The Development of Network Enabled Augmented Reality Mobile Applications

by

Owen Lin

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

Master of Engineering in Electrical Engineering and Computer Science

at the

MASSACHUSETTS INSTITUTE OF TECHNOLOGY

June 2010

Author . [illegible] . . . .
Department of Electrical Engineering and Computer Science
May 14, 2010

Certified by . . . . [illegible] . . . .
Eric Klopfer
Associate Professor
Thesis Supervisor

Accepted by . . . [illegible] . . . .
Christopher J. Terman
Chairman, Department Committee on Graduate Theses

The Development of Network Enabled Augmented Reality Mobile Applications

by

Owen Lin

Submitted to the Department of Electrical Engineering and Computer Science
on May 14, 2010, in partial fulfillment of the
requirements for the degree of
Master of Engineering in Electrical Engineering and Computer Science

Abstract

In this thesis, I designed, implemented, and evaluated network-enabled augmented reality mobile applications by extending an implementation of the MITAR iPhone client designed by the Scheller Teacher Education Program. In particular, I designed a multiplayer version of the client, which allows multiple users to interact with each other in a single game across multiple handsets and multiple platforms, and a data collection service that allows users to log media (such as pictures and text) throughout the duration of their game. The end result is an augmented reality client that fully takes advantage of the ubiquitous network connectivity offered by most modern mobile handsets.

Thesis Supervisor: Eric Klopfer
Title: Associate Professor

Acknowledgments

I would like to thank Professor Eric Klopfer for supporting me with his vision and direction for the MITAR project. Without his guidance, genius, and foresight, it would have been much more difficult to produce this thesis.

I would also like to thank Josh Sheldon and Judy Perry for being extraordinary project managers. I tremendously appreciate the time and energy that they have invested in me to make this thesis a reality.

Finally, I would like to thank my parents, Shiqi and Pei-Min, and my grandparents, Baoguang and Rose, for their unwavering support and encouragement. They have been a constant pillar of support that I can lean on.

Contents

List of Figures

Chapter 1

Introduction

Augmented reality is a term used to describe the merger of physical reality and virtual reality. Common examples of augmented reality are the heads up displays that planes use for navigational and combat aid and the yellow stripe that denote the "first-down line" in American football broadcasts. The Scheller Teacher's Education Program (STEP), under the leadership of Professor Eric Klopfer, has explored the use of augmented reality for creating interactive educational games. This thesis builds upon previous work done by STEP to produce a modern Augmented Reality client that takes advantage of the full capability of modern handsets. A recurring theme in this thesis is the use of the constant network connectivity available on these handsets to improve the educational experience for its users. Two features of this client that demonstrate the benefits of an ubiquitous network connection is the multiplayer game capability and data collection center.

1.1 Motivation

It is commonly believed that learning is greatly improved when one learns from a variety of sources–oral, visual, and textual[1]. This principle has long been known and commonly accepted in many academic circles, but in practice, many students are still primarily lectured at and expected to absorb information this way. The adoption of computers in education has improved the way education is done–now, in addition to text and pictures, students have access to audio, video, and a variety of other resources that augments a student's learning ability. The Scheller Teacher Education Program has taken the concept of multimodal learning and extended the idea further by designing several augmented reality games that allow students to interact and learn in a completely new dimension–through their environment. These augmented reality games help students learn by presenting virtual information about their whereabouts and allowing them to interact with their surroundings.

Augmented reality uses the physical world to augment the traditional media that a student may use to learn on a computer. A classic example of augmented reality in a noneducational context is the heads up display that fighter pilots have in their planes. As a plane flies, the computers onboard the aircraft gather data from a variety of sensors, processes this information, and overlays the result on the windshield of the plane. This method of augmented reality has been around for decades. The Teacher Education Program hopes to make this tool accessible for education: the goal of these mobile games is to give the users a more interactive way of learning, beyond simply reading about a subject in a book or being lectured about it.

1.2 Summary of Thesis Contents

- Chapter two will describe previous work done on augmented reality, particularly by the STEP lab.
- Chapter three will describe the design and implementation of the multiplayer functionality present in the client.
- Chapter four will describe the design and implementation of the data collection functionality present in the client.
- Chapter five will describe possible future work on this client and methods for extending the multiplayer functionality.
- Appendices A and B will describe the structure and public interfaces for multiplayer and data collection, respectively.

Chapter 2

Background

2.1 Related Work

The concept of augmented reality (and the related virtual reality) has been around for several decades. In the 1950s, a cinematographer named Morton Helig created one of the first examples of augmented reality technology. Helig believed that the movie experience could be much improved upon if more of the viewer's senses, not only vision and hearing, was involved in the process, so he built a machine called the Sensorama. The Sensorama was able to supply stereo sound, tilt the viewer's body, and simulate winds and smells[2]. Unfortunately, the Sensorama did not achieve commercial viability and success.

It would be several more decades before augmented reality as we know it today became more popular. In 1992, the phrase "Augmented Reality" was coined by Tom Caudell while he helped assembled planes at the Boeing Company. In 1993, Feiner, MacIntyre, and Seligmann of Columbia University publish their seminal paper on augmented reality in the Communication of the ACM[3]. In this paper, the authors described a system named KARMA, the Knowledge-based Augmented Reality for Maintenance Assistance. KARMA was a framework for creating augmented realities that helped explain maintenance and repair tasks by generating virtual overlays based on a user's current activities. By wearing a head mounted display, KARMA would overlay computer graphics over the image that the user was looking at. In 2000, Bruce H. Thomas developed the first outdoor augmented reality game, ARQuake[4]. In ARQuake, users would move about the physical world to play a first person shooter game generated by a computer. However, the users of the game had to carry a substantial amount of equipment in order to play this game, including a laptop, a separate GPS device, and an orientation sensor. This made ARQuake somewhat impractical for every day use. Finally, in 2008, Wikitude launched the AR Travel Guide for the Android G1 phone. AR Travel Guide allowed users to search Wikipedia for points of interest around their current location and overlay information about the point of interest over a camera view of the environment[5].

The research for this thesis takes many of these concepts one step further and applies them to education. Advances in mobile computing has truly made new, more practical forms of augmented reailty possible. Instead of carrying a computer and several other sensors like users would in ARQuake, a mobile smartphone such as the iPhone may be used as a substitute. Instead of preparing to play an augmented reality game hours before, setting up equipment and software, users may now simply download an augmented reailty game spontaneously through the internet connection on their mobile phones. Also unlike previous outdoor augmented reailty games like ARQuake, this research will focus on using augmented reality games for education, and for the first time, there will be a multiplayer component to these augmented reality games. This thesis will build upon this related work and the previous research done by the MIT Scheller Teacher Education program to create a new platform for network enabled augmented reality games.

2.2 MIT Schellar Teacher Education Program and MIT Augmented Reality (MITAR)

MIT's Schellar Teacher Education Program's first augmented reality game produced by the MITAR (MIT Augmented Reality) group was called Environmental Detectives, and was created in 2002. Using a mobile handheld device paired with a GPS system,

