Improving the TaleBlazer Multiplayer Platform

by

Arjun V. Narayanan

S.B., Massachusetts Institute of Technology (2015)

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

May 2016 [June 2016]

© Arjun V. Narayanan MMXVI. All rights reserved.

The author hereby grants to MIT permission to reproduce and to distribute publicly paper and electronic copies of this thesis document in whole and in part in any medium now known or hereafter created.

Signature redacted

Department of Electrical Engineering and Computer Science
May 16, 2016

Signature redacted

Certified by ...

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

Signature redacted

Accepted by ...

Professor Christopher J. Terman
Chairman, Masters of Engineering Thesis Committee
Improving the TaleBlazer Multiplayer Platform
by
Arjun V. Narayanan

Submitted to the Department of Electrical Engineering and Computer Science
on May 16, 2016, in partial fulfillment of the
requirements for the degree of
Master of Engineering in Computer Science and Engineering

Abstract

TaleBlazer is a platform for building and playing augmented reality, location-based, mobile games. Users design games in a web-based editor and then play the games on their mobile device with the TaleBlazer mobile app. The classical version of TaleBlazer is single-player, where each player plays in his or her own individual game world. A newer version of TaleBlazer is multiplayer, which involves multiple players sharing the same game world. This thesis describes new features added to the TaleBlazer multiplayer platform. The new features vary in size and scope. Some are large-scale new game mechanics, while others are on a smaller-scale. Overall, the added functionality allows game designers to design more complex games and gives players a better gameplay experience.

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

I’d like to thank Professor Eric Klopfer, Lisa Stump, and Judy Perry for giving me an opportunity to work on TaleBlazer Multiplayer for my MEng. I loved my time with the TaleBlazer team and I learned so much.

I’d like to thank Professor Klopfer for all of his help and feedback throughout my MEng.

I’d like to thank Lisa Stump for all of her technical help and guidance throughout this project. I’d like to thank Judy Perry for all of her help with design decisions and project planning. Without Lisa and Judy, I would not have been able to do this project. Their help was invaluable.

I’d also like to thank fellow members of the TaleBlazer team, including David Powell, Linda Wang, Zachary Neely, Stephanie McHugh, Zachary Sather, Sarah Edris, Manali Naik, and Bobby Fortanely. I enjoyed working with all of them and they were very helpful throughout the project.

I’d like to thank my predecessors on TaleBlazer Multiplayer, Sarah Lehmann and Tanya Liu for their work. It is because of their work that I have the opportunity to add new functionality to the multiplayer platform.

I’d finally like to thank Danny Fain, a freelance game designer whom I collaborated with during this project. Your help in testing out my work was invaluable. Our discussions regarding new feature ideas for TaleBlazer multiplayer were fun, useful, and enlightening.

Lastly, I’d like to thank my family for all of their support, love, and help in life.
# Table of Contents

1 Introduction to TaleBlazer ........................................................................................................ 10  
   1.1 What is TaleBlazer? .................................................................................................................. 10  
   1.2 TaleBlazer Gameplay ............................................................................................................... 10  
   1.3 The TaleBlazer Editor ............................................................................................................. 13  
       1.3.1 Blocks-based Programming Language ........................................................................ 14  
   1.4 Single-player vs. Multiplayer ................................................................................................. 17  

2 Introduction to TaleBlazer Multiplayer ..................................................................................... 18  
   2.1 Initial Motivation for Multiplayer ......................................................................................... 18  
   2.2 Goals for Multiplayer ............................................................................................................ 19  
   2.3 Types of Multiplayer Games ................................................................................................. 20  

3 Previous Work on TaleBlazer Multiplayer ................................................................................ 22  
   3.1 Important Vocabulary – Instance and Instance Code ......................................................... 22  
   3.2 Technical Architecture .......................................................................................................... 24  
   3.3 User Identification Inside of an Instance ............................................................................. 25  
   3.4 The Intermittent Connectivity Problem ............................................................................... 25  
   3.5 Previously Built Multiplayer Game Mechanics ................................................................. 26  
       3.5.1 The Give Action .............................................................................................................. 26  
       3.5.2 Teams ............................................................................................................................. 26  
       3.5.3 Player Icons on Map ..................................................................................................... 27  
       3.5.4 New Blocks for Multiplayer ........................................................................................... 28  

4 Overview of Problems to be Solved ......................................................................................... 30  
   4.1 Reviving TaleBlazer Multiplayer ........................................................................................... 30  
   4.2 Usability Issues, Technical Problems, and Underspecified Behavior .................................. 30  
   4.3 Player Communication .......................................................................................................... 31  
   4.4 Give Action Not Proximity Constrained ............................................................................... 31  

5 Reviving TaleBlazer Multiplayer ............................................................................................... 32  
   5.1 Problem ................................................................................................................................. 32  
       5.1.1 Problems with Game Start Process ............................................................................... 33  
   5.2 Solution .................................................................................................................................. 34  
   5.3. Future Work ......................................................................................................................... 34  

6 Small-scale Features .................................................................................................................. 36  
   6.1 Resuming a Multiplayer Game .............................................................................................. 36  
       6.1.1 Problem ........................................................................................................................... 36  
       6.1.2 Solution .......................................................................................................................... 37  
   6.2 Rejoining an Instance as a New Player ............................................................................... 38  
       6.2.1 Problem ........................................................................................................................... 38  
       6.2.2 Solution .......................................................................................................................... 39  
       6.2.3 Future Work .................................................................................................................... 39  
   6.3 How Long Should Multiplayer Instances Last? ................................................................... 40  
       6.3.1 Problem ........................................................................................................................... 40  
       6.3.2 Solution .......................................................................................................................... 40  
       6.3.3 Future Work .................................................................................................................... 41
List of Figures

Figure 1 – The dashboard for a “Gold Coin” agent .......................................................... 11
Figure 2 – Tabs in a TaleBlazer game ............................................................................ 13
Figure 3 – Screenshot of the TaleBlazer Editor ............................................................... 14
Figure 4 – Example of a custom script action ................................................................. 16
Figure 5 – Example usage of say blocks ........................................................................ 17
Figure 6 – Screenshot of the “game code screen” ........................................................... 23
Figure 7 – Example of a team dashboard ......................................................................... 27
Figure 8 – Player icons in a multiplayer game ............................................................... 28
Figure 9 – Example usage of the “player on team” and “for each” blocks .................... 29
Figure 10 – The “game page” for a multiplayer game .................................................... 38
Figure 11 – The new “multiplayer game introduction” .................................................... 43
Figure 12 – Making the instance code always visible .................................................... 44
Figure 13 – UI during agent pickup in a multiplayer game ............................................ 46
Figure 14 – The looks blocks in TaleBlazer ................................................................. 47
Figure 15 – Example usage of the looks blocks in a multiplayer game ......................... 50
Figure 16 – The “email view” for chat .......................................................................... 56
Figure 17 – The “Facebook view” for chat .................................................................... 57
Figure 18 – The final UI for TaleBlazer chat ................................................................. 59
Figure 19 – Unread chat counts ..................................................................................... 61
Figure 20 – Algorithm to compute the distance between a player and agent in terms of meters 65
Figure 21 – Settings for the give action in the TaleBlazer editor .................................. 66
Figure 22 – Proximity-constrained give action on the TaleBlazer app ......................... 68
Chapter 1

Introduction to TaleBlazer

TaleBlazer is a platform that allows users to build and play location-based augmented reality mobile games. The focus of this thesis is on multiplayer functionality for the TaleBlazer system. In order to understand multiplayer functionality, it is important to have some context and background knowledge on TaleBlazer. In this chapter, I first provide a general description of what TaleBlazer is. Then, I discuss the two fundamental aspects of TaleBlazer - gameplay and game design. Finally, I explain the difference between single-player and multiplayer TaleBlazer games.

1.1 What is TaleBlazer?

TaleBlazer is an augmented reality (AR) location-based game platform that allows users to make and play their own mobile games [1]. The games are played on mobile devices, but they take place in the real world with the TaleBlazer app tracking user movements using the device GPS. Users walk around in the real world with their mobile device. When they go to certain physical locations, they “bump” in-game characters and objects. They interact with these characters and objects on their mobile device.

TaleBlazer consists of two main parts – the TaleBlazer Editor and the TaleBlazer mobile app. The editor is a web app and the mobile app runs on Android and iOS devices. Game designers make their games in the editor and then users can play the games on the app. The following two sections discuss gameplay and game design respectively. These sections provide a high level overview of important concepts. More specific aspects of the gameplay and editor will be explained as necessary in this thesis.

1.2 TaleBlazer Gameplay

As mentioned before, as players walk around the real world, they “bump” into virtual items and characters. In TaleBlazer, we call these items and characters agents. TaleBlazer is a location-based game, so the agents are typically linked in some way to the real-world location
they are at. For example, there might be virtual soldier agents placed on the real Lexington battle green.

When a player bumps an agent, he or she triggers an interaction with that agent. A screen for that agent opens up on the player’s mobile device. This screen, which we call the agent dashboard, shows the agent’s name, image, and description. The game designer can also configure buttons known as actions that allow the player to interact with the agent. Game designers can specify built-in actions such as pick up and drop, or they can create their own custom actions. Custom actions will be discussed more in the next section on the editor. See Figure 1 for an example of an agent dashboard.

![Figure 1 - The dashboard for a “Gold Coin” agent. This agent has the built-in action “Pick Up”.

Agents are not the only things that can have actions. More generally, TaleBlazer games have the following key objects: agents, players, teams, and the world (the world represents the game world itself). All of these can have actions. For example, the world might have an action called “Reveal” which randomly reveals a previously hidden agent in the game world.
Agents, players, teams, and the world can also have traits in TaleBlazer. Traits are attributes or characteristics. For example, a player might have a trait called “health”. An agent representing a treasure chest might have a trait called “Remaining Value.” The world might have a trait called “Amount of Minerals Left.” Traits and actions can be closely related. For example, the player could have an action called “Inject Health” which would increase the “Health” trait of the player. The next section will explain how the game designer can encode such logic.

The game world can be divided into a number of regions, each of which maps to a specific real world location and contains one or more agents. The game designer can encode game logic to move a player between regions. Game designers might choose to use multiple regions in a game for various reasons - for example, to represent different chapters of a game at the same physical location, or to represent different physical locations.

Another important aspect of TaleBlazer gameplay is roles. Roles are essentially just different classes of players. Each role can have special privileges and abilities. For example, consider a game where all players are trying to fight a disease outbreak. Different roles might include: medic, scientist, and police officer. The game designer can specify a set of different roles for players to choose from when they start playing a game.

Finally, one might ask how a player views all this information about traits, actions, and regions while playing a game. At the top of the game window, TaleBlazer has a series of tabs that represent key functionality in the game. The most important of the tabs is the Map tab, which shows the geographic area where the game is taking place. The map shows the location of agents and the player. Other tabs include the Inventory tab (which shows all the agents a player has picked up), the Player tab (which shows the player’s role and all traits and actions for the player), and the World tab (which shows traits and actions for the game world). The game designer can pick which tabs are available in a game.
Figure 2 – Tabs in a TaleBlazer game. The game designer has configured this game to have four tabs. In the image on the left, the Map tab is selected. The colored shapes on the map indicate the locations of agents. In the image on the right, the Player tab is selected. It shows the role of the player (“Medic”) and the two traits for the player (“Health” and “Money”).

1.3 The TaleBlazer Editor

The TaleBlazer Editor is a web app that game designers use to build games. In the editor, the game designer can specify the real-world location for the game (we call this the map for the game). The designer can optionally divide the map into regions and then place the agents on the map. Agents can be hidden from the players in order to encourage exploration and searching. As players wander through the world, they will unexpectedly bump into agents.

