions on his/her player tab (also shown in Figure 3-1). After clicking on



Coins	0
Diamonds	0

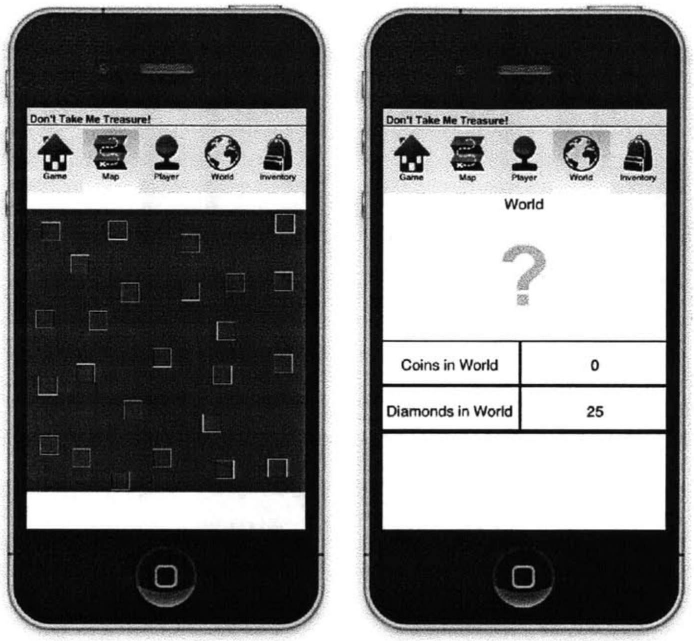
Coins in World	10
Diamonds in World	0

Figure 3-1: Don't Take Me Treasure, Start State

the "Diamonds" action, 25 diamond agents will be included in the world, and the captain can confirm that each player can see the new agents on the map (Figure 3-2).

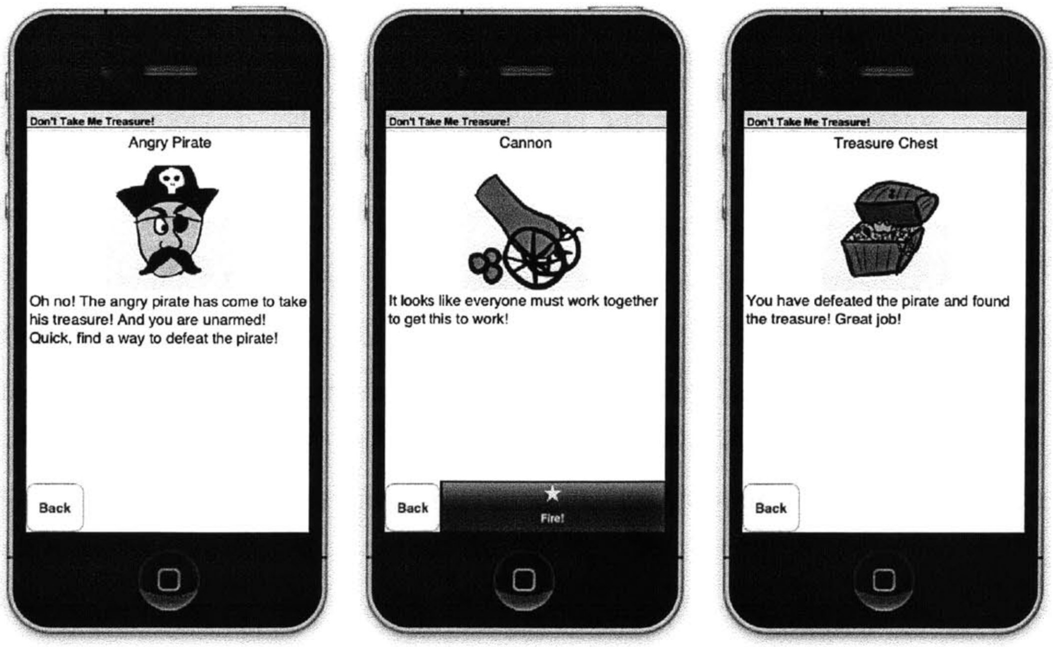
The players can try to pick up the diamonds as they see fit–it turns into a competition of sorts, especially since the players don't know what diamonds the other players are trying to pick up. With enough players, this also turns into a reasonable load test for the server.

Finally, the captain taps the "Pirate" action, causing two agents to appear on the map–an Angry Pirate and a Cannon. Players tap on both, but the cannon seems to be the only option to defeat the pirate. Each treasure seeker can only hit "Fire!" once, and the cannon has a hidden trait, "Number of Fires", which counts how many times it has been fired. Once it reaches a predetermined number of fires, it excludes the angry pirate from the world and includes the treasure chest (if there aren't enough treasure seekers to reach the predetermined number, the captain can fire multiple times to get up to the required number). From there, the players tap on the treasure chest and are informed that they have completed the game (Figure 3-3).



Coins in World	0
Diamonds in World	25

Figure 3-2: Don't Take Me Treasure, World after Diamonds



Oh no! The angry pirate has come to take his treasure! And you are unarmed! Quick, find a way to defeat the pirate!

It looks like everyone must work together to get this to work!

You have defeated the pirate and found the treasure! Great job!

Figure 3-3: Don't Take Me Treasure, Pirates Sequence

Chapter 4

Multiplayer User Interface

For single player games, TaleBlazer offers many affordances for how to progress gameplay. Many of these are also useful with multiplayer games. However, there are some new user interface (UI) flows/elements specifically for users looking to play multiplayer games. These elements seek to enhance and clarify the multiplayer experience.