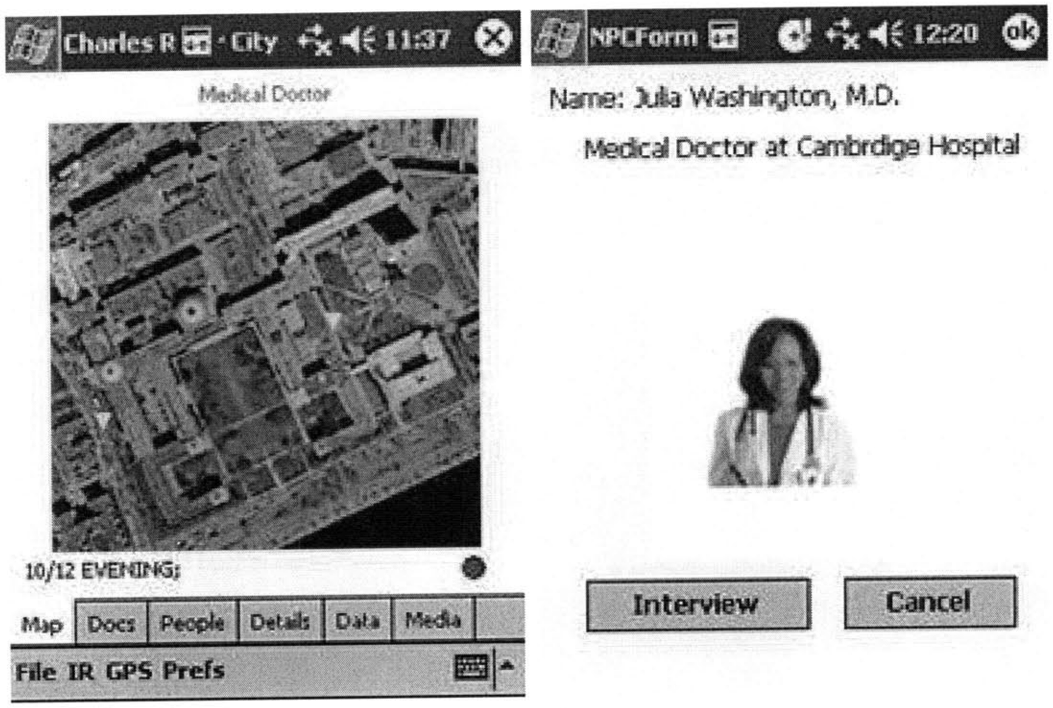


Figure 2-1: Sample screens from Charles River City on a Windows Mobile device

players would travel around the MIT campus, seeking clues about a toxic chemical. Players had the ability to interact virtually with their environment; for example, they could take water samples or speak virtually with Non Player Characters (NPCs). After the success of Environmental Detectives, the next game that STEP developed was Charles River City[6]. This game introduced the concept of roles. A team of players would play a game together, with each player having a different role. Different NPCs or actions would therefore be available to different roles, and it would take a coordinated team effort in order to solve the mystery of the game. This game also introduced triggers. With triggers, certain events can cause other events to occur. Player movement was tracked by a GPS unit attached to the device, and the game would respond to the user's change in location.

After the development of Charles River City, it became apparent that there was a need for a simple way to author augmented reality games without exposing the complexity of the proprietary game file format to the authors. Thus arose the split of the MITAR project into two components, the Game Editor and the Game Engine.

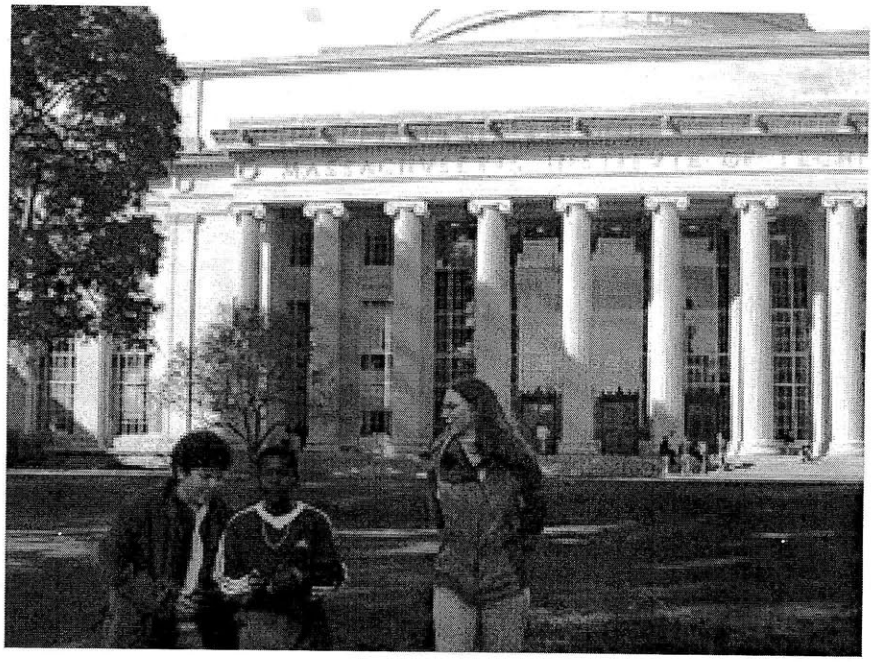


Figure 2-2: Users playing Charles River City on the MIT campus

The Game Editor is a simple, graphical method for users to create games that can be run on handhelds. Previously, games were created only by the members of STEP, but with the release of the Game Editor to the public, anyone could create an Augmented Reality game and run it on a MITAR client. In this sense, STEP's Augmented Reality games truly became a platform rather than a single instance of a novel idea. The Game Editor provided users with a graphical method of creating the logic behind AR Games, allowing users to drag and drop maps, non-player characters, triggers, etc. to be used during a game. The Game Editor automatically packages this user generated content into files then read by the Game Engine on the handheld devices.

The Game Engine is the underlying rules engine that runs the logic behind the augmented reality games on handheld devices. Given a set of input files, the Game Engine is able to translate it into a game playable on the handheld implementation. The Game Engine is further packaged into software that runs on a handheld device, called the client. The Game Engine began its life on Windows Mobile devices and has recently been extended to the iPhone, although there exists a certain disparity between the features accessible on each version. Although this thesis included a necessary understanding of the Game Editor, the primary focus of the work involved extending the Game Engine on a new platform for a new framework.

2.3 iPhone Implementation

With the advent of the iPhone 3G in the summer of 2008, STEP decided to implement a version of the platform specifically for the iPhone. Previous versions of the iPhone were not viable because of the lack of a dedicated GPS (as well as the lack of the ability to connect peripheral devices that provides the iPhone with GPS coordinates), but the iPhone 3G introduced users to an accurate assisted GPS chip that made the implementation of these games possible. Compared to the Windows Mobile devices that the games were previously run on, the iPhone provides the user with many more methods of interaction. For example, the iPhone allows the user to actually touch the screen with multiple fingers to interact with the device rather than simply with a stylus as was required by a Windows Mobile device. The iPhone also allows a user to interact with the device by changing the actual physical state of the device. For example, the user may shake the device to elicit a new response, or he or she may change the compass orientation of the device to produce some change. The iPhone has also achieved significant mainstream popularity as well and contains a novel method of delivering applications such as the Game Engine through the App Store. Any iPhone owner would be able to download and install the MITAR application simply by navigating to the MITAR application in the App Store and clicking the download button. The iPhone's popularity and the ease of distribution of apps on the device make it an ideal candidate for the initial iPhone version of the app.

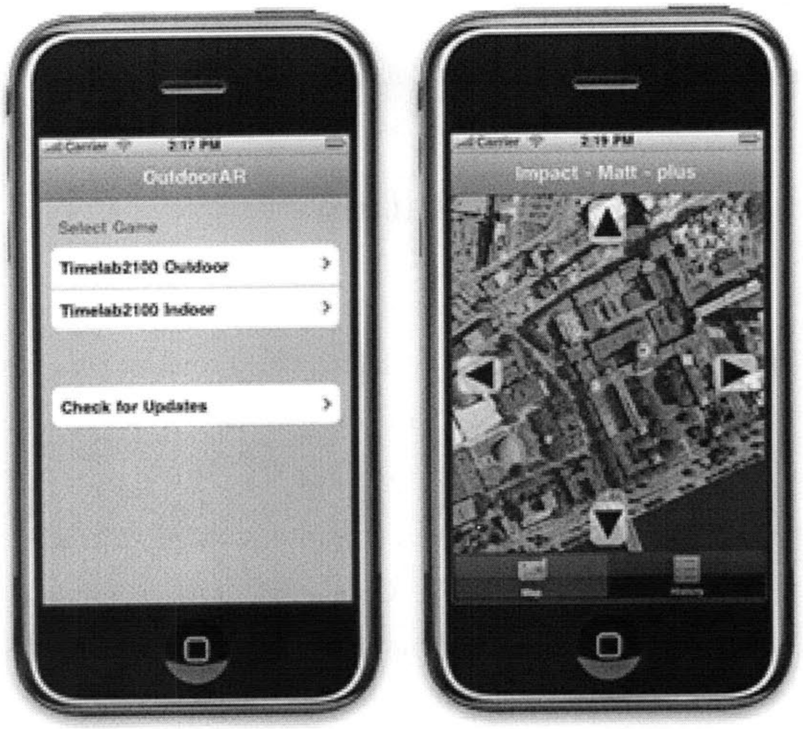
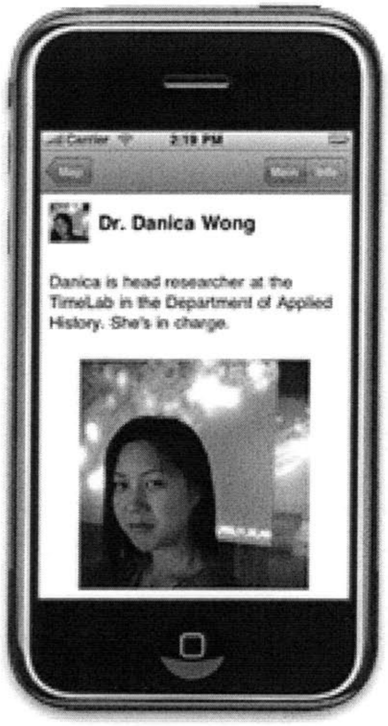


Figure 2-3: Starting a game on the first generation iPhone client

Under the direction of Tze Kwang Chin, my predecessor for this project, a prototypical iPhone application was developed in 2009[7]. This application is capable of the basic functionality present on the Windows Mobile device and can play many of the same games that the Windows Mobile client can. Games are distributed to the device through an online publishing mechanism; there is a central store where games for the client can be found, and users may download the game files simply through the interface. The basic architecture of this application is deeply discussed by Chin[7].