For each agent, the designer can specify a list of actions for interacting with that agent. As alluded to before, certain built-in actions are automatically provided. These include the ability to pick-up the agent into the player’s inventory and drop the agent back into the game world. The designer can hide these built-in actions if he/she does not want them. The designer can also define custom actions. He/she must specify what happens when the player performs a custom action.
action. This leads to a larger question of how game designers specify the game logic. They do so using TaleBlazer’s **blocks-based programming language**.

Figure 3 - Screenshot of the TaleBlazer Editor. On this specific screen, the game designer can create/delete regions and layout the agents on the map. The agents are the colored shapes. The designer places the agents by dragging the shapes around.

### 1.3.1 Blocks-based Programming Language

To specify the game logic, designers use a blocks-based programming language. There are various blocks available including `if/else` statements, blocks to set the value of a specific trait and blocks to show some text on the screen.

This is how a game designer specifies the game logic for a custom action. The designer associates each custom action with a configurable stack of blocks, which we call a script. For example, suppose a game has a “Treasure Chest” agent. The designer might define a custom
“Open Chest” action. Inside the action’s script, there could be blocks to tell the user what they have found inside the chest, blocks to increment a “money” trait of the player, or blocks to include newfound items in the world. See Figure 4 for one possible script.

The game designer can also use blocks to handle events in the game. In other words, the designer can create event handlers. For example, TaleBlazer provides a “when player bumps agent” block. This block is commonly used to show the player some introductory information about an agent when the player bumps the agent. Specifically, the designer uses a “say” block inside of the “when player bumps agent” block to show some text on the screen. See Figure 5 for an example of this. Another example is the “when player picks up agent” block. The designer can use this block to execute game logic whenever the player picks up an agent. For example, suppose we have a game where a player’s job is to collect all the coins in the world. The world could have a trait called “Number of Coins Left”. The designer could specify logic such as “when player picks up coin, decrement trait ‘Number of Coins Left’ of the world”.

Overall, the blocks-based programming language gives game designers precise control over the game mechanics.
Figure 4 - Example of a custom script action. For the “Treasure Chest” agent shown on the left, the game designer creates a custom action called “Open Chest” This action’s behavior is defined by a script of three blocks. Each block includes a new item in the game world.
Figure 5 - Example usage of say blocks. The image on the left shows the game logic in the editor. The image on the right shows how the text in a say block is shown in game.

1.4 Single-player vs. Multiplayer

The classic version of TaleBlazer is single player, meaning that each player is unconnected from other players. Every player can download and play a TaleBlazer game on his/her mobile device. However, each player is in his/her own separate game world. Every player sees his/her own copy of the agents in the game. One player does not see the results of another player’s actions in the game. In summary, each player can experience the game without affecting anybody else.

The focus of this thesis is on multiplayer, a newer version of TaleBlazer that is not yet in production. In this version, multiple players share the same game world. The players are all in the same world and see the same agents. Each player sees the effects of other player’s actions. The next chapter begins describing the history of multiplayer. It discusses the motivation for building multiplayer games and what exactly a player’s experience inside a multiplayer game should be.
Chapter 2

Introduction to TaleBlazer Multiplayer

A significant amount of thought and implementation went into TaleBlazer multiplayer before I began my work. This chapter and the next describe previous work on the system. This chapter explains the initial motivation for multiplayer and previous design decisions regarding the goals of the system. The next chapter describes previous implementation work. At the beginning of my MEng, I reviewed all of this previous work and understood the current state of the system. Then, I defined new challenges and problems for my own work.

The initial motivation for TaleBlazer multiplayer was that multiplayer games could provide a richer, more complex gameplay experience. Players can interact not only with virtual characters, but also with each other. In the context of education, multiplayer games can have a number of benefits. For example, players can collaborate to solve problems and can learn teamwork and leadership.

Previous developers decided that the goal of a multiplayer game is to provide a shared game world. Players all exist in the same world and each player can see the effects of other player’s actions. Some thought also went into the types of multiplayer games TaleBlazer needs to support. Three major categories of multiplayer games were identified – player vs. world, single cooperative team, and multiple cooperative teams.

2.1 Initial Motivation for Multiplayer

In general, multiplayer games can provide a richer experience than single-player games. In single-player games, players can only interact with virtual characters and items. In multiplayer games, players interact with real people. The game experience can be more complex and unpredictable.

In the context of education, multiplayer adds a social dimension that can enhance the experience. First of all, players could simply be more engaged in the educational content if they are playing with their friends. Second, they could be more motivated to learn if they are competing against others. Finally, in team games, users can cooperate with each other to achieve
more than they would individually. They can work together to co-construct knowledge. As they collaborate, students can learn both reliance on and accountability towards others.

Once we have a stable multiplayer platform, we can actually test these hypotheses. We can formalize the ideas in the previous paragraphs as research questions. Possible research questions include:

1. How does player engagement vary between single-player and multiplayer games?
2. Do users learn best when playing a multiplayer game with their close friends, acquaintances, or with strangers?
3. When given the opportunity, do users collaborate with other players in a multiplayer game? What types of game designs encourage users to collaborate the most?

The system is not yet ready to explore such research questions though. We are still in the stage of building the platform. Prior work and my own work focused on actually developing functionality. Using the platform for this type of research is outside the scope of this thesis.

In summary, building TaleBlazer multiplayer functionality opens up avenues for more complex and rich gameplay experiences. It also allows us to, in the future, explore many interesting research questions related to education.

2.2 Goals for Multiplayer

In the past, many design decisions were made about multiplayer. Prior members of the TaleBlazer team decided that the goal of multiplayer is to provide a shared game world in which players can interact with each other [2]. Every player should see the same game world. Every player should see the results of actions that other players take. For example, if one player’s action causes a trait of the game world to change, other players should see the updated trait value. The game world should also be consistent. For example, no two users should be able to pick up the same agent.

These goals are achieved by having each player connected to the Internet. Each mobile client communicates with a special server dedicated to multiplayer. The server maintains the shared game world and routes messages between the different players in the game. This will be discussed in more detail in Section 3.2, when I discuss the technical architecture of the system.
Multiplayer games need to be played in the presence of intermittent connectivity. Mobile devices can frequently lose and regain Internet connection. This should not break the multiplayer experience. This gives rise to the important idea of eventual consistency [2]. Players should eventually all see the same game world. If a player is disconnected, he/she will temporarily be unable to see the results of actions that other players are performing. However, when he/she reconnects, he/she will see the effects of those actions and eventually see the same game world as the other players.

Another important design decision made in the past concerns responsiveness [2]. As much as possible, players should not experience delays while the server is synchronizing their actions. The game UI should reflect the players’ actions as quickly as possible. Some delays will be inevitable, because the game world has to be consistent. Certain actions have to be mediated by the server in order to ensure consistency. The most important of these actions is the “pick up” action. Two users cannot both pick up the same object. If they both try to at the same time, the server must mediate the conflict.

2.3 Types of Multiplayer Games

Prior work also involved carefully brainstorming the different types of multiplayer games [2]. It was determined that there are three major types of multiplayer games. These categories were used to guide the system’s design and implementation. The categories are as follows:

1. Player vs. the world
2. Single cooperative team
3. Multiple cooperative teams

In a “player vs. the world” game, every player operates independently. An example of this type of game is as follows. Suppose there are many gold coins scattered across a region. Every player’s goal is to acquire as many gold coins as possible. Every player is trying to maximize his/her score and there is no cooperation. This is, in many ways, the simplest type of multiplayer game because of the lack of cooperation.
A “single cooperative team” game involves all players working together to achieve a common goal. An example of this game is where players are all working together to fight a disease outbreak.

A “multiple cooperative teams” game involves multiple teams playing against each other. An example of this game is “Capture the Flag,” where there are two teams. Each team is trying to acquire the other team’s flag and bring it back to their own base.
Chapter 3

Previous Work on TaleBlazer Multiplayer

The previous chapter discussed high-level design decisions made about TaleBlazer multiplayer in the past. This chapter discusses prior implementation work. At the beginning of my MEng, I understood all of this prior work and then posed challenges and problems for my own work.

Prior work on TaleBlazer multiplayer involved developing the technical architecture of the system, most notably building a multiplayer server to maintain the shared game world [2]. A related task was to find a way to uniquely identify players inside of a multiplayer game [3]. The multiplayer server needs some way to track all the players.

Previous work was also done to handle the intermittent connectivity problem [2]. Mobile devices can frequently lose and regain their network connection, and this should not break the multiplayer game experience.

Finally, some work was done on multiplayer game mechanics [3]. These game mechanics include:

- A way for players to see other players on the map
- The ability to give agents from one player to another
- The framework for teams
- Two multiplayer specific blocks in the TaleBlazer Editor – the “is player on team” block and the “for each block.”

This chapter first introduces some important vocabulary and then describes this previous work in more detail.

3.1 Important Vocabulary – Instance and Instance Code

Before looking at prior implementation work, it is important to know some vocabulary. In TaleBlazer multiplayer, players play in a shared world called an instance [2]. An instance is one instantiation of a specific TaleBlazer game. There can and will be multiple instances of each game. Each instance has a unique identifier called an instance code. When a player starts a new
instance, he/she will receive the instance code. Other players use the code to join the instance. They type the code into a special screen on the app called the “game code screen.” See Figure 6 for a screenshot of this screen.

Figure 6 – Screenshot of the “game code screen.” To join an existing multiplayer instance, users enter the instance code on this screen.
3.2 Technical Architecture

For single-player games, the system has the following architecture. There is the TaleBlazer app that runs on both Android and iOS. The app communicates with the main TaleBlazer server, which is a CakePHP server backed up by a MySQL database. The TaleBlazer server's job is to store and provide games that designers have created in the editor. There is also a separate Node.js server called the analytics server. This program tracks key events in games and collects data for users to analyze.

For single-player games, the mobile app communicates with the TaleBlazer server to find and download games. Once the game has been downloaded, no further communication with the TaleBlazer server occurs. Communication only occurs with the analytics server, which records gameplay events. Besides analytics, the user's gameplay experience is entirely local. Neither the TaleBlazer server nor the analytics server needs to mediate or approve the player's gameplay decisions. The user maintains his/her game world locally.

For multiplayer, there is an additional Node.js program called the multiplayer server [2]. This server maintains the shared game world and communicates with all the clients. Much of the prior work on TaleBlazer multiplayer involved building this server. Multiplayer functionality is implemented in both the Node.js multiplayer server and on the TaleBlazer app (server-side and client-side functionality). For the rest of this thesis, the term “server” refers to the Node.js multiplayer server, not the main TaleBlazer server or the analytics server. Actually, analytics is largely a separate topic unrelated to most of the work in this thesis.

The multiplayer server stores the master version of the game state. Each mobile device communicates with the server through a carefully designed protocol [2]. The most important messages in the protocol are update messages. These messages describe some change to the game state (e.g. the score of a certain player increased by 1 or a certain item was picked up by a certain player). The server's job is to listen for these update messages from the TaleBlazer mobile clients in the game. After the server receives a gameplay update, it applies the update to the master copy of the game state and then routes the update to all other players. Note that there is no direct client-to-client communication. All updates are mediated through the multiplayer server. Each client only communicates directly with the server.

As mentioned in Section 2.2, an important goal of TaleBlazer multiplayer is responsiveness. In most cases, a player does not have to wait for a server response to see the
results of his/her gameplay action [2]. The results of the action will immediately be apparent in
the local game world and the propagation to the server will happen afterward. The only case
where the server needs to mediate/approve the gameplay action is the “pick up” action. This is
because the same agent is not allowed to exist in the inventory of two different players at the
same time. If two players try to pick up an agent at the same time, the server handles the
conflict. One user will receive a message saying the pickup occurred successfully while the other
user will receive a message saying the pickup failed.