4.1 UI for Starting MP Games

4.1.1 First Player

The user interface to start a multiplayer game looks much like that which starts a single player game, but with a few key differences. Both types of games can be found in the standard game selection screen, though once tapped, they show some key differences. There is an icon and text to indicate multiplayer, and the text on the button suggests that the player must be connected to the server to play (Figure 4-1).

From there, the player is taken to a loading screen while the player connects to the server.

After the player receives the game file, if there are multiple roles or scenarios for that particular game, a role selection screen is automatically opened, as in single player. Note that only the first player has the capability of selecting the scenario–this is because each instance must only have one scenario, and once the game is started

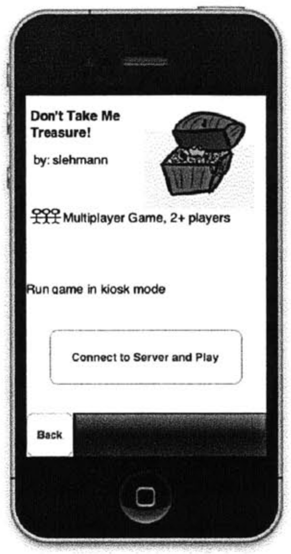


Figure 4-1: Game Page Screen for a Multiplayer Game

the scenario cannot be changed.

Now the player is taken back to the loading screen with more information populated (if there are not multiple roles/scenarios, the player will be taken directly here). From there he/she can reselect a role or scenario, start the game (if applicable), or go back (thus disconnecting from the server and failing to start up the game) (sequence shown in Figure 4-2).

4.1.2 Additional Players

If a player wants to share a single player TaleBlazer game that they're playing, they share a Game Code. Each TaleBlazer game has a unique code that, when typed into the game code screen, will take the player directly to the game's page (as in Figure 3-1). However, though multiplayer games have game codes that direct to their game pages, a player would generally be more interested in sharing the world that he/she is playing in. In this case, they would share the multiplayer "Instance Code" which refers to a specific instance of the game (starting with an "i" instead of the Game Code's "g"). The additional player(s) would enter this code from the same game code

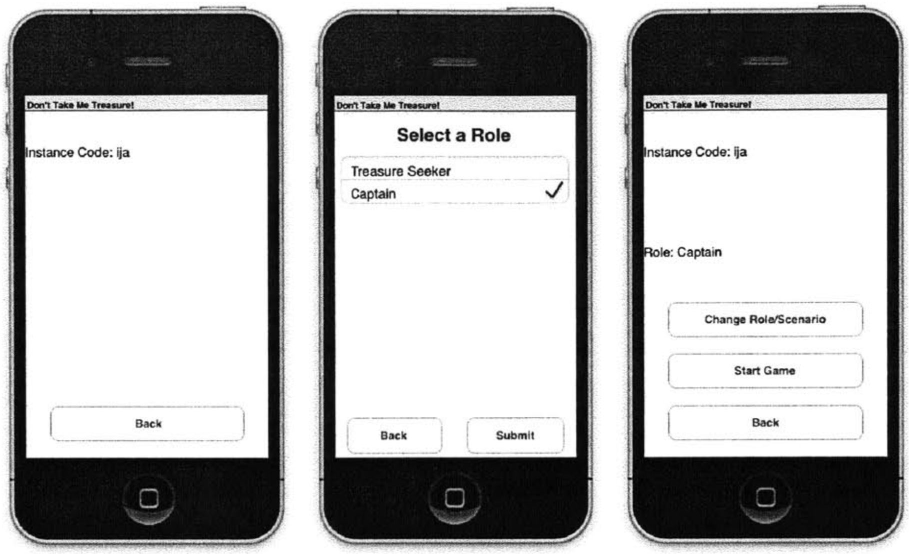


Figure 4-2: Sequence of Screens Starting a Multiplayer Game

interface.

Once entered, the instance code will take a player directly to the instance, connecting to the server and bringing up a similar loading screen to the one for first player. If there are multiple roles, the player will be prompted to choose a role. If the first player has started the game already, the player will be taken directly to the game. If not, the player will see a populated screen with the message "Waiting for First Player to Start".

Because there are possible blocks to be run when the game starts, the game should only be started once, and currently that duty lies with the first player. Unfortunately, if the first player were to disconnect before he/she starts the instance, any subsequent players will not be able to play in this instance and will have to start a new instance in order to be able to play.

If the player has already connected to the instance, he/she can select "Connect to Server and Continue" from the Game Page (the game will be stored under the "Saved Games" tab of the game selection screen).

4.1.3 Resetting/Deleting/Updating Games

Currently, there is no way to truly reset a multiplayer instance. Instead, if players wish to start a game over, they need to simply start a new instance. Pressing the reset button will remove all references to the particular instance in which the player has been participating for that particular game. Pressing the delete button also will not delete the instance. In addition to removing all references to the particular instance as during a reset, pressing the delete button will remove all metadata associated with the game, making it no longer available as a saved game. Unfortunately, there is not yet a way for the player to tell if the designer has updated the game since he/she started the instance. Instead, each instance that has been started will never be updated, and all new instances will have the properties of the game revision reflected.

4.2 UI during MP games

Because multiplayer games have more going on in general, it is crucial that there are also UI elements that aid the players in figuring out what events are happening and why.

4.2.1 Player Icons

Unless the game is utilizing an indoor region, in addition to the agents, the map also displays the icons of the other players. This allows those in the game to know where their collaborators/competitors are. It potentially allows them to see which agents are about to be visited (and possibly picked up) by noticing the players' proximity to those agents. In addition, these icons display the username of the player when tapped, so players can locate specific teammates if necessary (note that the other player icons appear as fuzzy blue dots) (Figure 4-3).