Dr. Danica Wong

Danica is head researcher at the TimeLab in the Department of Applied History. She's in charge.

Figure 2-4: Interacting with non player characters on the first generation iPhone client

One feature of the iPhone platform that was not greatly explored by previous implementations of the MITAR Client is the ubiquitous network connection available through the 3G and EDGE network chip. As a mobile phone, the iPhone's 3G chip allows it to send and receive data from cell towers. My thesis primarily explores the new features available from this feature: data collection and multiplayer.

Chapter 3

Multiplayer

3.1 Motivation

The ability to communicate with the Internet is a significant new feature of this new generation of mobile phones. My thesis attempts to take advantage of this new feature to create a framework for multiplayer functionality for the MITAR client. This multiplayer feature will allow users to create and play multiplayer versions of the MITAR games. This multiplayer functionality will initially target the iPhone client as the client device, but ultimately, it will be able to support multiple clients running multiple operating systems simultaneously playing the same multiplayer game. Thus, the objectives of the multiplayer component of my thesis are to create an operating system agnostic server for storing and tracking multiplayer game state, to create an iPhone implementation to demonstrate the server's capabilities, and to create a simple and portable framework for implementing new multiplayer features.

This multiplayer ability is compelling because it gives users a completely new way of interacting with each other and the game, and it forces the user to think more about the decisions that they make throughout the game. With a multiplayer framework available, it will be possible for game designers to force users to interact with each other. This thesis describes the design and implementation of singleton objects and visual puzzles, but other interesting features that may be implemented within this framework are network votes and special abilities. For example, the game designer may like the users to vote on a particular subject, and the outcome of the vote will determine a certain path in the game. A multiplayer game also forces the students who play the game to consider more carefully the consequences of their actions because it affects the other players' games. This feature also aims to increase interest in the games that the users play because it allows them to interact through the virtual world with their classmates.

In the following section, I will describe the initial design decisions, the actual implementations, and future work for this multiplayer feature.

3.2 Design

There are three main components to the design of the multiplayer system: the game format, the server, and the client. A significant obstacle to the creation of a robust multiplayer augmented reality game is that each mobile device does not have a unique address. Because a user is constantly moving across the map, the client's address changes as the user's device connects with different cell towers for network access. A consequence of this is that a central server may not consistently send data to a client because it does not necessarily know the current address of the client. Instead, the client must send requests to the server, and only then can the server respond with new data. A large effort in this thesis is spent on overcoming this problem. Throughout this description, I will refer often to the "client" and the "server". In this case, the client refers to any mobile handset that runs the MITAR program, and the server refers to the central communication hub that I will describe below. When I refer to an "instance of a multiplayer game", I am referring to a singular copy of the multiplayer game that is initialized, with users who are able to interact with each other through the game interface in some way.

3.2.1 Game Format

In order to understand the design requirements for the server and client, we must first understand what a multiplayer game entails.

The simple definition of multiplayer game that I will use here is this: when playing a multiplayer game, the state of one player's game on one client can by affected by the actions of another player on another client, and vice versa.

The foundation of multiplayer games on the client is built on top of one base multiplayer object class, named ArMultiplayerObject on the iPhone. The naming convention will be similar on other platforms, but for the sake of narrative clarity, I will refer to this ArMulitplayerObject class simply as "the multiplayer object". This multiplayer object is different from all other objects in a MITAR game in that there may only be one instance of this object in play across the entire multiplayer game. That is, if this multiplayer object encounters a change of state on one player's client, this change is visible on all other users' handhelds–it is a singleton instance. Much of the multiplayer component focuses on designing a server and client architecture that makes this multiplayer object possible and persistent across several multiplayer game instances. From this multiplayer object base class, we may easily implement several other examples of multiplayer functionality by extending this class. For this thesis, we will focus on the design of this multiplayer singleton object and a proof of concept extension, the visual puzzle.

Singleton Objects

The ArMultiplayerObject class represents a singleton object in a multiplayer game in the sense that there is only one copy of this object per instance of a game. In the previous implementation of the MITAR client, there exists a class for objects that players may interact with called the ArTemplatedObject. Multiplayer singleton objects, represented by the ArMultiplayerObject class, extend ArTemplatedObject. In addition to keeping track of the data for the object that users may interact with, the ArMultiplayerObject also manages the state of the object and makes sure that the state of the object is consistent across the game. Thus, whenever a user interacts with a multiplayer object, the multiplayer object first sends a request to the centralized server (which is described below) to check if the player is allowed to interact with the object. This is necessary because each client is not in constant contact with the central server. The object's state may have actually changed since the client was last updated. If the object is unavailable for the client to use because its state has changed, the client alerts the user. If the user may interact with the multiplayer object and he or she decides to change the state of the object, the multiplayer object will then send a request to the server indicating the changes. In this way, multiplayer objects maintain consistent state across multiple instances of the game on multiple clients.

Visual Puzzles

Visual puzzles are a new type of virtual object that players can interact with. Each player in a game may pick up a unique portion of an image. When several players in the game have picked up these image portions, they can hold their handhelds together and physically recreate the original image.

Inventories

To support the aforementioned multiplayer features, each multiplayer client will have an inventory. This inventory is simply a list of items currently held by the player. Items may be dropped from or added to the inventory.

3.2.2 Game Server

The multiplayer game uses a central server to model and direct a multiplayer game in progress. The alternative to having a central server is to have the game clients communicate amongst each other. However, because the client devices will frequently have different addresses, the multiplayer component needs some central data structure with a known address so that robust communication is ensured. For this reason, we chose the central server architecture rather than a peer to peer communication model.

The game server is the central location that clients communicate with to play a multiplayer game. The server's tasks are to:

1. Initialize multiplayer games when a host creates a new game request

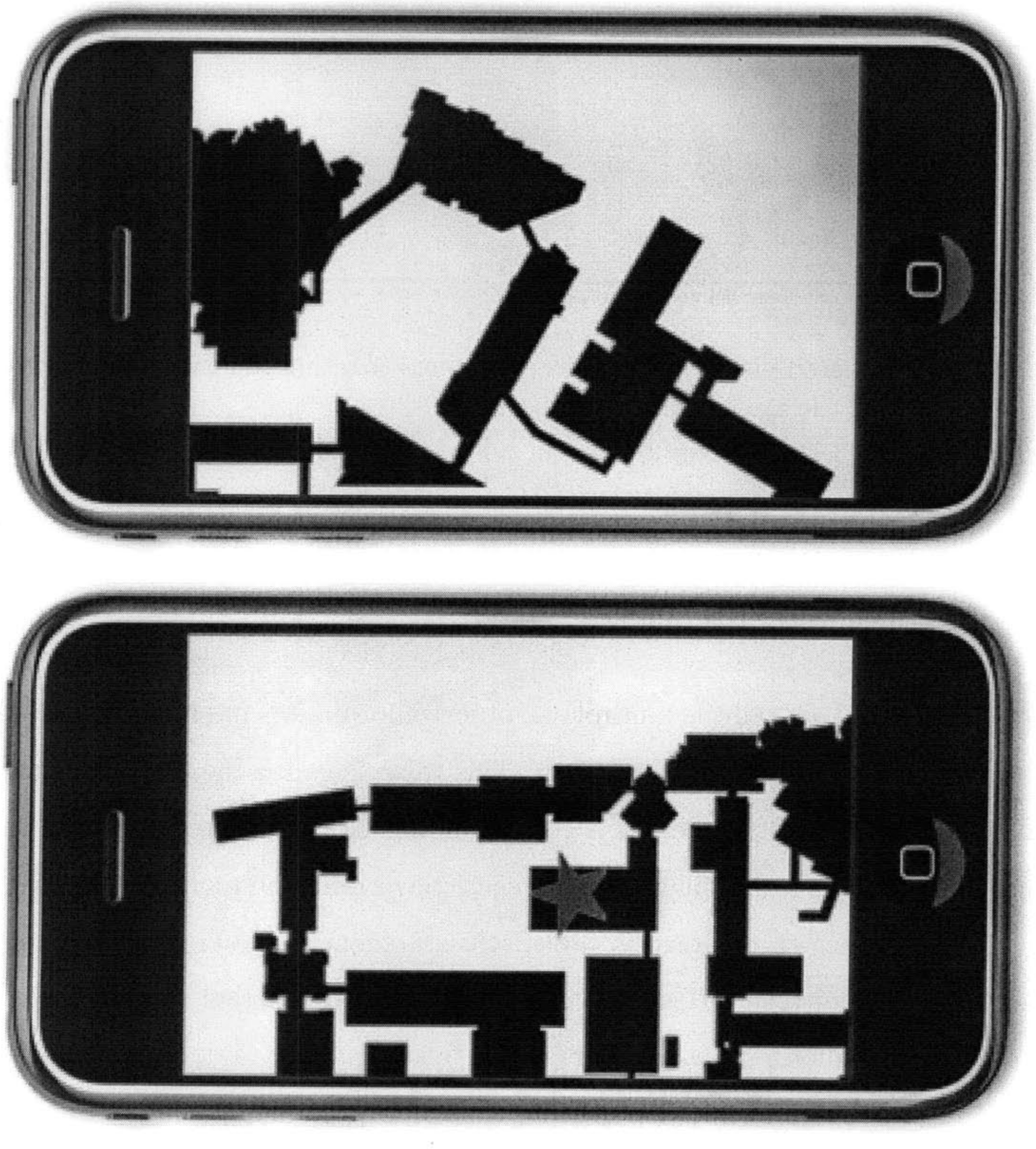


Figure 3-1: Individual screens from a visual puzzle

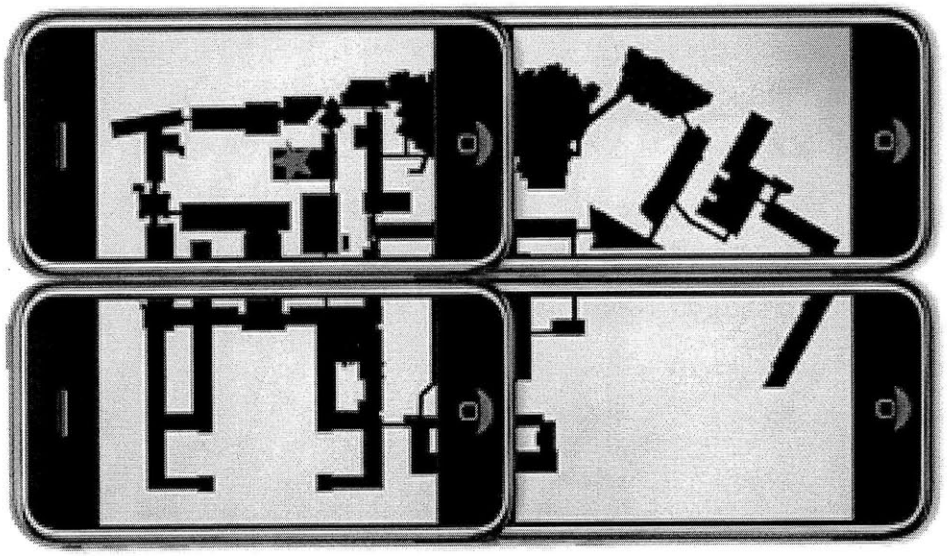


Figure 3-2: Completed visual puzzle; pieces are held together to form the underground map for MIT and the next goal for the users

2. Maintain the state of the game while it is being played and

3. End a game when it is complete.

The game state, i.e. all the multiplayer object information and metadata, is stored on a table in the database on the server. This table is called the Object Ownership table. Each multiplayer game has a corresponding Object Ownership table on the server. This table is a mapping between each player and the multiplayer objects in the game. In order to create this table, when a client requests a new game from the server, the client tells the server what multiplayer objects that should exist in the game. Using this information, the user creates the table. This table stores information on which user has a multiplayer object in his or her inventory. Each game also has a corresponding metafile, which stores the game state. This will be described in detail later in this chapter.

Hosting and Joining a Game

Because of the nature of the multiplayer game, there must be a host that initially requests a new game from the server. The host will request the server to create an instance of the game and allow other players to join these games. In order to make it easier for users to play a game, any player on a client may host a game. Previously, the plan was to have an "administrator" account, who is not a player of the game, whose role was to create and manage the multiplayer sessions. In the context of the classroom, this administrator would correspond to the teacher. This would also conceivably reduce the amount of mistakes, such as two players who intend to join the same game creating multiple game sessions. However, it was decided that having one special user who must perform all the initialization tasks was not only detrimental to the user experience, but also reduces the simplicity of the design. In the classroom setting, this one administrative user must set up each game that will be played, and the actual players of the game have no control over when they can start a game. Also, in order to implement a special administrator user, there must be an extra layer of authentication above the existing logon structure. For these reasons, the multiplayer component implemented a model where any user can host a new game.

When a player asks to host a game, the client sends a request to the server to initialize a new instance of the game. The server then generates a unique identifier for the game, creates the appropriate metafiles and tables within the database, and returns the unique identifier of the game to the client. Currently, the unique identifier is created by a concatenation of the name of the game, the username of the host, and a four digit integer that corresponds to the last four digits of the server time. Thus, for a user 'owenlin' playing a multiplayer game "TimeLab 2100", a unique identifier can be "TimeLab_2100_owenlin_9340". The server has a metafile that keeps track of open games, and this game is appended to the metafile. Now, other users playing TimeLab 2100 will be able to see this game instance available.

When another user chooses to play the multiplayer version of the TimeLab game, the client sends a request to the server to list the open TimeLab games. The server returns the formatted contents of the open games metafile, and the client displays this information to the user. If the user decides to join a game, he first logs on and then sends a request to the server to join the game. This log on is not authenticated; that is, the user may enter any user name that he wishes. Because this feature is expected to be used within a moderated classroom environment, we believe that this would be a good design decision. The user then enters a wait room. From there, he or she may choose a role for the game, and he or she can see the other players that join the game. The players remain in the wait room until the host decides to start the game. When the host decides to start the game, the server sends a message to all the clients in the game that the game is to start, and then all the clients begin their games.

Game In Progress

When a game is in progress, the server's task is to keep track of the state of the game and update all clients in the game with state changes when there is a multiplayer action taken. We will consider the pickup of a singleton object as an example of a multiplayer action.

In addition to the Object Ownership table, the server also has another data structure, an indexed queue, representing the history of actions taken during the game. This history queue is initialized to 0. When a new action is taken, a new action is added to this queue with the index incremented by one. Each client that plays the game also keeps track of this history. Both the server and the client's history index begins at 0. This history queue is stored as a separate metafile on the server.

When a player picks up a singleton object, the client sends a message to the server alerting it of this action. The server updates the table by setting the cell corresponding to the item and player to TRUE. This means that the player currently possesses the item. The client and the server both updates the history queue with the new action and increments the history index.

At this point, an action has been performed by a client, the server's table has been updated, the action has been added to the history, but not all the clients have been updated. Some clients have a stale state. The server must update the clients with the new game state. Each client periodically pings the server with its history counter. An important invariant is this: a client's history index will always be equal or less than the server's history index. If the two history indices are equal, then the client's game is currently updated, but if the two history indices are unequal, then the client's state is stale. If the server notices that the client's state is stale, it will send the client all the actions that have been taken since the client's history index. The client will then update its state with these actions.

This maintains persistent state across all clients of an instance of a multiplayer game, and all multiplayer actions are based on this procedure.

3.2.3 Client

The previous MITAR iPhone client was substantially modified in order to make multiplayer games possible. The client must be able to

- Log in and join a game,
- Create an inventory to model singleton objects held by a player, and
- Performing actions in the messages sent by the server.

Logging in and Joining a Game

When a player opens the game, he or she sees a list of games that are currently joinable. When a user clicks on one of these games, the client will enter a Wait Room, which, like previously described, will allow the user to see what other players are joining the game and choose a role. The player may also choose to host a new game.

Inventory

Because players can now pick up items during the game, the clients must implement a client that reflects the items that a player picks up. This inventory will be implemented simply as a list, and the players will be able to bring this list up. On the iPhone client, this list is simply implemented as another tab that the user can switch to.