3.3 User Identification Inside of an Instance

On both the server and the app, players inside of an instance are identified by their
mobile device id [3]. This id is guaranteed by the mobile development platform to be unique
among devices. On Android, each device is assigned a unique id that can only change upon
factory reset of the device. On iOS, this device id is specific to the installation of each app.

However, this device id is not readable and user friendly. Thus, players must also type an
in-game username whenever they join a game. This username identifies the player within the
game. Players do not ever know that the device id is being used behind the scenes.

3.4 The Intermittent Connectivity Problem

Prior work also focused on solving the problem of intermittent connectivity [2]. This
problem was discussed in Section 2.2. When a client temporarily loses connectivity with the
multiplayer server, two problems occur. First of all, any messages sent by the server to the client
will be missed. Second, any messages sent by the client to server will not be received.

Both of these problems are solved using sequence numbers. Each update sent by the
server to clients is assigned an “update number.” Every mobile client stores the last update
number that it received from the server. When a client gets disconnected, it will stop receiving
new update numbers. When a client reconnects, it sends the server the last update number it
received. The server supplies the missing updates.

The second problem is handled similarly. Every message the client sends to the server is
assigned what is called a “request number”. The server stores the last request number sent by
every client. If the server detects that it missed some requests from a client (e.g. it received a new
request with number 15, but the last request it received had number 10), it will ask the client for the missing requests.

3.5 Previously Built Multiplayer Game Mechanics

A lot of the prior work on TaleBlazer multiplayer focused on building the multiplayer server and designing the server/client protocol. However, some work was also done on multiplayer game mechanics [3].

3.5.1 The Give Action

First of all, previous work was done to allow players to give agents to each other. This functionality is handled by the built-in give action (built-in actions are explained in sections 1.2 and 1.3). When a player taps on an object in his/her inventory, he or she is shown a “Give” action if the action is enabled. When the user presses the button, he/she will be shown a list of players to select from. Some work was also done to handle loss of Internet connection during the give action. The game world should not become inconsistent. Either the giving player must retain the item or the receiving player must receive the item. The item should not disappear and both players should not have the item.

Note that just like any other TaleBlazer action (such as “pick up” or “drop” or even custom actions), the give action can be hidden for specific agents or for specific circumstances.

3.5.2 Teams

The framework for teams inside a multiplayer game was also built previously. The game designer can specify the different teams in the TaleBlazer Editor. When players join a game with teams, they will have to select one of the teams. Just like players, agents, and the world, teams can have traits. For example, the game designer can use a team trait to implement a team-wide score. The traits for a team are shown on the team dashboard. Players can access the dashboard for their team using the Team tab at the top of the game window. The Team tab is treated similarly to the other tabs discussed in Section 1.2. See Figure 7 for an example of a team dashboard.
Figure 7 - Example of a team dashboard. Team MIT has a trait called “Score.” Players on Team MIT can see the dashboard by tapping on the “Team” tab at the top of the game window.

3.5.3 Player Icons on Map

Work was previously done to enable players to see other players in the game. Every player in the game is shown on the map with a player icon. The icons move as the players move. Players can tap an icon to see that player’s in-game username. See Figure 8 for a screenshot of player icons in a game.
3.5.4 New Blocks for Multiplayer

Some work was also done on providing new blocks in the editor. For instance, a block was added to check if a player is on a specific team. This allows game designers to have different game logic for each team.

Another important block added was the "for each" block. This block allows the game designer to iterate through the players in the game. Many multiplayer games rely on such game logic. For example, consider a game where one team is fighting against the other. Suppose a player launches a volley of arrows toward the other team. We would need game logic to iterate...
through all players of that team and decrement their health. See Figure 9 for another example on using these new blocks.

Figure 9 - Example usage of the “player on team” and “for each” blocks.
Chapter 4

Overview of Problems to be Solved

The previous two chapters described past work on TaleBlazer multiplayer. This past work involved building the multiplayer server to maintain the shared game world and implementing some multiplayer game mechanics. Based on this past work and the state of the system, I identified problems to solve in my own work. The problem areas I identified are as follows:

1. No multiplayer functionality had been implemented in the year before I began my work. I needed to revive the existing functionality and fix any problems.
2. The system had some usability issues and problems with the technical architecture. Also, the behavior of some parts of TaleBlazer multiplayer was underspecified.
3. Players could not communicate inside of a multiplayer game.
4. The previously built give action was not restricted based on player location.

This chapter provides a high-level overview of these problem areas. Further detail is provided later in the thesis, in chapters devoted specifically to individual problem areas.

4.1 Reviving TaleBlazer Multiplayer

No work had been done on multiplayer in the year before I started my MEng. However, a lot of functionality was added to TaleBlazer. This new functionality had broken some of the previously implemented multiplayer functionality. I discuss these problems and how I fixed them in Chapter 5.

4.2 Usability Issues, Technical Problems, and Underspecified Behavior

When I began my work, the multiplayer system had some usability issues. For example, suppose a player accidentally exited out of the app while playing a multiplayer game. After
reopening the app, the player did not have a quick, easy way to get back into the game. As another example, players did not always know how to invite others to play in a multiplayer instance they had started. In another light, there were also certain technical problems. For example, certain user actions could cause instances to be created on the server, but have no players inside of them. Finally, some parts of the multiplayer platform had underspecified behavior. For example, it was unclear how certain blocks would operate in multiplayer games.

I built a series of small-scale features to address all of these problems. These features are discussed in Chapter 6. For each feature, I describe the problem being solved, the implemented solution, and some possible future work.

4.3 Player Communication

Players could need to communicate while playing a multiplayer game. For example, members of the same team might need to discuss their team’s strategy. If players are nearby each other, then they can just talk in-person. However, if they are far apart, they have no way of communicating. The previous version of TaleBlazer multiplayer offered no long-range communication mechanism. This motivates the need for some sort of in-game chat functionality. This is all discussed in detail in Chapter 6. I discuss the problem in more detail and then explain the new chat feature that I built.

4.4 Give Action Not Proximity Constrained

The previously implemented version of the give action was not constrained by player proximity. Players could give agents to each other regardless of how close they were on the map. This is not realistic. In the real-world, players can only exchange items if they are nearby each other. In order to build TaleBlazer games that accurately simulate real-world scenarios, we need to have a proximity-constrained version of the give action. The proximity-constrained give action is discussed in more detail in Chapter 8. I provide more detail about the problem itself and then explain the new functionality.
Chapter 5

Reviving TaleBlazer Multiplayer

No work was done on TaleBlazer multiplayer in the year before I began my work. Before I started implementing new features, I had to revive the previously implemented functionality. This chapter explains the problems that I encountered while doing this and how I fixed them. This chapter discusses specific details of the codebase and is intended more for developers. Readers who are more interested in higher-level aspects of the system and actual multiplayer game mechanics can skip this chapter.

5.1 Problem

In the year before I began, no work was done specifically on multiplayer. However, a significant amount of new functionality was added to TaleBlazer. Some of this functionality was specific to single-player games and some of it applied to both single-player and multiplayer games. All of this functionality was implemented in the TaleBlazer app. In other words, only the client-side code changed. The code for the Node.js multiplayer server did not change. The server is used to synchronize actions in a multiplayer game and maintain the shared game world. Only developers who are building functionality specific to multiplayer will change the server.

As I revived the previous multiplayer functionality, I encountered two types of problems. First, some of the new single-player specific functionality was being incorrectly applied to multiplayer games. When developers wrote the code, they did not introduce checks to see if the game was multiplayer. For example, consider the following problem. In multiplayer games, the game state is not persisted on the client side (on the mobile device). The game state is not written to a database on the device and is only kept in-memory while the user plays the game. This is because the multiplayer server maintains and persists the master copy of the game state. The server sends the client the game state whenever the client starts up a multiplayer game. The client only needs to maintain the state in memory as it plays the game. On the other hand, for single-player games, a copy of the game state is saved in a database the on mobile device. If the user quits and resumes a game, the device will load the game state from the database. During the
year before I began my work, some changes were made to when and how this game state is saved to this database. The developers who worked on this inadvertently made it so the game state was saved for multiplayer games as well.

The second type of problem I encountered was that the new functionality that applied to both single-player and multiplayer games was not actually implemented for multiplayer games. It was only implemented for single-player games. This problem arises because of how the code is structured. Throughout the codebase, there is code like “if (game is single-player), then do the following, else do the following.” When implementing new functionality, developers did not include the code in the multiplayer case. The code is structured this way to solve the first type of problem, where the single-player and multiplayer code paths are different. Also, in general, different developers have worked on single-player and multiplayer functionality and they have tried to keep their code disjoint through these if/else statements (for better or worse).

Both of these types of problems occurred very frequently in the code to start up a multiplayer TaleBlazer game. I spent most time fixing problems in this part of the code. I discuss the specific issues with the game start process more in the following subsection.

5.1.1 Problems with Game Start Process

The “game start” process is the sequence of steps taken from when the user presses a button to play a game to when the user actually enters the game world and sees the in-game UI (the tabs, the map, etc). The code for starting a game in TaleBlazer is complex. It is spread out across many source files and contains many methods. Some of the steps apply to both multiplayer and single-player games. For example, the in-game UI needs to be loaded in both types of games. Many steps are different. For example, for multiplayer games, the app needs to communicate with the multiplayer server to start an instance or join an existing instance. In single-player games, the app downloads the game from the TaleBlazer server (not the multiplayer server).

The game start process changed significantly in the year before I began my work. As with other functionality, some of the changes were specific to single-player and others applied to both single-player and multiplayer. Developers sometimes accidentally applied the single-player specific changes to multiplayer games. They sometimes accidentally deleted necessary steps for multiplayer and replaced them with incorrect single-player-specific steps. Developers also forgot
to implement changes applicable to both types of games for multiplayer. For example, a new mechanism was developed to download and cache media files (images, audio, video) in games. This change applied both to single-player and multiplayer games. However, the multiplayer game start code was not using this mechanism.

These problems essentially made it so that it was impossible to start a multiplayer game. The app kept crashing and/or freezing. I spent most of the initial part of my MEng trying to solve these problems and actually get a multiplayer game running.

5.2 Solution

I had to first fix the issues with the game-start process. To do this, I went back to the state of the codebase when multiplayer games could properly be launched. I understood this code and created a flowchart of all the steps required to start a multiplayer game. Then, I used this flowchart to fix the current game start process for multiplayer games. I added in any missing steps and removed any incorrect ones. I also added in new code that was implemented for single-player but forgotten for multiplayer (such as the new mechanism to download and cache media files mentioned before). This meant that I had to understand how single-player games were started as well.

After I actually got a multiplayer game up and running, I did a large amount of testing to identify other problems. I built small multiplayer games to test the previously implemented functionality. I fixed problems mainly with standard debugging. However, I sometimes did go back to the state of the codebase a year ago to see how the code had changed (similar to how I fixed the game start process).

5.3. Future Work

In order to make simultaneous development of multiplayer and single-player functionality easier, a number of steps can be taken. In many cases, we simply need more documentation in the code. This is very important in source files in which multiplayer and single-player code is mixed. Single-player developers need to be able to understand the multiplayer code and vice versa. They need to understand what the consequences of changing the code will be.
In another light, if certain source files contain a lot of different logic for multiplayer and single-player games, we might consider making two different versions of the source file. This separation will prevent developers of single-player functionality from accidentally breaking multiplayer functionality and vice versa. It also allows for somewhat separate/disconnected single-player and multiplayer development. Developers have to focus and it is unrealistic for them to fully understand both types of games.