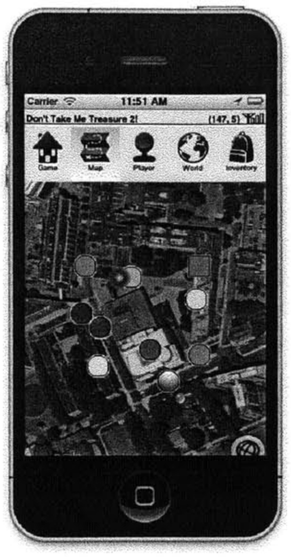


Figure 4-3: Player Icons on the Map

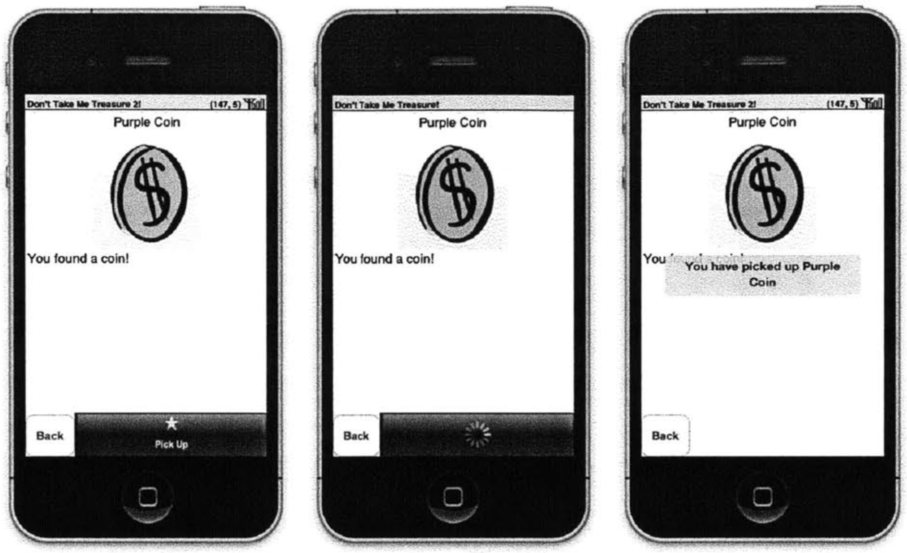
4.2.2 Agent Feedback

Because players need to wait for a server response to actually pick up an agent, the "Pick up" button changes to a loading animation when tapped, indicating that the device has registered the player's intent to pick up that agent. Once the the server confirms a successful pickup, the player sees a confirmation message regardless of what tab they are currently looking at (Figure 4-4). Other players who may be looking at that agent at the time of pickup will get a message saying that someone else has picked up that agent and the agent's dashboard will be subsequently closed. Finally, if a player is looking at an agent when it is excluded from the world (most likely due to actions taken by other players), he/she will get a message saying that the agent is no longer in the world and the dashboard will also be closed.

4.3 Summary of UI Additions

The TaleBlazer UI now supports the following:

- There is an interface for easily starting up multiplayer instances that is much



You found a coin!

You found a coin!

You have picked up Purple Coin

Figure 4-4: Sequence of Picking Up an Agent, with Feedback

less susceptible to human error, as well as a means for players to leave instances, start new instances, and delete references to instances on their devices.

- Players can see/identify all other players on the map.
- Players receive feedback about picking up agents, and if they're looking at an agent that is no longer accessible, it closes with a message indicating why it is no longer accessible.

Chapter 5

Server/Mobile Device Interaction

The multiplayer architecture is split up into two separate parts–the server code and the mobile code. The following chapter details how these parts of the code are set up and how they interact with each other, as well as how they resolve any differences that arise due to intermittent connectivity.

5.1 Server/Mobile Device Setup

The server is the primary location of all information about games being played – multiplayer game files aren't saved on the devices. Any changes that affect the game file must be sent to the server in order for those changes to persist across game play sessions. The game file on the server is the primary source for the state of the game, and all players save their game changes to and restore the game from the server.

Previously, the basic setup for the multiplayer server was mostly in place. The server was set up in a way that would fit the current code already running on mobile devices without any drastic changes, as it was very important to add this functionality without breaking what already existed in the code base. The server code itself was already started using node.js – a fitting choice given both its features and the fact that most of the rest of the TaleBlazer codebase is already written in JavaScript. In addition to the server setup, there was code for the mobile devices for connecting to that server and sending messages.

Node.js

Node.js is an asynchronous, event-driven platform which allows developers to write server-side Javascript. The main event loop is in a single thread, and concurrent worker threads process events as they come in [1]. Node.js allows developers to quickly set up a TCP server which listens for connections (in the case of TaleBlazer, from mobile devices).

Server Code

Primarily, the server needs to manage many mobile device connections. In order to do so, there must be a unique identifier for each player connected to the server. In the case of TaleBlazer, it is the username/user id. All players must be logged in to connect to multiplayer instances. Each player with a distinct username can be actively participating in at most one instance at a time. For example, if a player logs in and connects to an instance with one device, he/she cannot also connect to that instance with a different device logged in with the same username.

Many data structures exist to keep track of the many instances/sockets on the server and are accessible by each socket opened: instanceIdToInstanceObject, gameFileUpdateQueue, clientUpdateObjects. The instanceIdToInstanceObject keeps track of all players in the instance. In addition to basic information of that player, each player object contains information about the number of the last update sent to that player, the number of the last update received from that player, and the time the player was last seen (if the player has disconnected). The gameFileUpdateQueue has a queue of updates to be sent for each instance. Finally, the clientUpdateObjects keeps track of all messages sent to the clients (again, organized by instance), which is necessary for resending messages.