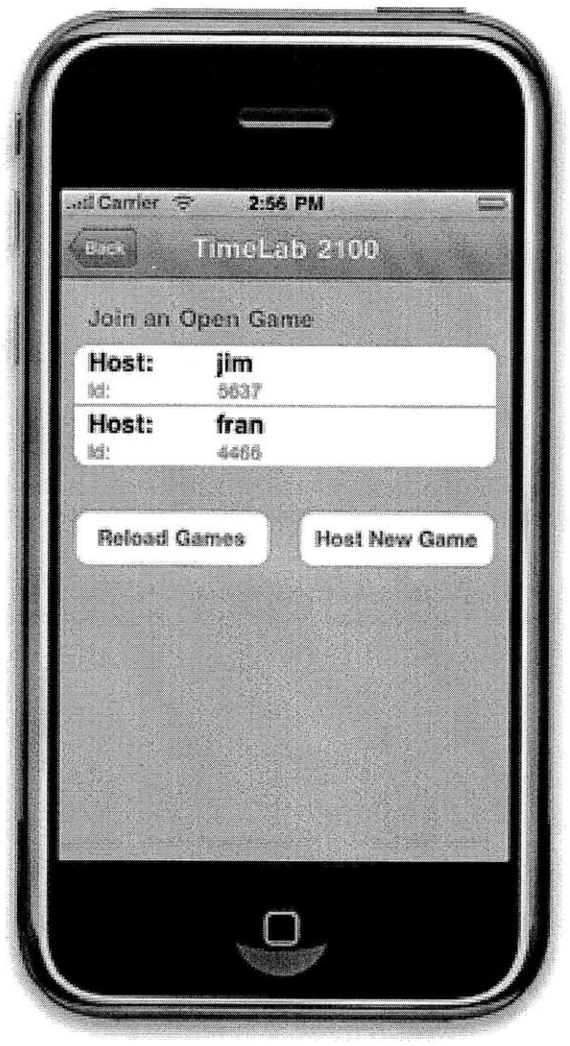
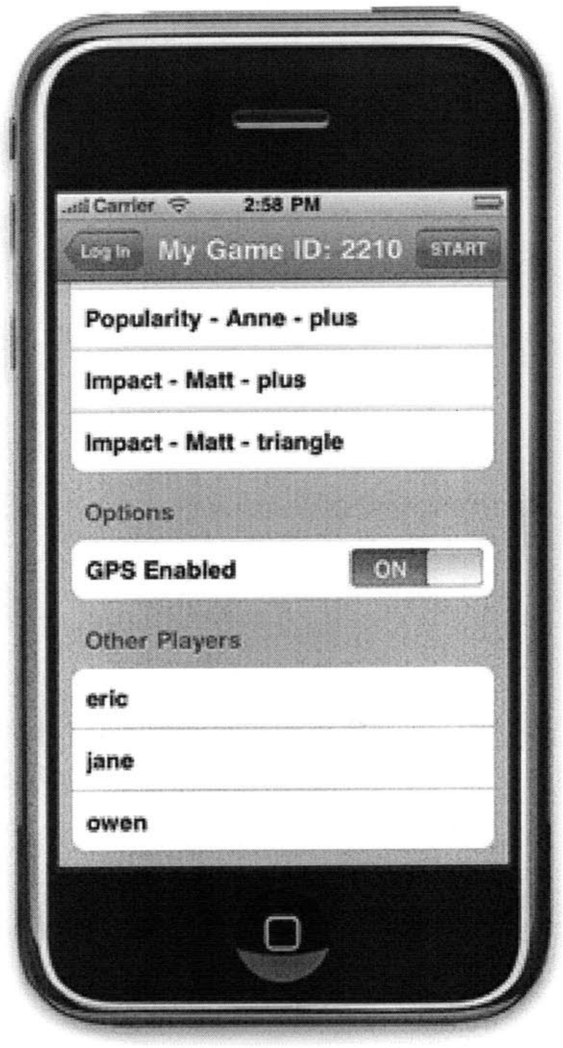


Figure 3-3: Upon choosing a multiplayer game, the user is given the option of joining a pre-existing game or hosting a new game



Popularity - Anne - plus

Impact - Matt - plus

Impact - Matt - triangle

Options

GPS Enabled ON

Other Players

eric

jane

owen

Figure 3-4: The user then enters a wait room. The Start button is visible only for the host of the game, who may press this button to start the game across all devices playing this game instance

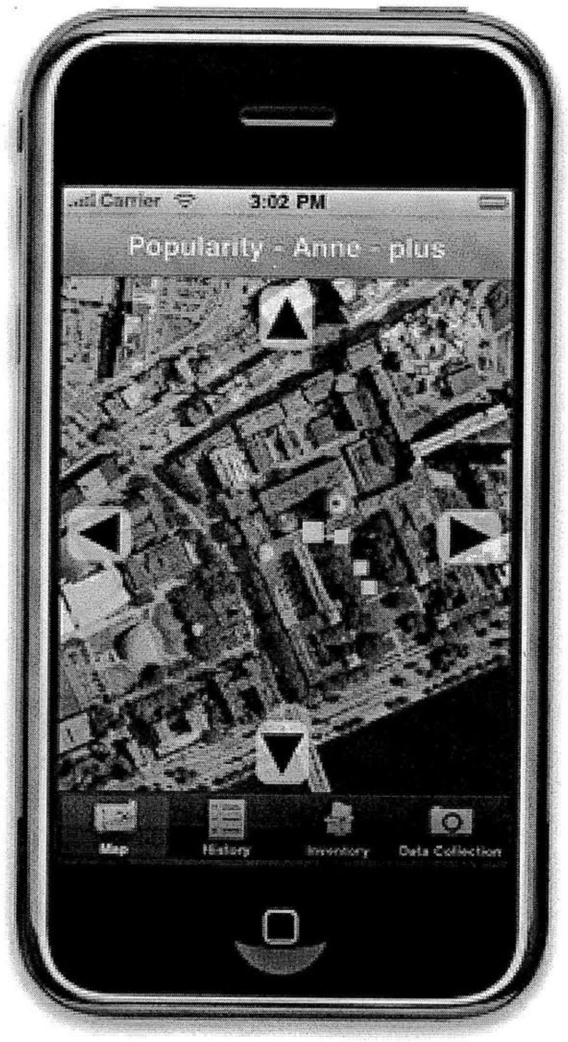


Figure 3-5: The screen of the handheld during a multiplayer game. The larger yellow square indicates that the user is about to encounter an object in the game.

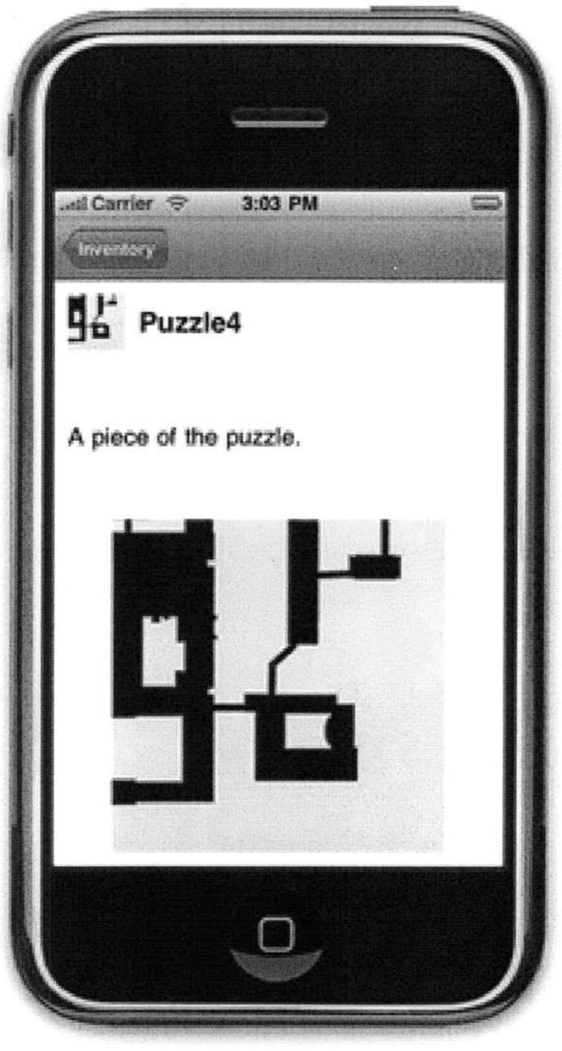


Figure 3-6: Once a user has picked up a multiplayer object, it resides in his or her inventory, where it may be viewed again later.

Changing Multiplayer State

Game state on the client is modeled with the client's history queue, as described in the previous sections. Periodically, the client sends a message to the server with its current history index. If the client's history index does not match the server's history index, then the client does not the have the most updated state. The server will send all the state changes since the client's history index to the client. The client will then execute these state changes.

Currently, there is one type of state change, the ITEM message. The ITEM message represents a singleton object that has been picked up by another player. The ITEM message contains the ID of the item, the player that initializes this action, and the type of action (either PICK-UP or DROP). When a client receives an ITEM message, it changes the visibility of that item according to the action type. The item will disappear from the player's screen, simulating an item being picked up by another player. Multiplayer changes are executed by this combination of ITEM messages and history queues.

3.3 Game Editor

Eventually, there will need to be several modifications to the Game Editor to support the multiplayer functionality. Because the Game Editor creates the XML files that all clients read to create a game, the Game Editor must allow the creation of multiplayer objects and the storing of the new games in a format that each client may parse.