In the specific case of the game start process we can mitigate the problems by using a different code design. The problem is compounded by the fact that the game start logic is spread out across so many files and so many methods. Each method calls some other method and the chain becomes complex. It could be better to have a central “game start manager” that controls the process. This manager would delegate tasks to “workers” in the program. When a worker finishes a task, the worker would send an event to the manager indicating that the task has been completed. Then, the manager can start up a new task. In this design, the manager is the only one who knows all the steps in the process. The logic is centralized in one place and not spread out across so many methods. With this pattern, it becomes easier to identify problems in the game start logic for both multiplayer and single-player games.
Chapter 6

Small-scale Features

This chapter discusses a series of small-scale features I built to bring TaleBlazer closer to production-level software. Some of these features solve key technical problems and usability issues. Others address underspecified/unclear behavior for parts of the multiplayer system.

The first feature allows users to quickly reenter a multiplayer game after exiting the game. The second feature allows users to rejoin an instance as a new player. They can leave the instance and rejoin with a new role and a different team. The third feature involves defining how long multiplayer instances should last. The fourth feature involves adding new in-game UI elements to improve the gameplay experience. The fifth and sixth features involve specifying and implementing the behavior of certain blocks in multiplayer games. It was unclear previously how these blocks should behave in a multiplayer game. Finally, the seventh feature solves a technical problem involving when instances and players are actually created on the server.

6.1 Resuming a Multiplayer Game

6.1.1 Problem

Oftentimes, when using TaleBlazer, users accidentally exit out of the app. They might inadvertently go to their device’s home screen (by pressing the “home button”), the app might unexpectedly crash, or the device might run out of power. Suppose a user was playing a TaleBlazer multiplayer game when this happened. The user would want to quickly rejoin the game. To do this, the user would have to go back to the “Game Code” screen to enter the instance code. This requires the user to remember the instance code. If the user has forgotten, he/she will need to ask their friends for the code again. This process is laborious and inconvenient. Users should be able to quickly reenter the game. This is especially important in highly competitive games.

This problem also occurs when users voluntarily decide to take a break from a multiplayer game. A user might decide to leave for a short period, say 15 minutes, and then resume playing. The user should not be required to remember the instance code or find it again.
6.1.2 Solution

To solve this problem, I implemented a “multiplayer resume” feature. For each game, the instance code for the last played instance is saved in a database on the mobile device. When the user visits the game page for the game, he/she will be shown a “Resume” button. The game page is the page at which users can choose to spin up a new TaleBlazer game. See Figure 10 for an example of the game page. When the user presses the “Resume” button, he or she will reenter the instance. All player data will still be present. The player will still have the items in their inventory that they previously had. The player’s traits will have the same value they previously had and won’t be reset.

To implement this, I added a new message to the server/client protocol called a “resumelnit” message. This message contains the id of the player who wants to resume playing, and the code for the instance in which the player wants to play. When the server receives this message from a client, it will check if the specified player is already in the instance. If not, it will ignore the message. If the player is already in the instance, then the server will send the client the current game state. The client receives this game state and relaunches the game on the mobile device. The user can then resume playing as normal.
6.2 Rejoining an Instance as a New Player

6.2.1 Problem

Recall that we use a mobile phone or tablet’s device id to uniquely identify a player within an instance. This means that each device can correspond to only one player inside of an instance. This can be problematic during user testing sessions. Game designers can create games with an arbitrary number of roles and teams, but they may not have sufficient devices available to test all combinations of roles and teams. Suppose a game designer only has two devices available, but his game has three roles (call them Role1, Role2, and Role3). The game designer finds a friend and they both begin playing in a new instance. Initially, the designer picks Role1

Figure 10 – The “game page” for a multiplayer game. The user can start a new instance by pressing the “Start New Game” button. The user can resume playing in the last instance he/she was playing in by pressing the “Resume” button.
and the friend picks Role2. After playing some time, the designer decides that he/she wants to test out Role3. However, he/she doesn’t want to start up a new instance and lose all existing progress in the game. The game designer would have to acquire another device in order to test out Role3.

This problem doesn’t just arise in user testing sessions. It can even occur with finished games. A very important case is when a user accidentally selects the wrong role when entering an instance. This user would not be able to rejoin the instance with the correct role without acquiring a new device. The user could spin up a new instance, but this wouldn’t make sense if many of his friends are already playing in the other instance.

6.2.2 Solution

To solve the problem, I implemented functionality for players to “delete themselves” from an instance. When a player leaves a game, they are asked if they want to “delete themselves” from the current instance. If they select “yes,” all references to the player will be removed and any items in the player’s inventory will be automatically dropped.

To implement this new feature, I added a new message called “deletePlayerFromInstance” to the server/client protocol. This message contains the id of the player to delete and the code of the instance to delete the player from. When the server receives this message, it removes all references to the player in the instance in the backend data structures. The server also executes code to drop any agents in the player’s inventory. This involves executing the “when player drops agent” scripts.

Once the player has been deleted from the instance, he/she can reenter the instance as a new player. The player will not exist in the backend structures and thus, it will be the same as if the user had never played in the instance.

6.2.3 Future Work

This new “delete player from instance” feature is useful, but it can be problematic because the feature is always available. Suppose a multiplayer game is almost over and a player is on a losing team. The player can just delete himself from the instance and join as a new player on the winning team. This is not desirable behavior.
In the future, we want to give game designers control over this feature. The designer should be able to decide when this feature is and isn’t available. For example, a game might have an “initial phase” where players can try out different roles and experiment. This initial phase could be controlled by gameplay logic (e.g. when all the players reach a certain landmark, the actual game starts and players can no longer rejoin as a new player).

Specifically, we will probably want to offer this feature as a block in the blocks based programming language. The designer could use this block to turn the feature on/off at any point in the game logic. This would handle most use cases of this feature. The only problematic case is where the designer wants the feature to be available for a specific length of time (e.g. players can experiment and try out roles for 15 minutes). TaleBlazer doesn’t currently support any sort of timers. This is discussed in more detail in section 10.7 as part of Future Work.

6.3 How Long Should Multiplayer Instances Last?

6.3.1 Problem

It was previously not specified how long multiplayer instances would last. Players did not know how long an instance would be playable. This is important information. Players should know when an instance expires and what will happen if they try to join or resume playing in an expired instance. Specifying this is also important for developers on the TaleBlazer team. Developers need to know when it is safe to delete instances from backend data structures.

6.3.2 Solution

To solve this problem, I first had to pick a value for how long instances should last. I discussed this with other members of the TaleBlazer team and with TaleBlazer game designers. It is memory/data-intensive to support multiplayer instances forever or for large periods of time. There are just too many instances to support. Also, supporting instances for a large period of time makes it difficult to develop new features. Changing and redeploying the server might make old instances unplayable. Thus, a relatively short timeout value should be picked. In the end, I chose a value of 48 hours. I believe this is a reasonable value as it gives players enough time to play the game and it is convenient for developers.
After 48 hours, players will no longer be able to join the instance or resume playing in it. When the player tries to join/resume, he/she will see an alert on the mobile device saying that the instance has expired. To implement this, I changed the multiplayer server to save the time of creation for each instance. Whenever a client sends a message to join or resume playing in an instance, the server uses this time of creation and the current time to compute the elapsed time since the instance was first made. If the elapsed time is greater than 48 hours, the server sends a special “expiration” message to the client. With this implementation, changing the specific value of 48 hours is not hard. The server is the only part of the system that knows this value. We can simply change the value in the code and redeploy the server. We do not need to make a new version of the TaleBlazer app. Users do not need to download any updates for TaleBlazer.

6.3.3 Future Work

The actual number “48 hours” could change in the future in response to user testing feedback. We could also consider allowing the game designer to specify how long the instance lasts in the editor. If the game designer has specified a value, then the server would need to use that value instead of the fixed value of 48 hours.

6.4 In-game User Interface Enhancements

6.4.1 Multiplayer Game Introduction

6.4.1.1 Problem

Suppose a user starts a new multiplayer instance and then wants to invite his friends to join the game. The user can do this by giving his friends the instance code and having them enter the code on the “Game Code” screen. However, if the user does not know this, then he/she will have a problem. When the user starts the instance, the TaleBlazer app does not tell the user how other players can join the game. The user will have to stop playing the game and somehow gain the information (by consulting online documentation, by asking somebody else, etc.). This is inconvenient and disrupts the user’s game experience.
6.4.1.2 Solution

When a user starts a multiplayer instance, he or she is presented with a short “multiplayer game introduction” telling them the instance code and how others can use this code to join the instance. To implement the UI for this screen, I reused the standard TaleBlazer UI for say blocks. See Figure 5 for an example of the say block. We decided that developing a different UI was, at this time, not necessary. It would take extra time and development effort that could be devoted elsewhere. For now, we decided that just conveying the information is the important task. See Figure 11 for a screenshot of the multiplayer game introduction.

6.4.1.3 Future Work

Because the “multiplayer game introduction” looks similar to the text displayed by say blocks in game, users might mistake the introduction for actual game content. We might want to differentiate the UI for the introduction so that users recognize it as meta-game content. We can do more user testing to judge whether such a UI change is necessary.
Figure 11 - The new “multiplayer game introduction.” This screen is shown whenever a player joins a new instance.

6.4.2 Instance Code Visibility

6.4.2.1 Problem

Suppose a user (call him Ben) is already playing in a multiplayer instance. A friend of Ben (call her Alyssa) then wants to join that instance. Alyssa asks Ben for the instance code. Ben is busy playing the game and has forgotten the instance code. He needs to find it somewhere in the app. In the previous version of multiplayer, Ben would have to go to the sandwich menu at the top right of the screen and then tap on “About Game.” Here, Ben would see the instance code. This is disruptive to Ben’s gameplay experience. Ben has to essentially pause playing the game and navigate to this screen to find the instance code. The problem is worse if Ben doesn’t know that the instance code is on this “About Game” screen. Ben will have to ask someone how to locate the instance code or simply navigate through the app until he finds it. In summary, finding the instance code is too difficult/disruptive for players inside a game.
6.4.2.2 Solution

I changed the app so that the instance code is always visible. It is included next to the game title. See Figure 12. In the scenario discussed previously, Ben can now pretty much instantaneously tell Alyssa what the instance code is. The tradeoff for this is that for games with long titles, a smaller amount of the title will be visible at any time (since we have limited screen space). We decided that this was an insignificant issue, since players do not use the game title in gameplay.

![Figure 12 - Making the instance code always visible. The instance code here is ibdbj. Notice how it appears at the top of the game window next to the title of the game.](image)

6.4.3 UI During Agent Pickup

6.4.3.1 Problem

In order to maintain the consistency of the game world, certain actions have to be mediated by the server. The most important of these actions is picking up an agent. The server
must mediate the pickup so that two different players don’t simultaneously both have the agent in their inventory.

Because of this server mediation, there can be a delay after a user picks up an agent. The user has to wait for the server to “okay the pickup.” Previously, there was no UI during this waiting period. This is not an issue when network connectivity is fast and the user almost immediately receives a response from the server. However, when the Internet connection is poor, the delay will be noticeable. During this delay, users might not realize that the pickup is in progress. Users might think that the app has simply ignored their request to pickup the agent or that some bug has occurred.