Using the asynchrony of node.js as well as the defined data structures, the server can effectively receive, incorporate, and propagate updates. In most cases, when an update is received from a mobile device, it is incorporated into the game file and then the same update is propagated to all of the devices (including the one that sent the update). All devices incorporate this update visually (ex. moving a player icon, making an agent appear). Though the initial sender most likely already visually incorporated this update before sending the message to the server, it still runs the update again, as the order of messages the server received may have altered the result of applying this message.

Something that needed to be added to the server was error handling in general. Exceptions caused server to completely stop running, which was certainly not desirable functionality, especially with multiple instances running that could be causing the error. Everything is asynchronous, so the normal JavaScript try/catch functionality couldn't be used. In node.js, there is a way to catch all exceptions, but then it loses the context of error, making it nearly impossible to recover from. There is also the concept of domains, which are a way to group many I/O calls together (cite node.js docs). It is possible to catch and handle errors within a specific domain so that they are caught before the general "catch all" net. In TaleBlazer, it is necessary to catch errors related to trying to send messages to disconnected clients. Therefore, all of the write calls are now wrapped in a single domain, so if the server fails to write to a socket, it closes the socket and notes the last time that particular player was seen.

Mobile Code

Mobile devices need to be able to open up and maintain connections with the server. The mobile devices themselves are responsible for ensuring the server connection, so they should have mechanisms in place to ensure that they maintain connection with the server. Previously, the mobile devices could open up connections to the server and send messages, as well as retry to connect if they knew they were disconnected. Now, in addition to that, should a mobile device become disconnected from the server, it can reconcile updates with the server as well. It can also detect if the server has been receiving its messages and try to reconnect to the server if it detects that the messages are failing.

Shared Code

Because both the server and mobile devices need to consistently update the game file in the same way, it was simplest for there to be shared files between the server and the mobile code. Note that though the mobile code doesn't directly manipulate the game files in a multiplayer game, it does in the case of a single player game, process of updating the game file is the same for both single and multiplayer games. There are three files in particular that are shared. One file, updateState, is responsible for parsing all of the update messages and updating the game file itself. Another, mobileUpdate, visually updates the game on mobile devices. Though this code is less useful on the server, many functions in updateState call functions in mobileUpdate to update the UI itself, so the references must be preserved.

The third file, blockEvaluator, evaluates all defined script blocks in the game. There are few instances where blocks will actually need to be evaluated on the multiplayer server itself (the only instance being the "when picked up" blocks, which is run on the server, as the server mediates pick up of agents). It is important that blocks only run in one place the results sent in messages (rather than blocks running in all locations). For example, the random block would simply cannot be run in multiple places and produce the same results each time, and if this were done in a multiplayer context, the players' games would quickly diverge.

5.2 Messages

The server communicates with the mobile devices using a set of predetermined messages. These messages contain information about all game events, enabling all devices connected to an instance to converge to a single shared game world.

5.2.1 Parts of a Message

Messages consist of two main parts. The first is the main information that the message contains (parameters for initialization, parameters for updating the game file, etc).

The second is the updateNum parameter (or requestNum in the case of the mobile messages). This number increases sequentially with each message sent and gives an order to each of the messages, and each message from a single entity has a unique updateNum or requestNum (for example, there would be a message #5 for the server and for each player). This allows the server and clients to apply each message in the sequence which they were generated and also helps the them quickly notice missing messages due to loss of contact with each other.

5.2.2 Initialization Messages

Before the player starts playing the game in an instance, it is important to initialize the connection between the server and the player's device. Therefore, a series of init messages open up the connection and deliver the game file so the player can start playing the game in the instance.

First Connection with the Server

When the first player taps the button to start an instance of a particular game, the mobile device first calls the repository server to get the next available instance code. Once the mobile device has received that instance code, it opens up a connection with the multiplayer server. The multiplayer server then recognizes this connection and immediately sends an "initServer" message. This message has no other information other than its type (initServer), and it indicates to the mobile device that the connection was successfully started.

As a response to the initServer object, the mobile device will then send its own init message. This message will include the player's username and user id, the id of the game that will be played, the instance id, the last updateNum of server that the mobile device received (should be -1 at the start), and the requestNum of the message (starts at 1).

The server then takes that information and initializes the instance by downloading the corresponding game file from the repository server and adding the player to the instance. The game file is sent to the mobile device in a "gameFileAndStateServer" message, which also includes an updateNum, and playerId. From there, the player can choose the role/scenario if relevant and start the game. Additional players connecting to the instance for the first time will go through a similar messaging process, except now that the instance is already initialized, the server won't have to make extra calls to the repository to get the game file and will instead send the updated game file reflecting the current state of the game.

Reconnecting to the Server

Inevitably, a player is going to have to reconnect to the server after their device gets disconnected from the server. Should the mobile device knowingly lose connection, it tries to reconnect every ten seconds, giving up after 20 consecutive failed tries. In this case, instead of adding a new player, the server goes through the process of reconnecting that player to the instance. Any messages missed are resolved by resending them as described in section 5.2.4.