3.3.1 Singleton Objects

Before the multiplayer functionality was available, all objects in MITAR games that allowed interaction were Templated Objects. In order to make multiplayer games possible, the editor must allow the game designer to create Multiplayer Objects, which are subclasses of Templated Objects, and write this change to the resulting XML file. The clients then parse this XML file and create a multiplayer object for each of these elements that it encounters.

Chapter 4

Data Collection

4.1 Goals

The goal of data collection on the MITAR client is to allow users users to collect data, including text inputs, pictures, audio, video, and other types of media, and have this information logged for constant storage in a central location. MITAR users may later review the information that was collected during gameplay, or download this data for statistical analysis. An example of this concept can be seen from the gameplay of Environmental Detectives, the first MITAR game that was designed and deployed by MITAR, in which students would travel around the MIT campus looking for clues to a mysterious chemical that is found nearby and taking virtual samples and measurements. With the implementation of data collection from this thesis, students are able to take this data, have this data upload transparently and automatically, review this data later, and use this real data for comparison. Another feature of this implementation is that these updates can actually be viewed in real time, which leads to the interesting possibility of an administrator or teacher sitting in a room and reviewing the actions and progress of a classroom playing a game through a computer.

This ability to track the user's actions in real time is extremely compelling for educational reasons. First, the teacher and student are able to see what data led a student to a conclusion. Previously, in an environment where there are many students, it is difficult for the teacher to follow each student's paths when the students are out playing a game. Now, the students may play the game and the teachers can check the students' progress at their leisure from the MITAR Command Center, the web application designed to review the data collected. Second, teachers can now review the results of student's gameplay and determine how the students reacted to playing the game and fix any errors they saw in their students' reasoning.

4.2 Design

Two important design goals for this implementation of data collection on MITAR was for the data collection to be both transparent to the user and portable to different platforms. By making the data collection transparent to the user, the user should not notice that data collection and communication to a server is taking place. For example, when a user takes a picture from his device, there should not be a pause between when he takes the picture and when he can resume playing the game because of the data being uploaded to a server. In short, gameplay should not be negatively affected by data collection. By making data collection portable, different clients (such as the versions of the MITAR client running on Windows Mobile and Android) should all be able to implement their own version of data collection and interact with the server. This takes some consideration when designing the architecture of the server–the communication between the server and each client should be a simple protocol that can be easily implemented on different platforms.

4.2.1 Game Partitioning/Granularity

An important question to address before designing this feature is how the game will be partitioned and how granular each game will be. For a game with four roles, will each instance of a game consist of just four users, one for each role in the game? Or would each game consist of several teams of four players working concurrently within this one game instance? There are arguments for either choice. The former option is the simplest to implement because the latter would introduce a new granularity layer to each game; the idea of teams. The second idea would be more consistent with real world usages of the app. Presumably, a teacher in a classroom setting would form a class of 20 to 30 children into teams of four, and each team would play this game concurrently. However, in the end, the first option's ease of use won out. Instead of setting teams manually, a process that could take a significant amount of time, a teacher could simply create a different instance of the game for each team and organize it this way. This decision was important for the way that the server is structured and the implementation of the client.

Server Architecture

The central server is the integral piece of this data collection feature. Dubbed the "MITAR Command Center," or "MITARcc" for short, the command center handles the upload and download of data. MITARcc runs on top of the Apache web server and is predominantly written in Python. The design of the command center went through several iterations.

First Iteration

In the first iteration, there were three main components to the data collection. The first component was the uploader. The second component was the user interface to the command center. The third component was the authentication. The uploader was composed mostly of several python scripts that would simply take upload requests through the HTTP POST/GET message requests. The user would supply information about his id, latitude, longitude, datatype, date, and associated file, and the upload scripts would format and save the file and append this metadata to a special file. Each instance of a game has its own file to store metadata. When a teacher or administrator enters the web address for the MITARcc, he is presented with a login page and a request to enter a password. Once the user logs in, he or she is able to create new data collection game instances, view completed game instances and games that are currently being played on a map in realtime, and download data that was already collected. The administrator or teacher is also able to automatically generate username and passwords for students, who would then use this to log into a game.

Authentication was fairly straightforward; there is a password file which contains a mapping between a username and the MD5 hash of the user's password. When a user logs in, the command center simply checks if the MD5 hash of the entered password matches the MD5 hash in the password file. When users upload data to the server, the server again does a simple authentication; it simply checks the game name that the data originated from, and then it finds a mapping of the username for that game to the metadata file. This file is then used to store the metadata for the data uploaded.

Second Iteration

Although this architecture worked at first, it soon became apparent that improvements needed to be made. There were several important issues: scalability, complexity, consistency. The second iteration of the MITARcc addressed these issues while maintaining the same public interface for clients. First, the fact that all the metadata from a data upload was written to a file was a bottleneck. While the metadata was being written, there is a lock on the file, and if another upload attempts to take place simultaneously, the second upload must wait for the first upload to finish before it can write to the metadata file. The behavior of the users in the game suggests that these simultaneous uploads occur fairly often. Usually, some event in the game prompts the users of the game to upload some data. Thus, when several players are playing a game together, the data collection is highly clustered. For example, four players in a game might visit a Non Player Character that asks the players to take a picture of their surroundings. As these four players attempt to take the photo and upload it to the server, they will all attempt to access the metadata file at the same time. Additionally, the probability that there is a collision increases with the number of players in the game. The second iteration of the server fixed this problem by migrating the metadata storage to a SQL database. This removes the performance bottleneck from the previous iteration. When a player decides to create an instance of a game that collects data, the server will generate a new ID for the game if necessary and create a table in the database to hold the metadata for this game. The ID generation will be discussed more in the following sections. The following table shows the metadata that is stored by each table:

uname	username
stud_group	student group (currently unused)
lat	latitude of data collected
lon	longitude of data collected
ftype	filetype
path	filepath of the uploaded file (if data is multimedia file)
title	title of data collected (if data is text)
summary	body of data collected (if data is text)
data	an automatically generated timestamp of the data collected

Note that for media files, the binary data itself is not stored in the database. Rather, the URL for the file is stored. From this URL, a client may download the actual media file. For pictures, the client may also use this URL to find the location of a thumbnail of the data.

A second issue addressed by this iteration of the data collection server is the problem with complexity, especially within the web interface. In the first iteration, an administrator or a teacher must log into the MITARcc to create a game, and then users could join the corresponding game through the client. An administrator or teacher was required for each instance when data collection is used. This greatly hindered the process of data collection; a casual user would have to go through many steps in order to start a game with data collection, and the use of the web interface was required. This is contrary to the principle of transparency with the data collection server. To remove this obstacle, it was decided to remove the extra layer of administrator/teacher from the MITARcc. There is no longer an administrator/teacher with special access abilities; instead, any user on a client may start a game with data collection. This greatly simplifies the design of the server because the server no longer needs to provide authentication for administrators/teachers from the server or authentication when user joins a game because that functionality is handled by the multiplayer component to the server. A result of this is that the web interface for the

MITARcc is also greatly simplified.

Users may only use the new web interface to read data; web users no longer have the ability to modify data or start games. On the web interface, the user can choose a game and then a game instance. Once the user chooses a game instance, the game map automatically zooms to a region where all the data collected in the game instance is visible. The MITARcc uses Google Maps technology to display information. Data collected is displayed as a pin. Clicking on the pin opens an overlay window, showing a thumbnail of the media or the collected text if the data is text. Clicking on the thumbnail opens a new window with the full sized media. The user may download this data if he would like.

The third issue addressed by this iteration of the data collection server is consistency with the multiplayer server. The multiplayer component and the first iteration of the data collection component were not connected and many of the components were not reused. The second iteration of the data collection server solved this problem. Now, the data collection and multiplayer games use the same method to generate new games and create new tables in the database. In fact, the two components can be seen as modular components to the same basic framework for building more multiplayer features into the MITAR client.