6.4.3.2 Solution

To solve this problem, two additional user interface elements were added. With these UI elements, users know that their pickup request is in progress. First, when the user presses the button to pick up an agent, that button changes its text from “Pick Up” to “Pick up in progress.” The button is greyed out to make sure players understand that the action is currently unavailable. Second, I made a change to the Inventory tab (recall that TaleBlazer has an Inventory tab where users can see all the agents they have picked up). I added a greyed out row for agents whose pickup is currently in progress. This sends the signal that the agent could in the future be part of the user’s inventory, but is not right now. If the user’s pickup request is accepted by the server, the greyed out row will convert to a normal row. The “Pick up in progress button” will disappear. If the agent pickup is rejected because some other player picked up the agent first, the user will receive a toast message saying that another player picked up the agent (this was previously implemented functionality and was discussed in section 3.2). The greyed out row will be removed from the inventory and the “Pick up in progress” button will disappear. See Figure 13 for screenshots of the added UI elements.
6.5 Defining the Behavior of and Implementing Looks Blocks

6.5.1 What Are Looks Blocks?

In the editor, TaleBlazer provides a number of what are called “looks blocks.” These blocks change a player’s user interface, but do not affect the game state. These blocks include:

- Say blocks, which simply show some text on the screen (see Figure 5 for an example usage of the say block)
- Switch to tab blocks, which switch a player’s active tab
- Blocks to show and hide actions for players, agents, teams, and the world
- Blocks to show and hide traits for players, agents, teams, and the world
- Blocks to include or exclude certain tabs in the game UI
6.5.2 Problem

The behavior of looks blocks in multiplayer games was previously underspecified. For example, suppose a game has the following logic: “when player bumps Policeman agent, say ‘hello’”. When a player bumps the Policeman agent, should the text “hello” appear on every player’s screen? Or does it only appear on the screen of the player who actually bumped the agent?

As another example, suppose a game has two agents, one representing a virtual “Blacksmith” character and another representing a virtual “Hammer” item. Suppose also that the “Pick Up” action of the hammer is hidden by default. Now, suppose the game has the following logic: “when player bumps Blacksmith agent, show action Pick Up of Hammer agent.” When the player bumps the Blacksmith, the Pick Up action of the Hammer agent will become visible. Should this change in action visibility apply to all players? Should players who haven’t bumped the Blacksmith agent and go directly to the hammer also see the “Pick Up action?”
Most other blocks affect all players. The changes caused by the block are propagated to the server and then the server routes these changes to all other players in the game. The key question is whether looks blocks should behave the same way.

### 6.5.3 Solution

#### 6.5.3.1 Deciding How Looks Blocks Should Operate

We decided that the looks blocks should not operate like the other blocks. Looks blocks should only affect the current player’s UI. I will first explain how this decision applies to the two examples given previously. Then, I will explain the more high-level reason for this decision.

In the first example, the “say hello” is part of that specific player’s interaction with the Policeman agent. It doesn’t make sense to show the text “hello” on the screens of players who are not bumping the Policeman. The say block is specific to one player’s game experience. In the second example, we argue that the new visibility of the Hammer’s “Pick Up” action is a property specific to the player who bumped the Blacksmith. The change in visibility is a direct result of that player’s actions and should not occur for other players.

This leads into the larger reason for implementing looks blocks in this way. We want to be able to support selective visibility of actions, traits, etc. inside the game world. This visibility could depend on what the player is currently doing (as in the case of the player bumping the Policeman), actions the player has taken before (as in the case of the player bumping the Blacksmith agent), or depending on the player’s role and team. The last of these is very important for game designers. We want to allow players of different teams and roles to see the game world differently.

This is best illustrated by example. Consider a game with two roles “Engineer” and “Medic.” In this game, there might be an agent called “Blueprints” with an action called “Read.” The game designer would want to specify that only players of role “Engineer” can perform this action “Read.” Players of role “Medic” should not have this ability. If the looks blocks operate in the stated way, the game designer can implement this functionality easily. The designer can first hide the “Read” action by default. Then he/she can specify logic such as “When player bumps Blueprints, if player is of role Engineer, show action “Read” of Blueprints”. Since the show action block does not affect all players, players who are of role “Medic” who later bump the
Blueprints will not see the “Read” action. See Figure 15 for another example usage of the Looks blocks. That example relates to selective visibility based on teams.

In summary, our approach to the looks blocks is to have them apply only to the current player. Say blocks only execute on the current player’s device. The visibility of actions and traits is specific to each player. Which tabs are included in the player’s UI is specific to that player. Players are in the same game world, but they might view the world and interact with the world differently.

6.5.3.2 Implementing the Looks Blocks

One might think that the looks blocks are very simple to implement. They only need to be applied locally, so we do not need to involve the multiplayer server at all. This is true for the say block, but not the other blocks. As described in section 6.1, a player can quit a multiplayer game and resume playing. When the player resumes playing in the instance, he/she receives the game state from the server. Suppose a player executes some logic to show an action of an agent. Then, the player quits and resumes the game. If the server does not store the change in action visibility, the player will lose that change.

Because of this, a proper implementation of the looks blocks does need the multiplayer server. Whenever a visibility change happens, a standard update message is sent to the server describing this visibility change. The visibility change is just like any other gameplay update. (e.g. changing a trait). When the server receives the update message, it updates the master game state to store this visibility change. The server stores each player’s visibility of traits, actions, and tabs. However, the server does not relay this change in visibility to other players. Other players do not need to know about this visibility change. Note that the actual implementation of the looks blocks was not difficult. The harder challenge was to actually formulate how they should behave in a multiplayer game.

6.5.3 Future Work

Currently, the game designer cannot specify logic to change the UI of every player in the game. This functionality could be useful. For example, suppose that when a game ends, the designer wants to “disable” the UI on every player’s device. This would indicate to players that the game has finished. Specifically, when the game ends, the designer might want to remove
certain tabs from the UI or hide certain actions from agents. To allow game designers to encode such logic to simultaneously change the UI of every player, we will need to add additional blocks in the editor. This idea is still in its infancy and will need to be fleshed out significantly.

![Example usage of the looks blocks in a multiplayer game. The game designer can selectively show an agent's actions depending on the player's team. Here, the "Hug" action for the "Beaver" agent is only shown to members of Team MIT. If looks blocks applied for all players, this would not be possible. Players not on Team MIT who later bump the beaver agent would incorrectly have the Hug action already visible (the if/then statement to show the action would be irrelevant, since the visibility has already been changed).](image)

6.6 Multiplayer “When Game Starts” Block

6.6.1 Problem

In the TaleBlazer Editor, there is a “when game starts” block. For single-player games, designers can use this block to specify logic to be executed at the start of a game. For example, the block can be used to set the initial value of traits for the game world (e.g. a “Number of Minerals” trait of the world could be set to have initial value 1000).

It was underspecified how this block would operate in multiplayer games. Specifically, would this block execute when the instance is created or each time a player joins the instance? I decided between these two choices and then implemented the code to actually execute the block.

6.6.2 Solution

After discussions with the TaleBlazer team and other game designers, I decided that the “when game starts” block should execute once when the instance is created. It should not
repeatedly execute every time a player joins a game. We felt this was the most intuitive
definition for the block. The game really starts when the instance is created, so this is very
natural.

There are some interesting consequences of this decision. Because the block executes
when the instance is created, the block is not linked to any players. The block is executed before
there are any players in the instance. Thus, any player specific blocks inside of the “when game
starts” block must be ignored. It is meaningless to have logic such as “when game starts, set
score of player to 0.” It is unclear what player this logic is referring to. There are no players in
the game yet. It could possibly mean the player who started the instance, but this is a confusing
definition. It is better to use the “when game starts” block to only initialize properties of the
game world itself. Other blocks can be used to initialize player specific data (see the discussion
in the next section about a possible “when player joins” block). This allows for a clean
separation between world-specific properties and player-specific properties.

To implement the “when game starts” block, I wrote some new server-side code. It is
natural to execute the block on the server because the server is actually creating the instance.
Whenever the server creates a new instance, it reads and executes all the blocks under “when
game starts.” I needed to modify the standard block execution code to ignore any player-specific
blocks. This includes blocks that explicitly refer to a player and the looks blocks discussed in
Section 6.5. Recall that the looks blocks execute locally on each player’s device. When we
execute the “when game starts block” on the server, there is no player device, so the looks blocks
are meaningless.

6.6.3 Future Work

The major weakness of this approach is that the game designer is allowed to use blocks
inside of “when game starts” that actually have no effect (looks blocks and player-specific
blocks). This is confusing for the game designer. A better approach is to actually prevent these
blocks from being used in the editor. This is less confusing for the game designer and actually
makes implementing the “when game starts” block easier. We no longer will have to ignore
certain blocks when executing on the server.
We will have to decide how best to do the user interface for these disabled blocks. For example, will the blocks be greyed out in some way? Will there be an error message when you try to save and compile the game logic if you are using one of these prohibited blocks?

A related block we can implement is a “when player joins” block. This block would execute every time a player joins an existing instance. It can be used to initialize data specific to each joining player (such as a player’s health trait). The block can also be used to show tutorial information to each joining player.

6.7 Preventing “Dead Instances” and “Dead Players”

6.7.1 Problem

In the previous implementation of multiplayer, there were some problems with instance creation and the adding of new players to an instance. The problems came from the time at which these things happened.

When a user starts a multiplayer game, he/she is shown what we call the “role selection screen.” This screen prompts the user to select their username, their role, their team, and the “scenario” for the game. Scenarios are different themes/cases for the same game. For example, a game designer might use scenarios to allow the player to select between a long version of a game and a shorter version of a game.

The key thing is that TaleBlazer games cannot be played until a scenario is selected. In the previous implementation of multiplayer, the multiplayer server would create the instance before the role selection screen was prompted. If the user exited out of the app or simply hit the back button while on the role selection screen, then there would be what we term a “dead instance.” The instance would exist in all of the backend structures on the server, but it would not have a scenario and nobody would be playing in it. Furthermore, no one could ever play in the instance in the future. Nobody would know the instance code for this instance (the user did not actually start the game and get this information).

A similar problem occurs when a player is joining an existing multiplayer instance. The user is prompted with the role selection screen to select their username and role. In the previous version of multiplayer, the server would add the player to the instance before he/she completed the role selection screen. If the user backed out of the role selection screen, then there would be
what we term a “dead player” in the multiplayer instance. The player would exist in the backend structures on the server but would have no role, username, or team. This is actually a bigger problem because it can cause problems for in-game player interactions. For example, when a player wants to give an agent to another player, he/she will be prompted with a screen to select the player to give to. On this screen, the giving player could see such a “dead player” who doesn’t actually exist in the game.

6.7.2 Solution

I added an extra message to our server/client protocol called “acquireGameStateForRoleSelection.” The message takes as a parameter the specific instance for which the user wants the game state (more specifically, the instance code). When the server receives this message from a client, it simply relays the current state for the instance. It does not create a new instance or add a player to an existing instance. The server does not modify its state in any way. It just relays the message.

When a user tries to start a multiplayer instance or join an existing instance, the client will first send the “acquireGameStateForRoleSelection” message to the server. The client will extract from this game state the data necessary to show the role selection screen. If the player is starting a new instance, the game state will indicate that a scenario has not been selected. The user can select it on the role selection screen. However, if the user is joining an existing instance, the scenario will already be set and the user won’t be able to choose it.

Once the user has successfully completed the role selection screen, the client will send an “init” message to the server. This message will contain the inputted scenario (only if the user is creating a new instance), username, role, and team. When the server receives this “init” message, it will do one of two things. If the user is creating a new instance, the server will initialize a new instance and set the scenario. If the user is joining an existing instance, the server will add the player to the instance in its backend structures with the specified username, role, and team.

If the user backs out of the role selection screen, the client won’t send this “init” message. Note that no state changes on the server. The server will have simply relayed the current game state to the client. It will not have created any “dead instances” or “dead players.”
Chapter 7

The Chat Game Mechanic

7.1 The Problem

In multiplayer games, players could need to communicate. When players are nearby each other, they can just verbally talk with each other. When they are far apart, this is not possible. TaleBlazer does not have some sort of long-distance communication mechanism.