5.2.3 Game State Update Messages

After the connection with the server is established, there needs to be a way to pass along all that is necessary to update the game file as events occur in the game. For this, there are game state update messages. The bulk of the messages sent between the mobile devices and the and the server are of this type. Previously, many game state message updates were included, but many needed to be improved upon (detailed in future sections). Each game state update message that originates on a mobile device has a "propagate" parameter which indicates whether the message should be propagated. The propagate parameter is true on the initial message which gets sent to the server and is false in the messages that get sent to each of the connecting players. When the message initially leaves the mobile device, the propagate parameter is set to true. The server then applies the change to the game file indicated in the message, sets the propagate parameter to false, and sends out a copy of the message to each of the devices connected to the game so they can visually apply that change to their game. Table 5.1 lists the current game state update messages in place in the server. Note that moveAgentToInventory is the only message that requires a response from the server before the change is applied locally.

Table 5.1: Possible Game State Updates

Method Name	Parameters
startGame	propagate
movePlayer	playerId, x, y, regionId, propagate
moveAgent	agentId,setOrChange, x, y, regionId, propagate
changeAgentVisibility	agentId, visibility, propagate
setChangeTrait	entityId, entityType, traitDefId, setOrChange, value, propagate
moveAgentToInventory	agentId, playerId, giverId, propagate
dropAgent	agentId, playerId, propagate
addPlayer	player
disconnectPlayer	playerId
setRole	playerId, roleId, propagate
setScenario	scenarioId, propagate
setTabVisibility	tabId, visibility, playerId, propagate
unlockEntity	entityId, playerId, propagate

Local Changes

There are some messages that will only change the state of the world of a single player. For example, setTabVisibility only includes/excludes the tabs of the player whose id is specified. The unlockEntity message works in the same way–it should only unlock the entity for the player specified. In the future, there will be more messages that indicate that something should only be changed for a single player. Currently, there are aspects of the game (action/trait visibility) that are not persistent over multiple game sessions because they are not yet propagated to the server, where the only saved copy of the game file exists (so if the player chooses to exit the game, these settings will be lost when they start up again). This needs to be fixed in the next version of the multiplayer code.

Adding/Disconnecting Players

These messages are sent from the server when players are added or disconnected. In order for the map tab to have feedback about where players were, it was necessary to include these messages. Upon receipt of these messages, a mobile device can either add or remove the corresponding icon of the player indicated.

Dealing with Conflicting Updates

There are some game state updates that could cause major conflicts with each other. Most updates are fine as long as they are applied in order–the game state itself may not make sense in the context of the game, but each player will still see the same thing. For example, the health of a particular agent could be taken below zero by a delayed change trait message. Some could possibly be more devastating. For example, consider the game that has an agent with a health trait. There exists an action that decreases the health of the agent by 5 points and an action that sets the health to full again. Because the system automatically applies these changes on the initiating player's game first to move along the game, the results of these actions could cause the players to eventually see different values for the same trait.

For example, say the starting health for the agent is 25 and is currently 15 due to by actions taken by the players. Now, Player 1 has disconnected from the server but nonetheless triggers the action that decreases the health by 5 points. Therefore, Player 1 will see the health as 10. While Player 1 is disconnected, Player 2 triggers the action that sets the health to full again. Player 2 (and everyone else connected) will see the health back at 25. When Player 1 reconnects to the server:

1. Player 1's device will apply Player 2's change of resetting the health to full (so Player 1 will see the health as 25) and his/her change will finally be propagated to the server.

2. The server and subsequently the rest of the players will apply Player 1's change to decrease the health by 5, so everyone else will see the health as 20 (note

that the system cannot apply the change trait to Player 1 again, as the change should not be applied more than once to be consistent with the block).

In this game, Player 1's game state has diverged irreparably from the rest of the players' game state. To remedy this possible situation, in cases of messages that have the "setOrChange" property, the server will always convert "change" messages to "set" messages. These set messages can always be applied on the client devices and as long as the set messages are applied in order, the players' game states will not diverge from each other.

5.2.4 Resend Updates Messages

In the all too common and unpreventable event that the mobile device disconnects from the server, there needs to be a system in place to resend any missing messages, and previously, there was no system to do this. The system now in place for resending uses selective acknowledgements for resending messages (mobile devices also receive positive acknowledgements that their messages have been sent, as they propagate to all other devices, including their own, from the server)[4]. Before any messages are resent, there needs to be a way to detect whether a message has been missed in the fist place. In the case of the mobile device, there is a variable that keeps track of the updateNum of the last message received from the server. If the message received is not one more than the last message received from the server, then the device knows that there are messages missing.

Assume that the last message received had updateNum 16, and the device has just received a message with updateNum 21. The mobile device starts a queue of messages received early and puts the message with updateNum 21 in it until the mobile device receives the missing messages. It determines what range of messages are missing (in this case 17-21) and sends a "requestMissingUpdates" message to the server. This message contains no requestNum, and includes the start (inclusive) and end (not inclusive) of the range of missing updates. If the mobile device receives additional messages out of order (for example, a message with updateNum 22), it assumes that the server has not yet gotten the message requesting missing messages so it resends the request and appends that new message onto the end of the queue.

In the meantime, both the client and the server have been keeping track of each message that they have sent. Therefore, when the server receives a request for missing updates, it references the object containing all of the sent updates and sends all of the missing updates in "resendMissingServerUpdates" message.

Finally, the mobile device will incorporate all of those messages. Afterwards, it will iterate through the queue of messages received early and following that, mobile device will be back up to date. However, given the earlier assumption that if the server sends a second message out of order, then it hasn't received the request for missing messages, it is possible to receive two sets of missing update messages. For this, the mobile code makes the assumption that if the updateNum of the received message is less than the last received (valid) update num, then it has already been applied and the message is ignored.

The server resolves missing mobile device messages in the same fashion. It keeps track of the last requestNum seen from each connected device, and in that way resolves missing messages for all connected devices as they occur.