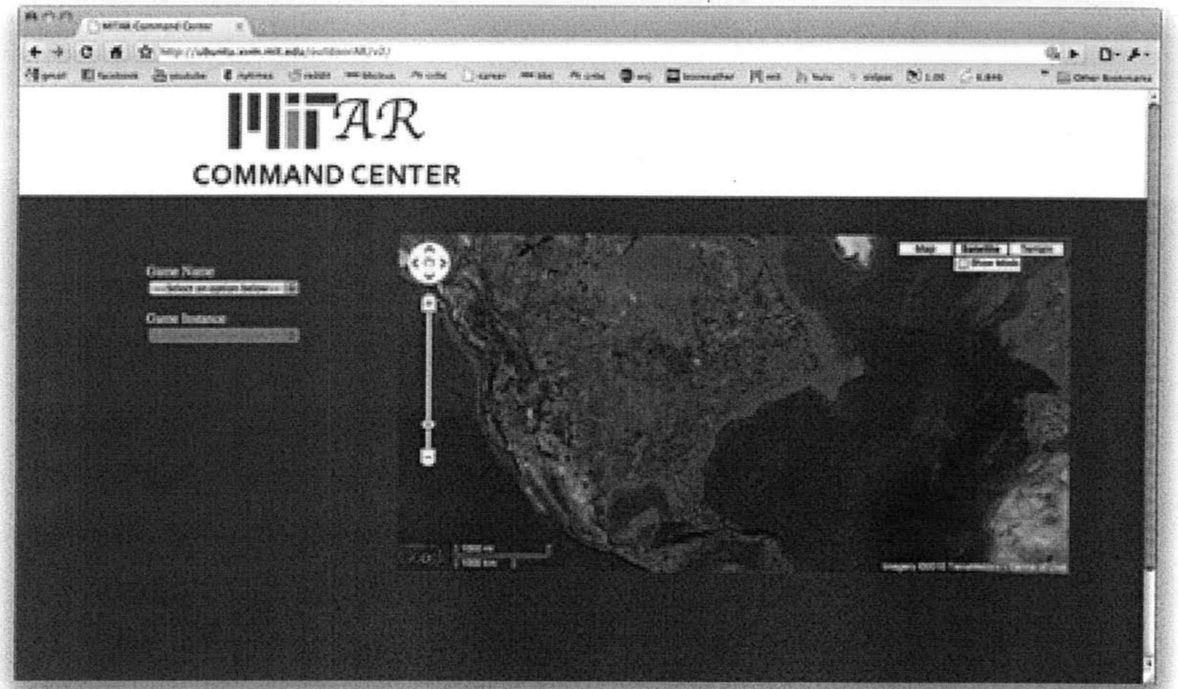


Figure 4-1: The initial view for the MITAR Command Center. The user is presented with an interactive map, and a simple choice for selecting games to view

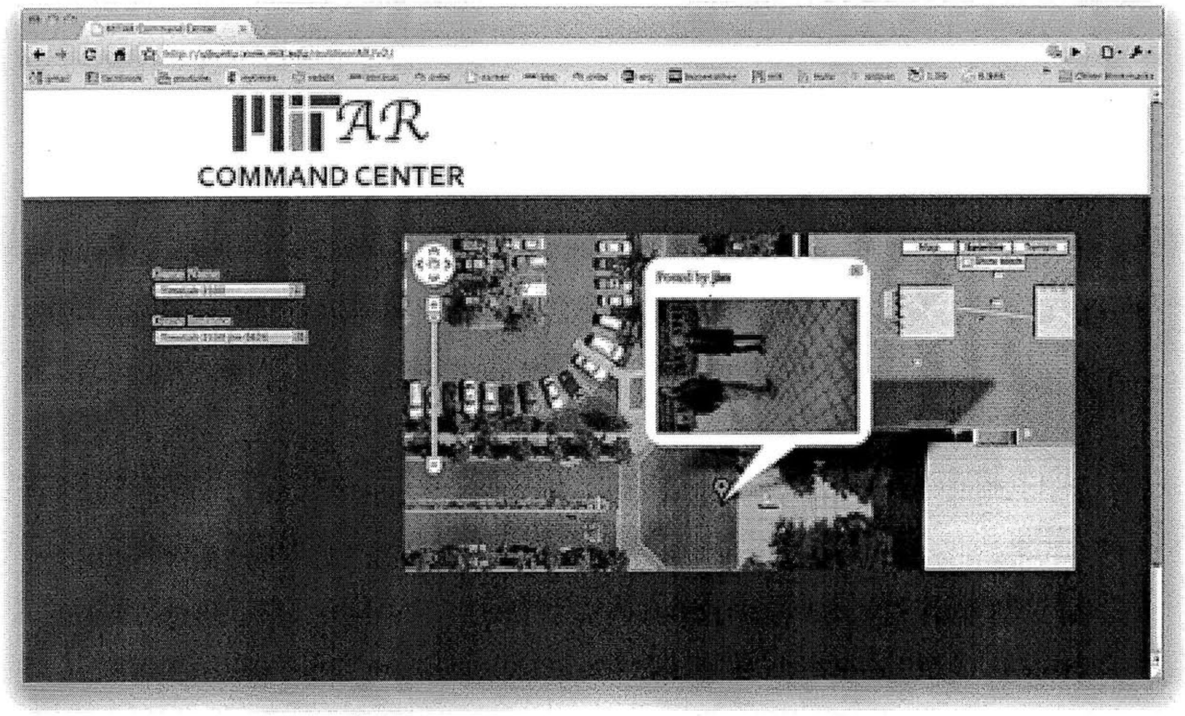


Figure 4-2: Examining a picture that was collected on the MITAR Command Center. Data on the map is represented by the pins. When the user clicks on a pin, a thumbnail of the data is shown, as in this figure.

4.2.2 MITAR Command Center

After the second iteration, the MITAR Command Center has a much simplified design, and the web interface is now written solely with AJAX, Javascript, and HTML rather than Python and HTML. There are three main components to the MITARcc: the interface (composed mainly of HTML, Javascript, and CSS), the database communication layer, and the data collection object model.

The MITARcc interface uses the Google Maps API to present the user with an interactive map that shows the location of the data that has been collected. When the user first selects a game and then a game name, MITARcc sends a request to the data collection server for the data in the game. This request is handled by the database communication layer. An AJAX request is sent to the server, which returns a string formatted with the data collected and the metadata for this collection (such as location taken, the name of the user who collected this data, etc.). Appendix B contains information regarding the formatting of this string and the scripts on the data collection server that are called. Then, this string of data is passed to the data collection object model, which uses Javascript to parse the string and create a ArDataObject from it. An ArDataObject is a Javascript object associated with each piece of data collected. The ArDataObject also contains a Google Maps Marker object, which displays information on the map when it is clicked upon. Lastly, this ArDataObject is added dynamically added to the map in the interface, and the user may click the Marker object to view more information on the object and download the file.

4.2.3 Extending Server Data Types

With the previous server data collection table architecture, it is relatively easy to extend the data types collected by the server. The column 'ftype' in the previous table is simply an enumeration of the data types that are accepted by the data collection server. In order to extend a new data collected type to the server, one must:

1. Define a new enumeration for the filetype. Currently, ‘p’ is used for images and ‘t’ is used for text.

2. Create a new public interface for the new filetype upload. This can be modelled from the upload_pic.py script described in Appendix B.

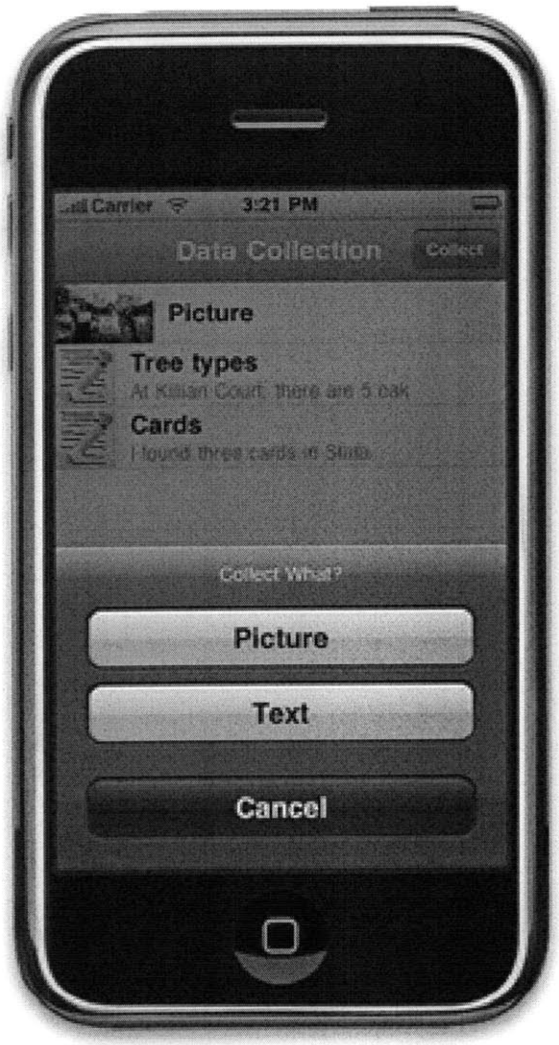
3. In the command center parser, check for the new file type enumeration and create a handler for it. Currently, nothing is done if the parser finds a filetype that it does not recognize.

Once the server has been extended to accept the new filetype, the client may upload the new file type simply by communicating with the public interface defined by step 2.