Communication in multiplayer games is important for a variety of reasons. Communication allows for team play dynamics such as leadership and collaboration. We hypothesize that without a way to communicate, players are more likely to ignore the teams and play by themselves. Communication also could enhance learning in educational games by preventing players from "getting stuck." If players cannot solve a certain gameplay challenge, they won't get bogged down and frustrated. They can ask their fellow players/teammates for help. Also, if TaleBlazer provided an in-game communication tool, players could practice synthesizing information that they receive from other players. Players would receive clues from other players and would put the clues together to solve gameplay puzzles. This is a valuable learning experience. Finally, a communication tool gives players an additional medium for learning. By discussing concepts with other players, players get more practice with the concepts. There is more opportunity for deep and lasting learning. All of this motivates in-game "chat" functionality in TaleBlazer.

7.2 Priority of Chat

One might ask why I chose to focus on this problem of communication instead of other problems. Why is providing chat functionality more important than adding other game mechanics (some examples of these are given in Chapter 10)?

One reason is that chat is a natural evolution of the previous work done on teams (see section 3.5.2). If players can play on teams, then they will want to communicate with their team members. They will want to formulate strategies with team members and collaborate to solve problems. Indeed, as mentioned before, the teams functionality might not be as effective without
a communication mechanism. Players could be more likely to ignore the teams if they cannot communicate with team members.

Second, as discussed in the previous section, player communication can have a big impact in educational games. If players can communicate with others, they might simply be more engaged in the game. Players can also work together to co-construct knowledge. Players can engage in leadership. They can assist other players who are struggling with in-game concepts. Communication allows for many good learning experiences.

Third, chat can be implemented without significant changes to the technical architecture. In other words, the system is already set up in a way to provide chat. The Node.js multiplayer server can listen for chat messages and relay the messages between the mobile clients.

Finally, in a larger sense, we (the TaleBlazer team) believe that communication is a foundational requirement. If players exist in a shared world, they should feel connected to each other. They should be able to interact with other members of the shared world. This requires the ability to communicate with others. Other TaleBlazer game designers who we talked with felt similarly about the importance of chat.

7.3 Concerns with Providing Chat

There are a number of possible issues with providing a chat environment. These issues are all especially important in the educational space. First, players might waste time with social chat unrelated to the actual gameplay. Second, players might use the chat to transmit inappropriate content. Finally, players could engage in abusive and bullying behavior towards other players.

Because of these issues, we anticipated that providing player-to-player chat was not a good idea. We decided to, for now, only support team chat and all-players chat. Making all chat public (at least on a team level) may reduce the chances of these problems. Also, moderators or chaperones can see problems occur and intervene in real time.
7.4 User Interface for Showing/Organizing Chat Messages

As mentioned in Section 1.2, TaleBlazer has a series of tabs at the top of the game window that represent key functionality in the game. I implemented a separate tab dedicated to chat.

I first had to decide on a UI for showing chat messages to the user. I proposed two possible ideas called the “Email View” and the “Facebook View.” I built paper prototypes for both of these ideas and gathered feedback from other users and the rest of the TaleBlazer team. The following subsections explain this paper prototyping process.

7.4.1 Paper Prototype 1 – The “Email View”

The first proposed approach for chat was called “the email view.” In this approach, the chat messages are organized like emails. The most recent chat messages appear at the top and the older ones appear at the bottom (just like in any email client). Certain messages are targeted towards teams and others are targeted towards all players. See Figure 16 for a sketch of the email view.

Figure 16 – The “email view” for chat. This UI is similar to an email inbox. More recent chat messages appear at the top. Notice how this view mixes team and all-players chat. Some of the messages were sent to Team MIT and some were sent to all players.
7.4.2 Paper Prototype 2 – The “Facebook View”

The second proposed approach for chat was called the “Facebook View,” mainly because of its similarity to the widely used Messenger app developed by Facebook. In this view, there are two separate channels – one for all players chat and one for team chat. When the player clicks on a channel, the player is redirected to a screen for that channel. On this screen, messages at the top of the screen are older and messages at the bottom are more recent. Messages sent by the current player are aligned to the right and messages sent by others are aligned to the left. See Figure 17 for a sketch of the “Facebook View.”

![Facebook View Sketch](image)

Figure 17 – The “Facebook view” for chat. This UI is similar to the Messenger app and other mobile chat clients. There are two channels, one for all players and one for the player’s team. These are shown on the left. When the user taps on a channel, he/she is shown all the chat messages in that channel. More recent chat messages appear at the bottom. The user scrolls downward to see more recent chat messages.

7.4.3 Feedback from Paper Prototyping

Feedback was collected from a variety of people, including other members of the TaleBlazer team and some TaleBlazer game designers.

Most people thought that the “email view” would make the chat less distracting. It is less similar to other chat UIs, such as the UIs of Messenger, Google Hangouts, WhatsApp, etc. Thus,
users would not get absorbed in the chat. They would treat it as simply a list of messages from other users. Chat would only be used for necessary game communication.

However, for some of the same reasons, people thought that the “email view” was very problematic. The UI was too different from other chat UIs to be useful to players. When testing out the paper prototype, users found the email view non-intuitive in that new messages appeared at the top. They were used to scrolling downward to see the most recent messages. Another problem users identified with the “email view” is that it mixed team chat and all-players chat. Users wanted a clearer separation.

Users were concerned that the “Facebook view” would be distracting, but overall felt it was the better approach to take. They felt TaleBlazer should be consistent with other chat UIs. They wanted a fast, fluid chat interface that they were familiar with. However, users said that having a separate page just to show the two channels (all-players and team chat) was unnecessary. They felt it was an unnecessary level of redirection. Users could not quickly switch between the all-players chat and the team chat. They felt it would be better to have tabs at the top of the chat screen to switch between the two channels. This would be a more efficient approach. See Figure 18 for a screenshot of the final UI for the chat tab.
Figure 18 – The final UI for TaleBlazer chat. Users type messages in the text box and propagate them by tapping the “Send” button. The messages are organized in a table. The table is scrollable. There are tabs at the top of the chat window to switch between all players chat and team chat.

7.5 User Interface for Unread Chat Messages

Users need to know when they have received a new chat message if they are not currently on the chat tab. At first, we thought it was best to show a toast telling the user that he/she received a new message. Users would see a short popup message on their screen. The popup would fade after some period of time. Users could check the chat tab to see the new message whenever they wanted.

However, there are a couple of problems with this approach. First of all, it can be distracting since chat messages can appear very often. If a player is focused on solving a puzzle or is interacting with an agent, he/she does not want to see unrelated chat messages on his/her screen. Second, since the toast is temporary, users might miss the message if they are looking up at the real world. They would have no indication that they can go back to the chat tab and see the
missed messages. The insight here is that there needs to be some permanent indicator on the UI for missed chat messages.

For these reasons, we decided to show an “unread chat count” on the chat tab itself. This is similar to how other chat clients handle unread chat messages. Whenever a user receives a chat message, but he/she is not on the chat tab, then the unread count shown on the chat tab will be incremented. When the user goes back to the chat tab, the count will be set to 0.

Because we have two channels, all-players chat and team chat, this general description is not entirely accurate. Each channel has its own unread count. The unread count shown on the chat tab is actually a sum of the unread counts of the two channels. If a user is on the team channel and receives a message for all-players, then the all-players unread count will increment. When the player switches to the all-players channel, then the count for that channel will reset to 0. See Figure 19 for a screenshot of the unread chat counts.
7.6 Implementation

The previous sections discussed the UI for the chat feature. This section discusses the technical details regarding how chat is implemented. In essence, chat messages are treated similarly to standard gameplay update messages (see section 3.2). Whenever the user types in a chat message and hits “Send,” the client sends an update message to the server. Within it, this update message contains the id of the player sending the chat, the actual chat text, and the “target” of this chat message. The “target” can be either “all players” or “team.”
When the server receives such an update message, it first checks the target. If the target is “all players,” it routes the chat message to the rest of the players in the game. If the target is “team,” it routes the chat message only to members of the sending player’s team.

The server persists the chat messages along with other aspects of the game state. The chat history is stored along with all the other game data (traits, locations of agents, etc.) for each instance. This persistence is necessary because intermittent connectivity can cause a player to disconnect and then reconnect to the server. That player should be able to recover the chat history and see all the messages sent in his/her absence. This also applies when a player quits and resumes a TaleBlazer multiplayer game. The player should be able to see all the chat messages sent while he/she was not in the game. In another light, persisting the chat data allows us to later view the chats and analyze how players are using the chat functionality.
Chapter 8

The Proximity-Constrained Give Action

8.1 The Problem

The previously implemented give action allows players in a multiplayer game to give agents to each other. However, there are no limitations on this mechanic based on a player’s real-world location. Players can exchange agents regardless of where they are in the map.

This can be problematic because it is not how “giving” works in real life. In real-life, people can only give items to each other if they are nearby each other. The give action can violate the realism of TaleBlazer multiplayer games. For example, consider a game in which players collectively fight a disease outbreak. The game might have different regions corresponding to different cities/areas in which the disease has broken out. Players in this game shouldn’t be able to exchange items (e.g. medicine, medical equipment, etc.) if they are not in the same city. This would make the game less realistic and plausible. The game would be a less realistic simulation of an actual disease outbreak.

Not all games actually need to be realistic. For example, certain fantasy and futuristic games are not grounded by real-world rules. In these types of games, an unrestrained give action might not be a problem. However, since many TaleBlazer games have a goal of realism, this is an important problem to solve. Thus, I propose a new proximity-constrained give action. Since this mechanic is not useful for all games, the game designer should have the ability to enable or disable the feature. Furthermore, the designer should be able to control the actual distance threshold at which players can exchange items with each other. This allows the designer to simulate different scenarios. For example, in some games, players might only be able to give to each other if they are right next to each other, simulating a “hand off” in real life. In other games like the one concerning a disease outbreak, players might be able to give to each other if they are in the same region. This can simulate players being connected to all other players in their city, but not to people in a different city.
8.2 Priority of Proximity-constrained Give Action

As with chat, one can ask why I chose to build the proximity-constrained give action instead of other possible game mechanics. The reason is that I believe the added realism is powerful. Games become more like real-life and thus become more immersive. Also, as I just discussed, the game designer can vary the distance threshold to simulate different real-world scenarios. Furthermore, proximity-constrained give is a natural extension of the previously implemented give action. The technical architecture for the give action has already been implemented and can be augmented with new algorithms to measure player proximity. So, in summary, proximity-based give has a natural technical implementation and it can be useful and powerful for game designers.

8.3 Background – Location and Distance in TaleBlazer

In order to understand how I implemented the proximity-constrained give action, it is necessary to understand how player and agent locations are specified in TaleBlazer. It is also helpful to understand how the game designer specifies an agent’s bump radius (this will be defined shortly).

TaleBlazer games take place in a square geographic area, specified by the game designer in the editor. On top of the area, a coordinate grid is overlaid. This grid has (0,0) in the bottom left corner and (200, 200) in the top right corner. The coordinates of this grid are called game coordinates. The designer can use game coordinates to specify the locations of agents in the game.

The game designer can also specify the distance at which a player automatically bumps an agent. This threshold is called the bump radius of the agent. The designer can input the bump radius value both in terms of game coordinates and in terms of meters. As the player walks around the game world, the TaleBlazer app computes the player’s distance to all of the agents. If the player’s distance to an agent is less than the agent’s bump radius, the bump interaction is triggered.

If an agent’s bump radius is specified in terms of game coordinates, the app computes the distance between the player and the agent in game coordinates. Since player/agent locations are stored using game coordinates, this is easy. The standard Euclidean distance is used. If the
agent’s bump radius is specified in terms of meters, the app needs to compute the distance in meters. Some extra work is needed in this case. The algorithm to compute the distance is given below in Figure 20. It relies on converting game coordinates to latitude and longitude values. The specific way this conversion is done is not relevant to thesis and is not presented here.

