Augmented Reality on the iPhone Platform

by

Tze Kwang Chin

S.B., C.S. M.I.T. 2008
S.B., Mathematics M.I.T. 2008

Submitted to the
Department of Electrical Engineering and Computer Science
in Partial Fulfillment of the Requirements for the Degree of

Master of Engineering in Electrical Engineering and Computer Science

at the Massachusetts Institute of Technology

September, 2009

Author ....
Department of Electrical Engineering and Computer Science
August 21, 2009

Certified by ....
Eric Klopfer
Associate Professor
Thesis Supervisor

Accepted by ....
Dr. Christopher J. Terman
Chairman, Department Committee on Graduate Theses

Augmented Reality on the iPhone Platform

by

Tze Kwang Chin

Submitted to the Department of Electrical Engineering and Computer Science
on August 21, 2009, 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 and implemented an iPhone implementation of the Outdoor Augmented Reality client developed by the Schellar Teacher Education Program at MIT. The work began as a simple port, but it soon became clear that a redesign of the current system architecture was necessary to provide better cross platform compatibility, especially in light of the possibility of a future Android implementation of the game client. I designed a flexible and extensible new architecture that achieves that purpose. Furthermore, the new architecture also adds more features to the game such as having basic AI for game characters.

Thesis Supervisor: Eric Klopfer
Title: Associate Professor

Acknowledgments

I would like to thank Professor Eric Klopfer for providing the vision and direction for the Outdoor Augmented Reality project. I would also like to thank Judy Perry and Josh Sheldon for being fantastic project managers who have kept me on track as the work progressed from week to week. Many thanks to Lisa Stump as well for providing invaluable advice and for sharing her experience as senior developer of the Windows Mobile branch of the project.

THIS PAGE INTENTIONALLY LEFT BLANK

Contents

THIS PAGE INTENTIONALLY LEFT BLANK

List of Figures

THIS PAGE INTENTIONALLY LEFT BLANK

List of Listings

THIS PAGE INTENTIONALLY LEFT BLANK

List of Tables

THIS PAGE INTENTIONALLY LEFT BLANK

Chapter 1

Introduction

In the book *Augmented Learning* [2], augmented reality (AR) refers to a *"combination of the real and the virtual in any location-specific way, where both real and virtual information play significant roles"*. Augmented reality games thus superimpose virtual information onto the real world. There are multiple ways in which this superimposition can be done. However, this thesis will only consider the solution whereby location aware handheld devices are used to provide the user with location sensitive data. For example, the device could display a history of MIT when the user is present at MIT, but when at Harvard a history of Harvard is displayed instead. It is important to note that the virtual information need not be fact, but can be fictitious. It is also not limited in scope to descriptive material such as history, but can also include virtual people and places that can be interacted with on the device.

1.1 Educational Motivation

The purpose of creating augmented reality games is to untether educational simulations from the desktop [3]. By doing so, the players are able to physically participate in the games, as opposed to being limited to virtual interaction in a desktop environment. Players can now interact as they usually would in their daily lives; that is, with body language, facial expressions and multiple other nuances which cannot quite be communicated on a computer. At the same time, players can simultaneously interact with objects in the real world and objects in the virtual world. The simultaneous interaction causes players to think differently about the real spaces that they are playing in, making context and location important to game play. Augmented reality games thus provide a unique experience capable of stimulating a greater interest in learning the educational material that the game hopes to impart.

It is important to clarify that augmented reality games are not single player endeavours, though they can be. Teamwork is an important social skill, and these games hope to promote that. It is conceivable that players are split into teams, where each team is assigned a handheld, and teammates work together to complete the game's objectives. It is also possible to have each player receive a handheld, but have each handheld provide different pieces of information such that group collaboration is required to understand the bigger picture. There are many more possible scenarios but we see that there is precedent to enable group communication in these games.