4.3 Client Architecture

The data collection server has been implemented on the iPhone MITAR client. Data collection on the client follows the model-view-controller design pattern. The view is a series of screens on the iPhone that allows the user to take data and display previously captured data. The controller is a series of classes (one for each media type) that handles the upload of the corresponding data to the server. Finally, the data collected is stored under a new class, ArData, which holds the corresponding data fields.

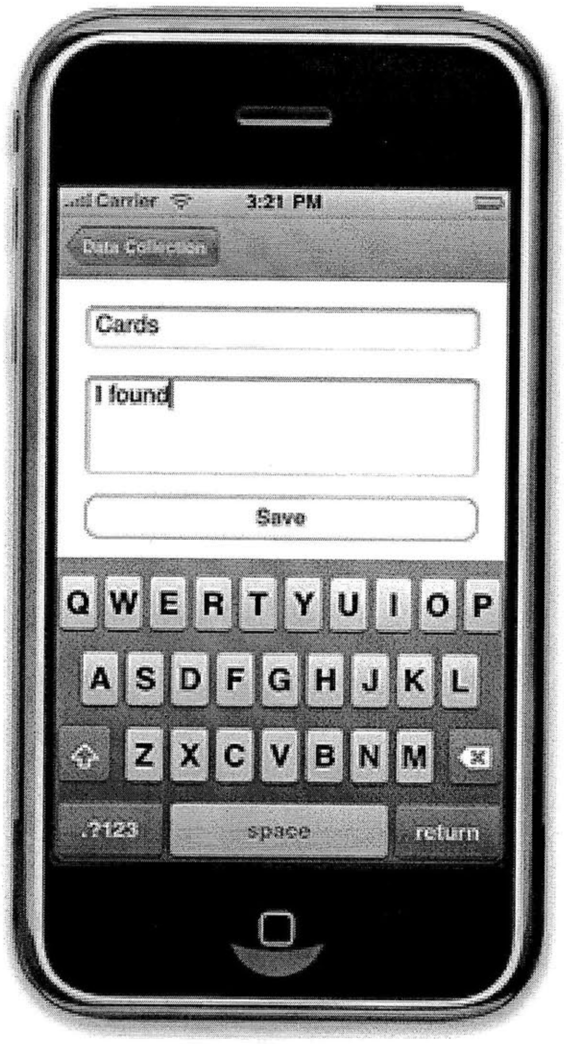
Although the data collection has only been implemented on the iPhone client, it is simply an exercise of porting the code from Objective-C to a different programming language in order to get the data collection server on another client, since POST requests are standard to any internet device.



At Killian Court, there are 5 oak

I found three cards in Stata

Figure 4-3: The user is presented with the type of data they would like to collect. Currently, only text and pictures are available. Note in the background that there are several picture and text data that has already been collected



Cards

I found

Figure 4-4: Collection of text data

Chapter 5

Conclusion and Future Work

In this thesis, I took advantage of the recent technological breakthroughs in mobile computing to add a category of completely new features and user interaction to augmented reality games on the MITAR platform. In particular, I created a simple framework for developers to take advantage of the internet capability of mobile devices like the iPhone, and built a platform for server side communications with the mobile devices to be deployed. The multiplayer implementations of the MITAR client and the data collection feature are two examples of this new framework.

Now that this thesis is complete, it is possible to implement several different features on top of the multiplayer and data collection frameworks. For example, with the multiplayer framework, currently only the singleton object and visual puzzle proofs of concept are implemented. Using the singleton object as a reference, it is possible to create several other multiplayer features. For example, it is possible to create multiplayer votes. A non player character within a game may prompt the players to vote on a choice of actions, and the outcome of the vote will affect the outcome of the game. Special abilities may also be implemented. An example of a special ability is if a player picks up a key object and every player in the multiplayer game will be able to enter a new part of the playing map. Both of these features may be simply implemented by simply extending the multiplayer framework. The data collection framework may also be expanded in the near future. MITAR is evaluating the use of a centralized server to publish games, linking every client to the server. With the inclusion of authentication and authorization, the MITARcc may be expanded to this role, although this will be the topic of another thesis. Finally, the types of statistics and data collected by MITARcc may easily be extended by augmenting the list of acceptable filetypes as described in Chapter 4. Also, one of the main goals of this thesis was to make the multiplayer and data collection features as portable and transparent as possible. We believe we have succeeded. In the future, it will be possible to implement these exact same features on other mobile platforms such as Windows Mobile and Android simply by using the protocols defined by the frameworks.

Appendix A

Multiplayer Server and Database Interface

As described in Chapter 3, multiplayer state is maintained through the Object Ownership table and the associated metafiles. The MITAR clients interact with the multiplayer server through several public interfaces visible as python CGI scripts. This chapter will describe these interfaces.

action_update.py

Inputs player name, game name, action, item name

Description This script is called whenever a multiplayer object's state is changed (i.e., it is picked up or dropped). The state change is saved as a transition in the form a tuple (player, get/lose, object_name)

Output "true" if action is successfully performed, "false" otherwise

check_game_started.py

Inputs game name

Description This script checks to see if a certain game has started.

Output "true" if the game has started, "false" otherwise

check_state.py

Inputs game name, player state/history number

Description This script checks to see if a player is at the most up to date state. If the player is up to date, then it prints nothing. Otherwise, it prints out a list of transitions from the player's current state to the most updated state

Output State transitions since player's last update in the form: "player get/lose object_name"

close_start_game.py

Inputs game name

Description This script closes a game and starts it

Output "true" if the game is started, "false" otherwise

delete_game.py

Inputs game name

Description This script deletes an instance of a game

Output "true" if the game is deleted, "false" otherwise

end_game.py

Inputs game name

Description This script ends an instance of a game

Output "true" if the game is ended, "false" otherwise

game_players.py

Inputs game name

Description This script lists the players who are in an instance of a game

Output The usernames of the players in the game, separated by newlines

join_game.py

Inputs game name, username

Description This script allows a player to join a game

Output "true" if the player joins the game successfully, "false" otherwise

list_open_games.py

Inputs game file name

Description This script lists all instances of a game that are open

Output The full game names of the open games, separated by a newline

new_game_request.py

Inputs game name, host username, items

Description This script is called when a user wants to set up a new game. This creates the game, initializes the Object Ownership table, and sets the players to only the player that requested the game, and opens the game for joining

Output "true" if the game is created successfully, "false" otherwise

Appendix B

Data Collection Server and Database Interface

As described in Chapter 4, data collection is maintained through an interaction between the data collection server and the MITAR clients. The MITAR clients interact with the server through several public interfaces visible as python CGI scripts. This chapter will describe these interfaces.

gameData.py

Inputs game name

Description This scripts lists the data collected for a particular game

Output The data collected during a game in the following order: username, student group (currently not used), latitude, longitude, filetype (audio, video, picture, etc.), URL where actual file can be found, title of data collected (if text), summary of data collected (if text), timestamp. Distinct pieces of data are separated by two newlines.

list_game_instances.py

Inputs game name

Description This script lists the instances of the games that are currently available

Output The game instance names, separated by a newline

list_games.py

Inputs None

Description This script lists all game file names that have data stored

Output Game file names, separated by a newline

new_game.py

Inputs game name

Description This script creates a table in the database to store the data metadata. Only use if not multiplayer. Data collection table is already created by multiplayer if the game instance is a multiplayer game.

Output "true" if the table is created, "false" otherwise

upload_pic.py

Inputs game name, username, group (not currently used), latitude, longitude, file

Description This script uploads a picture to the server

Output "true" if the item is uploaded, "false" otherwise

upload_txt.py

Inputs game name, username, group (not currently used), latitude, longitude, text title, text summary

Description This script uploads a text data to the server

Output "true" if the text is uploaded, "false" otherwise

Bibliography

[1] Kaufmann H. "Collaborative Augmented Reality in Education." Institute of Software Technology and Interactive Systems, Vienna University of Technology, 2003

[2] Heilig, Morton L, *Sensorama simulator*, August 1962, http://www.freepatentsonline.com/3050870.html

[3] Feiner S, MacIntyre B and Seligmann D. *Knowledge-based augmented reality.* Commun ACM 1993; 36(7): 5262.

[4] Piekarski W and Thomas B, *ARQuake: the outdoor augmented reality gaming system.* Commun ACM 2002

[5] "Wikitude World Browser." *Wikitude.* Web. ¡http://www.wikitude.org/category/02_wikitude/wo browser¿.

[6] Cheung P. "Charles River City: An Educational Augmented Reality Simulation Pocket PC Game." Massachusetts Institute of Technology, 2003.

[7] Tze Kwang Chin, *Augmented Reality on the iPhone Platform.* Massachusetts Institute of Technology, 2009.