All of this work on computing the distance between agents and players will be very useful in implementing the proximity-constrained give. I now discuss the implementation of proximity-constrained give in both the editor and in the app.

```javascript
function getMetersDistanceBetweenAgentAndPlayer(player, agent) {
  R = 6378100; // radius of earth in meters
  (AgentLatitude, AgentLongitude) = getLatLngFromGameCoordinates(agent.locationInGameCoords);
  (PlayerLatitude, PlayerLongitude) = getLatLngFromGameCoordinates(player.locationInGameCoords);
  x = (AgentLongitude-PlayerLongitude) * Math.cos((PlayerLatitude+AgentLatitude)/2);
  y = (AgentLatitude-PlayerLatitude);
  distance = Math.sqrt(x*x + y*y) * R;
  return distance;
}
```

Figure 20 – Algorithm to compute the distance between a player and agent in terms of meters. Note that the algorithm converts the game coordinates into latitude and longitude values using the library function getLatLngFromGameCoordinates.

### 8.4 Implementation in the TaleBlazer Editor

As mentioned before, we want to give game designers control over how agents can be exchanged in the game. We allow four options:

- Players cannot give agents to each other at all (in other words, the give action is disabled)
- Players can exchange items regardless of their location (this is the standard give action)
- Players can exchange items if they are in the same region (recall that regions are different areas of the map that can, for example, correspond to different chapters of a game)
- Players can exchange items if they are within the same region and within a certain distance from each other. The distance threshold can be specified both in terms of game coordinates and in terms of meters. This is consistent with an agent’s bump radius.
These four options are presented to the game designer in a new section of the TaleBlazer editor called “Multiplayer Settings.” This section can be expanded in the future with other multiplayer settings. See Figure 21 for a screenshot of this section in the editor.

![Multiplayer Settings](image)

Figure 21 – Settings for the give action in the TaleBlazer editor. This is where the game designer can enable the proximity-constrained give action and specify the distance threshold. The designer can input the threshold in meters or in game coordinates. The designer uses the dropdown to switch between the two units (the dropdown is located in the last option next to the value 50).

### 8.5 Implementation on the Mobile Device

Implementing the client-side code for proximity-constrained give presents some challenges. Normally, when a user presses the button to give an agent to another player, they are prompted with a screen to select the player they want to give to (we call this “the player picker page”). If proximity-constrained giving is turned on, we need to check if players are in range. To check if a given player is in range of the current player, we do the following. If giving is only allowed for players in the same region, we simply check if the two players are in the same region. The region each player is in is already stored as part of the game state, so this is easy. If a distance threshold is specified, we compute the current player’s distance to the other player and check if it is less than the threshold. We compute the distance between two players in the same way we compute the distance between an agent and a player (discussed in section 8.3). If the distance threshold is specified in game coordinates, we compute the Euclidean distance between
the two players’ game coordinate locations. If the threshold is specified in meters, we use the
algorithm from section 8.3 to compute the distance in meters.

If proximity-constrained giving is turned on, the UI for the player picker page will be
slightly different. First of all, we show some extra information. We show the region each player
is in. For players in the same region as the current player, we also show the player’s distance to
the current player. If there is only one region in the game, the region information is omitted.
Second, we had to decide how to differentiate between out of range and in range players. We
decided to show the out of range players as greyed out. There is a label next to the out of range
players saying that they are out of range. See Figure 22 for a screenshot of the player picker
page.

One might ask why we do not simply hide the players that are out of range. The reason is
that players can change from being out of range to in range. This is discussed in the following
subsection.

8.5.1 Player Starts Walking Towards Another Player

One of the key issues to handle is that a player can move around while they have the
player picker page open. Indeed, if they want to give an agent to another player who is too far
away, they will likely start walking so that they can get closer. Players will expect that the
distance to that player on the screen will dynamically change. In order to implement this, we use
a timer to periodically refresh the data on the player picker page. The timer is set to be every
second right now, but this value should be fine-tuned in the future. In general, using a shorter
refresh timer is better, because the distance values the player sees will be more accurate/up-to-
date as the player walks around. However, if the refresh timer is too short, it can affect
performance. The app will be doing distance computations too frequently. These computations
will be unnecessary because player locations will not change so often. Furthermore, there are UI
issues with too short a refresh timer. For example, the screen might “flicker.” The player picker
page will be refreshing so quickly that the screen will appear to constantly flicker. The screen
will not be responsive to user touches while it is refreshing so quickly.
Figure 22 – Proximity-constrained give action on the TaleBlazer app. This is the “player picker page”. Here, a player selects which other player to give the agent. The screen shows every player’s username, team and role. The player cannot give the agent to Lisa because she is out of range. Her radio button appears greyed out. The player can give the agent to Kurt because he is in range.
Chapter 9

Play Testing

I tested the small-scale features discussed in Chapter 6 as I developed them. I mostly tested them by myself, but with multiple devices present (to simulate multiple users). I conducted more formal sessions to test the new game mechanics of chat and proximity-constrained give. These sessions all involved members of the TaleBlazer team playing sample multiplayer games.

In another light, I also collaborated with a freelance professional game designer interested in TaleBlazer multiplayer. This game designer is an MIT alumnus and has a great deal of experience building and testing AR, location-based games. He built his own complex TaleBlazer multiplayer game and provided feedback on the system.

9.1 Testing Sessions for Chat

I conducted two testing sessions with the TaleBlazer team. The purpose of these sessions was to identify bugs in the chat functionality and load test the system. Each session involved 4-5 players. We played a very simple multiplayer game with chat enabled. The details of the game were not actually relevant to the tests, since we were just testing the chat functionality. The key point was that the game had teams, so we could test both all-players chat and team chat. The game also had many different tabs, which allowed us to test the unread chat counts functionality.

To identify bugs, we tried sending messages to each other and made sure the messages were correctly received. We sent messages in the all-players chat and made sure that everyone received them. Similarly, we sent messages in the team chat and made sure only members of the same team received the message. We also tried sending messages to players not on the chat tab and checked that the unread chat counts were correct. Finally, we tried quitting and resuming the game multiple times and checked that the chat messages still existed (meaning the chat history was correctly persisted on the multiplayer server).

Regarding load testing, we checked that the chat functionality worked correctly when many players were rapidly sending messages. We checked that no messages were lost and that
the unread chat counts were still correct. Finally, we tried sending messages with a lot of text and made sure that no problems were encountered.

In the first testing session, the basic functionality worked. We were able to successfully send messages to each other and the chat history was recovered when we quit and resumed the game. We didn’t encounter any problems under load. No messages were lost and there were no problems with very large messages. However, there were problems with the unread chat counts throughout the test. When there were more than two players in the game, the unread chat counts were incorrect. They were always much higher than expected, regardless of the load on the system. When many players were rapidly sending messages to each other, the unread chat counts actually spiraled to very large numbers. I fixed the problems with the unread chat counts before the second test. The specific details of the problems aren’t very relevant, but they involved the chat messages “bouncing” between server and client. The server would send the chat messages to clients and the clients would incorrectly send the messages back to the server and so on. During the second test, everything went smoothly.

Note that all of this testing was done to test the actual chat functionality. I ran out of time to test how users actually use the chat within the game. We will need to do this in order to judge how effective the current chat implementation is. This is discussed more in the chapter on future work in section 10.2.

9.2 Testing Sessions for the Proximity-constrained Give Action

I conducted two outdoor sessions to test the proximity-constrained give action. These sessions involved 2-4 users and took place on the MIT campus. The users were members of the TaleBlazer team and were playing a very simple game. This game had a number of agents scattered around the campus. All players were on the same team and the goal of the game was to collect all the agents and give them to a single player. The proximity constrained give action was enabled and the threshold was set to 20 game coordinates.

While playing this game, we first tested that users could pick up agents successfully while outdoors. Then, we tested that a player (call him/her Player1) could not give to another player who was out of range (call him/her Player2). Player1 should properly see Player2 as greyed out on the “player picker page” (recall that the “player picker page” is the screen where a player picks which other player to give an agent to). We then tested that as Player1 started
walking towards Player2, the distance values on Player1’s player picker page would dynamically update. When Player1 finally got within the distance threshold, he/she should be able to give an agent to Player2.

The first time we played this game outside, we encountered some problems. The proximity-constrained give action appeared to be generally working. Players saw each other on the “player picker page” when they were close by and were able to give agents to each other. However, when they were far apart, they didn’t see each other as “out of range.” They actually didn’t see each other on the player picker page at all (the player was just omitted). Also, the code to dynamically update the player picker page did not appear to work. As players walked toward each other, their player picker pages did not change at all. The second time we played this game, things worked out a lot better. The proximity-constrained give action functionality worked properly.

One issue we encountered during both sessions was intermittent connectivity. The MIT campus Wi-Fi has many different subnets, so we were moving in and out of subnets somewhat often while playing the game. When a player disconnected and then reconnected, most of the time, the player was able to get back into the game. However, sometimes, the player was unable to resume playing. The server would recognize that the client was trying to reconnect to the game. The server would try to provide the missing updates and acquire the missing client requests as explained in section 3.4. However, the server and client would somehow get into a bad state. The requestNum and updateNum values would become bogus. For example, the server would ask the client for requestNumber 4. However, the client had not yet made a fourth request. It was hard to reliably reproduce these sorts of problems. They happened unpredictably and the system was a little unstable.

The work on solving the intermittent connectivity problem was done before this thesis. The results of these testing sessions suggest that we need to revisit this work in the future. We need to identify what problems are occurring and why.

9.3 Long-term Collaboration with a Freelance Game Designer

TaleBlazer’s main focus is on educational games. TaleBlazer has mostly been used in the past to create games that are relatively simple in gameplay, but have clear learning value. In
other words, the games have been more like interactive learning experiences and have not had very intricate or involved gameplay mechanics.

We wanted to take things in a different direction and see how TaleBlazer can be used to create complex games. To do this, we collaborated with a freelance game designer who has great interest and experience in the area of augmented reality mobile games. This designer wanted to create multiplayer games that weren’t directly related to education, but included complex, strategic gameplay.

The designer used the TaleBlazer multiplayer platform to create a multiplayer, team-based game called “Espionage.” The game takes place in a park in Waltham, Massachusetts. The theme of the game is as follows. Teams are competing to find secret documents locked away somewhere in the world. There are virtual characters scattered throughout the world that provide important clues and items. Real world landmarks can also provide clues. The game involves solving puzzles, strategic thinking, collaboration, and a little randomness/luck. The game utilizes the two new game mechanics of chat and proximity-constrained giving and most of the small-scale features described in Chapter 6.

As the game designer built and tested his Espionage game, he identified bugs in our system. Many of these bugs hadn’t been identified before simply because the multiplayer games made during testing were too simple. As of now, the game designer is still building his game. In the future, he hopes to do a large-scale test of the game. This could actually be an ideal setting to first test how players use the chat feature while playing a game.

In another light, we also had lengthy discussions with the freelance designer about new functionality to add to TaleBlazer multiplayer. We came up with many interesting ideas and categorized them based on priority and difficulty. Most of the ideas were new game mechanics. However, some were also changes to the editor to make the game design process more streamlined. Many of these new feature ideas are described in the next chapter on “Future Work.”
Chapter 10

Future Work

Some future work was already discussed in Chapter 6 for the small-scale features I implemented. This chapter discusses additional future work.