5.2.5 Pings

Sometimes when mobile devices disconnect from the server, they do so silently. In this case, both the server and the disconnected device operate as if they are sending messages to each other, but in all actuality are not. If the a device ends up in this state, it needs to be resolved as quickly as possible so it can stay current with the server.

Therefore, if a mobile device hasn't gotten a message from the server in the last 15 seconds, it sends a "ping" message. The server should respond to the ping message with an "ack" message. Upon receipt of the ack message, the mobile device knows that it is still connected to the server.

If the device doesn't receive an ack from the server, it takes note and tries pinging again in another 15 seconds if it doesn't receive any messages from the server. within that time If after 3 times there is no response from the server, the device disconnects and retries the connection. From there, if the device is within range of wifi or a cell phone signal, it should be able to regain connection with the server and incorporate any missed messages.

5.2.6 "NoUpdate"

Sometimes the server progresses all the way up to the step where it increases the update number to send a message but doesn't actually have a message to send. One thing that would cause that would be failed pickups (when any player tries to pick up an item that is already picked up). In this case, the clients would not receive a message at that update number and proceed to request the message for that update number (and the message does not exist). Therefore, there's a placeholder message that is simply referred to as a "noUpdate" message, which the clients simply process only for the updateNum.

5.3 Garbage Collection

As the system scales, it becomes necessary to garbage collect any stale data on the server and mobile devices.

5.3.1 Old Messages

Because the mobile devices and server keep all sent messages in case they have to be resent, old messages must be garbage collected.

Mobile

Each message sent to the server from a mobile device is associated with a unique combination of player id and requestNum. This information can be utilized to remove old messages from the mobile message queue as the mobile device receives them. If the server is propagating this message, it should already have incorporated the data from this message into the game file. Therefore, once a mobile device receives a server message with its own associated player id, then it can remove the message with that request num from its message queue. Currently the message queue is being purged of all game state update messages as they are incorporated in the server, but not init messages (as they don't have associated responses for the mobile code to look for).

Server

Most importantly, because the server will eventually be handing many messages, the old messages from the server must be garbage collected. Currently, there is no mechanism in the code for doing this, as there is no acknowledgement indicating which players have received which message. For now, as the number of instances running concurrently on the server will be very small, this is not an issue. Later, this will need to be implemented, possibly by getting rid of messages that are very likely to have been received by all, and if a mobile device requests a message that has been garbage collected, just resend the game file to that device.

5.3.2 Old Users on the Server

In addition to garbage collecting old messages, it will also be necessary to garbage collect old users. After running many instances, there are bound to be many users referenced in objects on the server who are no longer connected to the server. Therefore, every few minutes, the server should clean out old users who haven't sent the server any messages in a predetermined amount of time. Doing this is crucial for the future when the server is scaled. Note that cleaning out users on the server is different than cleaning out players–the game file will still contain all references to players, so that even if the corresponding user has been cleaned out on the server, the reference to the player's state in the game will be preserved in the game file. The server will just automatically have to send the device a clean version of the game file.

Chapter 6

Playtest

6.1 Basic Play Tests

In order to ensure stability, I conducted basic play tests. They would test the most basic TaleBlazer game features, especially those features that would need to be mediated by the server.

6.1.1 "Picking up Pencils"

From the beginning, I decided that the most basic play test that the server would need to pass would consist of many agents in the world and multiple players would try to pick up those agents. This type of game was known as the "picking up pencils" game, named after the initial choice of object to be picked up. Eventually, pencils evolved into coins, but the basic functionality of the play test remained the same.

In these basic tests, I was looking for these different things:

- Neither the server nor the mobile devices would crash
- Players were able to pick up agents and see them in their inventories
- No two players would have the same agent in their inventories
- All agents were accounted for at the end of the game.
- Players could initiate pick up of agents while disconnected and agents would move to their inventories when they reconnected if nobody else had picked them

up in the interim.

- (Later) Players could drop agents and other players could pick them up.

Most of the tests I conducted myself indoors with 2-3 phones. In order to simulate possible outdoor conditions, I made myself a couple of buttons to aid in testing and added them to the Settings tab for testing. First, I made a "Simulate Tunnel" button that would disconnect the player from the server. The device would then retry the connection afterwards, giving me time to make sure I could satisfy the requirement that the players be able to initiate pickup while disconnected (and complete the pickup after reconnecting). In addition, so I could flood my server with messages (and test connectivity before pings were implemented), I created a "Send movePlayer" button which sends a movePlayer message with the player at the last known location. In this way, I could test to make sure that the server and mobile devices could handle resending many messages at once, as the most common failure mode that would cause things to crash was the resend functionality not working properly (Figure 6-1).

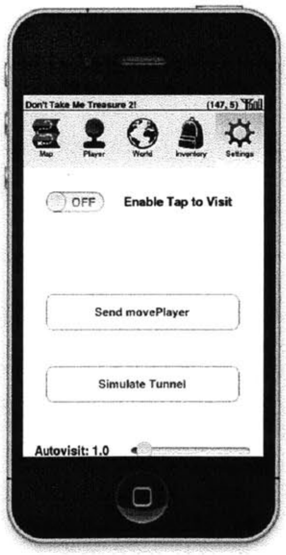


Figure 6-1: Interface for Testing Multiplayer

When I fixed all of the bugs in this controlled test, I gathered a group of people to test the same game outside (3-4 players, all on phones with data plans), testing the same basic things. In addition, we all gathered and try to pick up one agent at the same time and confirm that only one player got the agent. These tests would expose potential resending message bugs as well as offer up a chance for everyone to comment on the user interface of multiplayer in general. By the end of coding, all of my basic play tests were passing all of my requirements.

6.2 Advanced Play Tests

From there, I moved onto more complex play tests. In addition to the requirements listed above, the more complex game would also test:

- Making sure trait values were being propagated as expected for all players
- Making sure that agents were being included/excluded from the world for all players

On August 7, 2013, a group of high school freshmen/sophomores visited the lab and helped perform this complex test. Instead of the usual 3 or 4 devices, the test had 16-17 wifi-enabled devices connected to a single instance. In addition, because the wireless network outside wasn't conducive to maintaining enough connection to the server for smooth play, testing was done inside. Prior to the test, they were introduced to the concept of TaleBlazer, but otherwise had no previous interaction with the software. To test, I used the game "Don't Take Me Treasure as described in Chapter 3. The students were told to take note of the names of agents picked up to compare with their inventories later. The test was run twice, as the students were split into two testing groups.

6.2.1 Play Test Results

The first set of students had a mostly smooth game. One student reported that he had an item in his inventory that he didn't note picking up, but none of the other students also had that item, confirming that the item was not duplicated.

The second set of students ran into more problems. One of the devices disconnected and caused a loop with the server when asking for missing messages, a problem which didn't come up more than once in my earlier tests, but should be fixable with a stricter check before sending missing messages requests (it was requesting messages 5 to 5, which isn't a valid range). Additionally, there was another phone that was trying to create multiple connections with the server, causing the device to behave incorrectly, as multiple connections from one username cause the mobile device to be disconnected from the server. This bug will have to be fixed in future iterations.

Chapter 7

Future Work and Conclusion

Now that the multiplayer server is mostly stable, work can be continued to extend the multiplayer server functionality to support more interesting game mechanics.

7.1 Future Work

7.1.1 Teams

As explained in chapter 3, Multiple Competitive Team games rely on the concept of teams. To support teams, work must be done on the following:

- Looking into improving methods of communication for team members. In most games (especially in the proposed environmental disaster game in chapter 3), it is crucial to be able to relay information to teammates. There is currently work started on a messages tab, but more work would need to be done on that tab to support different types of chat (chat among everyone, chat among team members, player to player chat, etc.).

- Differentiating team members on the map/adding support so the player can see who is on his/her team

- Implementing ways for team members to give agents to one another

Once teams are implemented, they should be able to also have assigned traits and actions, as well as be tested for in the block set. This will allow for certain features such as team score and being able to perform different actions on agents based on what team the player is on.

7.1.2 Drones

There may be certain games where the game designer will want each player to be able to have a copy of a certain item. However, the game designer has no control over the number of players eventually in the game, so he/she doesn't know how many of that type of agent to add. Therefore, it is necessary to add a type of agent known as a "Drone". Once drones are in place on the map, they can keep creating an infinite number of copies of themselves. For example, this will come in handy when a game designer wants everyone in the game to have a key to proceed to the next part of the game.

7.1.3 Blockset

The addition of multiplayer in general opens up a realm of different possibilities for how the blocks should work. For example, consider the inInventory? block, which determines whether or not an agent is in the player's inventory. In the traditional TaleBlazer single player architecture, this block was pretty straight forward. However, with the addition of multiplayer elements, the block needs more parameters to answer the following questions:

- Is the agent in my inventory?
- Is the agent in everyone's inventory?
- Is the agent in everyone on my team's inventory?
- Is the agent in someone's inventory?
- Is the agent in someone on my team's inventory?

All blocks referring to players will need to be augmented to take additional arguments of this type when multiplayer games are being created. In addition, it becomes necessary to iterate over all players on certain events. For example, game designers might want to add points each player on a team following their completion of something and need to be able to iterate over all players to set this trait.

Finally, there could also be multiplayer-only blocks added to the block set. For example, there exists a "when game starts" block, but there should also be a different "when player enters instance" block, as not all players are going to be guaranteed to be in the instance when the game starts.

7.1.4 Player Interaction

For multiplayer games, it would also be interesting for players to be able to bump each other and affect each other as they effect agents. For example, in the epidemic game suggested in Chapter 3, it would be interesting for players to be able to directly infect each other and have to inoculate each other to the effects of the disease.

7.1.5 Helpful Future UI Elements

The current system still needs additional feedback mechanisms to ensure that the players can follow the events of multiplayer games.

Integrating Broadcast Messages

In many TaleBlazer games, the act of bumping one agent (or performing an action on that agent) often causes other agents to appear/disappear. In a multiplayer game, only the player directly responsible for that act will have the proper context for why the agents on the map changed. Therefore, in order to provide context for all players, it is necessary to be able to broadcast messages to all players and have them be displayed like the agent feedback previously shown. Game designers will be responsible for adding these messages to the game using the "Broadcast Message" block.

Notifications for Agents Appearing/Disappearing

There may be some games where it doesn't make sense for messages to be broadcast (for example, when many possible agents appear at the same time). Therefore, there needs to be some general visualization that notifies the players of agent changes. To alert players to new agents in the world, they should be highlighted in some way for a short period of time. Likewise agents that have been picked up or otherwise excluded from the world should slowly fade away as opposed to just disappearing.

Visualizing Server Connection Status

Finally, similar to the feedback for GPS connection status, there should also be visual feedback for the server connection status. If the device is receiving messages from the server normally, the map tab will show a green icon. If the device hasn't received a message from the server within a predetermined amount of time, the map tab will show a yellow warning icon. If there is no server connection whatsoever, the icon will be red.