The first future work item is to make TaleBlazer production ready. This involves testing the system thoroughly, for example under heavy load and in the presence of intermittent connectivity. After the system is production ready, more complex and interesting game mechanics can be built. The second and third future work items below involve possible extensions to the chat and the give action. The fourth item is a feature called “local agents.” This feature would allow users to have single-player sequences inside of a multiplayer game. The fifth item involves players being able to bump each other, instead of only agents. There are many possibilities for a player-to-player bump interaction. The sixth future work item involves more options for teams in a TaleBlazer game. The final option involves allowing game designers to use time as a game mechanic. More specifically, the designer can use timers to specify game logic to happen at specific time intervals.

10.1 Making TaleBlazer Multiplayer Production-Ready

As of now, TaleBlazer multiplayer is in an alpha stage. The only people using the system are members of the TaleBlazer development team and the freelance game designer mentioned in the previous chapter. Before we begin developing new features, we should get the system to production-readiness. I did some work in this area by building the small-scale features discussed in Chapter 6, but there is still work to do.

One task this involves is to rigorously test the system under the conditions of intermittent connectivity. In section 9.2, I discussed some problems that are unpredictably occurring with our procedure to reconnect a user. We need to find the cause of these problems and make the necessary fixes to our approach. In another light, we also need to simulate many more use cases related to intermittent connectivity. For example, an important case is when a user’s mobile device switches from Wi-Fi to data. The user will lose connection to the multiplayer server for at
least a split second and will need to reconnect. Before we add new game mechanics, we should ensure that the system can handle these types of situations.

In addition to testing the system under intermittent connectivity, we also need to do further load testing of the system. First, we should test longer multiplayer games. We haven’t yet had players play a “feature-length” multiplayer game, which can last for over an hour. We also need to test the system when there are many players in the same instance or when there are many instances running at the same time.

Most other tasks involved in getting the system production-ready will arise from extensive user testing. We need more people to create games and test them out. We can release beta versions of the software for people to use and provide feedback.

10.2 Chat

10.2.1 Judging the Effectiveness of the Current Chat Implementation

Before we do future work on chat, we will have to judge how effective the existing chat functionality is. The current chat is free-form - users can send any sort of messages to their team and to all other players in the game. The following are some research questions we can explore regarding the current free-form chat. We will have to perform user testing and data collection to answer these questions.

1. Do players teach or learn game content using chat? In what types of games does this teaching/learning through chat happen most? In other words, are there certain game design choices that cause players to use the chat in a more productive way?
2. Does the chat functionality distract players? If so, in what ways?
3. Does inappropriate and abusive chat occur and if so, what can we do to combat this?

Answering these questions will involve examining chat data after games take place. Chat messages are currently persisted on the server, so the data is available. However, we will likely have to do some work to make the data more accessible and easier to analyze.
10.2.2 Some Possible Extensions of Chat

As mentioned in the previous section, new chat features should be driven based on the results of analytics and user testing. In this section, I propose some general ideas that have come from discussion with other members of the TaleBlazer team and other mobile game designers.

One thing we almost certainly want to do is to provide game designers control over the type of chat in the game. Different games might require different forms of chat. Some games might want the free-form, flexible current version of chat. For other games, this might be too distracting. To make the chat less distracting, we could provide what we call a “limited-message chat.” The game designer can specify a set of allowed messages in the editor. The user can then select from these messages when trying to communicate with other players. We might even explore using images rather than text (a sort of hieroglyphics-based chat).

Right now, we only support team and all-players chat. Some game designers might want player-to-player chat as well. Indeed, if the game designer is limiting chat to a fixed set of messages, player-to-player chat might be less distracting. The key idea is that we want to give game designers control over the type of chat messages allowed and over whom the players can chat with.

10.3 More Options for the Give Action
10.3.1 Giving Agents Only to Team Members

A very straightforward extension of the give action is to support giving only to members of your team. This mechanic is useful when the teams are completely disconnected from each other and have no incentive/need to collaborate.

10.3.2 Market Areas

Multiplayer games (especially Massively Multiplayer Online Role Playing Games like World of Warcraft) often have “market areas.” These are special areas of the game world where players can buy/sell objects and make trades.

We might want to support such market areas in TaleBlazer as well. In other words, we would want players to be able to exchange items only in specific geographic areas of the map. This is related to the proximity-constrained give action already implemented.
10.4 “Local” Agents

In single-player TaleBlazer games, there are often short tutorial sequences. The player bumps into one agent and receives some information about the game. The player might be instructed to pick up some object or perform some simple actions. The player is then directed to bump another agent who provides some more introductory information. In other words, there is a short, controlled linear sequence at the beginning of the game.

A game designer might want to provide similar functionality in multiplayer games. Whenever a player joins an instance, they could go through a short tutorial that teaches them some key game mechanics. Providing such a tutorial experience is difficult in the current version of TaleBlazer multiplayer. As mentioned in section 2.2, the goal of multiplayer is to provide a consistent shared game world. All objects in this world are shared amongst the players. Suppose a tutorial sequence involved the player picking up a “Key” agent and unlocking a “Door” agent. The first player who goes through the tutorial will pick up the key and unlock the door. The world will be permanently altered and the game designer will be forced to reset the state in order to allow other players to properly go through the tutorial. If two players are joining at the same time, we have an even bigger problem. It is impossible for both players to properly go through the tutorial.

A potential solution for this issue would involve what we call “local” agents. What this means is that every player will have his/her own individual copy of the agent. Certain items in the game world will not be shared but will be specialized to each player. We will need to decide how best to implement this inside of the TaleBlazer editor. One idea is simply to have a toggle to switch an agent from global to local.

These “local” agents are useful for more than just tutorial sequences. They can be used in the core gameplay as well. A common thing that game designers do at game start is to provide each player with the same object. For example, in a game focused on construction/building, each player might have his/her own hammer. The players might be able to modify and specialize their hammer in different ways. It would not be possible for players to swap hammers in this game. In the current implementation of multiplayer, the game designer has no way to provide each player with his or her own copy of a hammer. With local agents, the designer could simply specify that the hammer is local.
10.5 Players Bumping Each Other

A player can “bump into” agents as he/she walks around the game world. When a player bumps an agent, the player begins an interaction with the agent on his/her mobile device. Right now, players can only bump agents. One feature idea is to allow players to bump other players in the game. When two players encounter each other in the real world, a new screen would popup on both players’ mobile devices. The players could then interact with each other using their devices.

There are many possibilities for what happens during the bump interaction. The players could see what is in each other’s inventory. One player could even try to perform a “steal” action and take an item from the other player’s inventory. Both of these options are useful game mechanics. Many games have the concept of a prestigious or special item. Capture the Flag is one example, where the flag is the special item. Players will likely want to see who has this item and try to take it from them.

There are many issues to think about when implementing this functionality. For example, we need to decide how this player-to-player bump would be triggered. Would a player tap on the other player’s icon when they are close enough? Could the bump automatically take place when the players are sufficiently close? We will also need to decide what the UI will be for the interaction itself, once it is trigged. In another light, we need to decide where in the editor the game designer will specify the game mechanics for the bump interaction. The designer needs to control which actions are available to the players during the interaction.

10.6 More Team Creation Options

10.6.1 Balancing Teams

Game designers could desire a certain level of parity among the different teams to ensure fairness. For example, the designer might want each team to be the same size. Or, he/she might require that each team has x players of each role. The designer might also want to prevent teams from having too many players of a given role.

There is currently no way for game designers to ensure such balance among the teams. When players join a multiplayer TaleBlazer game, they are given the option to select their team. Any player can select any team. In the future, we will need to allow the game designer to specify
constraints on the teams in the TaleBlazer editor. We will need to decide what constraints we will support and how the designer will input those constraints.

10.6.2 Dynamic Teams

The teams in a multiplayer game are preset by the game designer. There is currently no way for players to dynamically form a team in game.

This type of game mechanic is useful in many games. For example, consider a game that tries to simulate the relationship between ancient civilizations. In this game, a possible scenario is for a dissenting group to rebel from the government and form a new regime. Assuming the different regimes/governments are represented as teams, this would require the formation of a new team on the fly.

10.7 Timers in TaleBlazer

The notion of “time” is an important mechanic in games. For example, a game might only last for a specific length of time. Another possibility was discussed in section 6.2.3. Game designers might allow players to rejoin a game instance as a new player for a certain period of time. After that time, they no longer have that ability. One last interesting possibility involves what the gaming community often calls “overpowered abilities.” These are special abilities that have a high impact on the game. A game designer would naturally want to limit how often these abilities are used. Otherwise the game might become too easy.

All of these scenarios could be handled if TaleBlazer provided timers in the editor. In the blocks, game designers could start timers. When the timers expire or reach a certain point, they could trigger special game logic.

When we implement timers, one of the key issues we will have to handle is where the timers will run. Do they run on the multiplayer server or do they run on the TaleBlazer client? If they run on the server, how do you handle intermittent connectivity? A player might not even be connected to the server when a timer “goes off.”

As a final note, timers aren’t necessarily specific to multiplayer games. You could offer timers in single-player games as well. However, the implementation will likely be different. For
single-player, things are less complicated and we can simply run the timers locally on each device.

10.8 Team Independent Role

If a TaleBlazer game is used in a classroom, a teacher will likely want to moderate the game. The teacher will want to chat with both teams, identify struggling students, and intervene to keep students focused. The teacher might even want the ability to freeze/pause the entire game. He/she could also moderate the chat.

This suggests the need for a special moderator role. The moderator will have special abilities that other players do not have. This moderator role can apply even if a game is taking place casually among a group of kids. One kid might need to serve as the referee in the game. The referee can award points or break ties in the game. Kids will likely take turns being a referee.

This idea is still in its infancy and there are a number of interesting concerns. We cannot make this role just a traditional TaleBlazer role. This is because anyone can select a normal TaleBlazer role. We cannot have every player just picking this privileged role. Also, the moderator might want privileges that are not currently available in the game logic. There is no way to provide a role with the ability to arbitrarily pause and restart the game in the TaleBlazer editor. This could suggest that the moderator role should be more like a traditional Admin role in other software. The moderator could type in a special password to gain access to a list of hidden abilities.

Overall, this is a very interesting new feature idea, but it will have to be fledged out significantly. We will have to consult with various users in order to get their feedback.
Chapter 11

Contributions and Conclusion

11.1 Contributions

The contributions of this thesis include:

- Rebooting the TaleBlazer multiplayer platform after a year of dormancy
- Building a series of small-scale features to fix key usability issues and technical problems and also address underspecified behavior in parts of the system. These features get the multiplayer platform closer to production-level software. The features are as follows:
  - A way to quickly resume playing in a multiplayer game after exiting the game
  - A way for users to rejoin an instance as a new player, with a new role and team
  - Specifying how long multiplayer instances should last and implementing functionality for them to expire after this length of time
  - Additions to the in-game user interface
  - Defining the behavior of and implementing the “looks blocks” for multiplayer games
  - Defining the behavior of and implementing the “when game starts” block for multiplayer games
  - Preventing “dead instances” and “dead players” from being created
- The new chat game mechanic, which allows players in a multiplayer game to communicate over long-distances.
- The new proximity-constrained give action. This is a more realistic and powerful version of the previously implemented give action.
11.2 Conclusion

TaleBlazer is a platform for building and playing location-based augmented reality mobile games. The traditional version of TaleBlazer is single-player. The TaleBlazer multiplayer platform is much newer, but is rapidly growing. Multiplayer games can provide a richer gameplay experience than single-player games and can be very effective in the context of education.

This thesis built on previous work done for the multiplayer platform and presented new functionality. This functionality brings TaleBlazer multiplayer closer to production-level software. It also enables game designers to create more complex games and allows for richer in-game player interactions. Future work on multiplayer will involve bringing the software to full production-readiness and adding more game mechanics.
Bibliography