7.2 Conclusion

The TaleBlazer game platform allows users to both play and design their own augmented reality location-based games. Following the work done for this thesis, TaleBlazer has a stable multiplayer server that is ready to be extended to various multiplayer-specific features. The server can run instances with many players in these instances, and each player in a particular instance will eventually see the same game state. Soon, users should soon be able to create and play their own multiplayer games, allowing for more possible TaleBlazer games than ever before.

Appendix A

Elements of a TaleBlazer Game

A.1 Game Features

Game designers have many elements that they can choose to mix into a TaleBlazer game. Currently, TaleBlazer supports all of these features:

World

The "world" is the name for the universe that the game takes place in. Game designers can further define the world by adding traits for the players to see/possibly change or actions for the players to perform.

Map

Every TaleBlazer game starts out with a map based on the location where the game is supposed to take place. There are currently 2 types of maps: a normal Google map, known as a dynamic map (can only be used on devices with a data plan), and a custom map where the designer can specify the image in the background.

Maps can consist of different regions. Regions can be used to express different locations/times. For example, two regions on top of one another could define agents in that location at different times, and switching regions is one way to allow users to move between times. In addition, if a region is predominantly indoors, it won't be able to get a GPS signal. In this case, the game designer can specify this region as being and "Indoor Region", meaning that all of the agents can be visited by simply tapping on them.

Tabs

The game can consist of any number of tabs, and game designers can choose which tabs to include in the game. The Game Tab and the Map Tab are included by default, but designers can also include tabs such as the World Tab, which shows traits and actions that players can perform on the world, the Player Tab, which shows information specific to that player's role, and the Inventory Tab, which shows all the agents that the player has picked up. More tabs can be found in the game editor.

Roles

Within a game, players can choose different roles, and game designers can control what players of specific roles can see and and what actions specific roles can take. For example, one player could be a locksmith who can open treasure chests, and another player could be an appraiser who can see how much that treasure is worth.

Agents

Agents are anything that the players can interact with in the game–from people they can speak with to artifacts they can pick up and use to help with their quest. Agents can be visible or hidden (to be revealed by a script later), as well as password protected. They can have associated actions and traits.

Actions

Actions are options that players can choose to take in the game, and can be defined on Agents, Roles, and the World itself. Actions can bring up text, videos, or run scripts defined by blocks. For example, opening a treasure chest could cause treasure to spill out and appear on the map–a sequence that could be defined by a series of blocks which include the treasure into the world when a player hits that action.

Traits

Traits are properties of an Agent, a Role, or the World, which can be assigned numerical or string values. These values can be set or changed via blocks, and can also either be visible or hidden. For example, a player's score might be a visible trait assigned to the Role, while a hit counter for an enemy might be a hidden trait (the game designer "kill off" an agent when its hit counter reaches a certain amount).

Scenarios

Scenarios were created mostly to accommodate large groups playing the same game. If the game isn't entirely sequential, multiple scenarios allow small groups of players to start at different points of the game, an experience which is superior to having a large mob of people moving along the same game trek. More generically, a scenario can be used to create different versions of a similar game, acting as a global variable that dictates how the game progresses, with all options defined by the designer.

Blocks

In addition to basic operators and if/then blocks, TaleBlazer also incorporates the following game-specific blocks for use, detailed in Table A.1 [7]:

A.2 Glossary of Additional TaleBlazer Terms

game code – unique, searchable code assigned to each game
game file – text file consisting of a JSON object which includes all information necessary to run the game
instance – consists of a game and all of the players currently playing that game in the shared game world
instance code – unique, searchable code assigned to each instance

Table A.1: Types of TaleBlazer Blocks

Control Blocks	
Block Description	Parameters
When game starts, run script	none
When player sees world tab, run script	none
Action script (can be renamed)	none
Game Blocks	
Block Description	Parameters
Include/exclude agent	agent
Test to see if agent in world	agent
Test role of player	player type, role
Test scenario of game	scenario
Looks Blocks	
Block Description	Parameters
Show/hide action	action, entity (world, role, agent)
Show/hide trait	trait, entity (world, role, agent)
Say something	thing to be said (plain or rich text)
Switch tab	tab
Include/exclude tab	tab
Movement Blocks	
Block Description	Parameters
Move player/agent region	entity (player or agent), region
Test if player/agent in region	entity (player or agent), region
Move agent to x,y	agent, x, y
Set/change agent x	x or Δ x
Set/change agent y	x or Δ x
Get x of player/agent	entity (player or agent)
Get y of player/agent	entity (player or agent)
Is agent in inventory	agent
Trait Blocks	
Block Description	Parameters
Set/change trait value	agent, trait, value or Δ value
Get trait value	agent, trait

repository server – external server that contains all assets necessary to run a TaleBlazer game, including game files, pictures, and videos

Bibliography

[1] "Intro to Node.JS for .NET Developers". http://www.aaronstannard.com/post/2011/12/14/Intro-to-NodeJS-for-NET-Developers.aspx. Accessed: 2013-08.

[2] "MITAR Games". http://education.mit.edu/projects/mitar-games. Accessed: 2013-08.

[3] "Outbreak@MIT". http://education.mit.edu/ar/oatmit.html. Accessed: 2013-08.

[4] "Retransmission (data networks)". http://en.wikipedia.org/wiki/Retransmission_(data_networks). Accessed: 2013-08.

[5] "StarLogo on the Web". http://education.mit.edu/starlogo/. Accessed: 2013-08.

[6] "StarLogo TNG". http://education.mit.edu/projects/starlogo-tng. Accessed: 2013-08.

[7] "TaleBlazer". http://taleblazer.org. Accessed: 2013-08.

[8] Eric Klopfer. *Augmented Learning.* The MIT Press, 2008.