1.2 Summary of Thesis Content

Chapter 2 will describe previous work done on augmented reality

Chapter 3 will describe the implementation of the Outdoor AR client on the iPhone

Chapter 4 will describe a new framework for Outdoor AR games

Chapter 5 will describe future work for the Outdoor AR platform

Chapter 2

Background

2.1 Previous Games

The first augmented reality game created by MIT's Schellar Teacher Education Program (STEP) was Environmental Detectives [4]. Its premise is that there has been an environmental spill at MIT. The players objective is to locate the source of the spill, discover why it happened, and to find a way to remedy the situation while also reporting any potential health and legal risks. The handhelds allowed players to take water samples at various locations to analyze chemical concentrations, and also to obtain information by interviewing virtual characters. A time limit was also imposed on the game, such that players would not be able to collect all the water samples and interview all the possible characters, forcing them to prioritize. From this we see that the initial game supported time sensitivity, virtual characters and interaction with virtual objects.

The followup to Environmental Detectives was Charles River City [1]. In this game, there has been an outbreak of a myterious illness coinciding with a major event in the Boston area. Players are assigned roles which provide them with special capabilities, such as being able to take different types of samples, or receive unique information. The virtual world changes over time as the disease progresses, which results in different virtual characters and objects appearing at different points in time. Moreover, it is possible for players to trigger events. For example, talking to one virtual character may cause another to appear elsewhere. In short, Charles River City introduced the notion of time dependence, player roles and cascading events.

Note that both of the above games worked only outdoors since they relied on GPS data. However, workarounds were developed to enable players to retrieve information from indoor locations as well. One such idea is the notion of gates and clue codes. Indoor locations with data are represented by gate objects in the game. In order to access the information associated with the gate, the player is required to enter a clue code, which can be found by physically entering the building and looking for the piece of paper with the clue code on it.

An attempt was made at developing a better indoor solution in the Public Opinions of Science using Information Technologies (POSIT) project. A variation of the engine was created whereby location was tracked by checking WiFi access point signal strengths. There was some degree of success in this game, but technical difficulties such as location inaccuracy led to no further investigation into the idea. However, the indoor version introduced the notion of centralized state, whereby each device communicated with a central server such that there was a common virtual world in which all the players existed. This situation allowed for global events to affect players simulteneously, and also allowed players to affect each other. For example, if one player were to pick up an item in some room, that item would no longer be available to other players that subsequently visit that room.

2.2 The OutdoorAr Game Editor

The development of Environmental Detectives and Charles River City laid the groundwork for what could be expected of outdoor augmented reality games. With this paradigm in place, development on an outdoor augmented reality game editor began. The goal of the project was to *"make the process of designing games accessible to even the least technical users"* [5]. This editor significantly simplified the creation of new games. However, it also introduced a tight coupling between the editor and the game engine. Both modules need to be updated when new features are implemented.

2.3 The iPhone Platform

All previous games have been developed on the Windows Mobile platform and targeted at Pocket PC (PPC) devices. Prior to Summer 2008, the iPhone was not a viable platform because its location system used WiFi based positioning using Skyhook and cell tower signals, which is not accurate enough for the purpose of our games. However, the advent of the iPhone 3G that summer changed that because that version of the iPhone comes bundled with actual GPS support. That feature by itself does not warrant such a huge shift in development. The two main reasons why an investigation into the iPhone platform is warranted is because of its technological capabilities and its commercial success among the target audience of the games.

The iPhone is able to send and receive data over the cellular network, and has a rich multi-touch user interface. The ability to send and transmit data to arbitrary destinations over the Internet will allow us to once again have centralized state, like in the POSIT game. The novel multi-touch interface will also allow us to perform more complicated gestures that are not possible on PPCs. Furthermore, the touch interface also replaces the stylus on PPCs, which alleviates part of the learning curve of using the device. Other features of interest include its camera and accelerometers. However, these features will not be leveraged by this thesis.

The commercial success of the iPhone among the target audience is greater than that of Windows Mobile devices, such that even students are starting to have iPhones themselves. It is desirable to get these games on devices that the students have, rather than schools having to provide them. This approach provides us with greater scalability since we can now reach a broader audience.

THIS PAGE INTENTIONALLY LEFT BLANK

Chapter 3

iPhone Implementation

3.1 Game Terminology

We will first define some concepts which will be used for this chapter. A *role* is some role that a given player plays. A *chapter* represents a section in the storyline of a game. A *game object* is an object on the map, such as an item or a non-playable character (NPC). The different game object types are represented by a *template.* A *visible* game object is one that appears to the player on the main map. Game objects are visible under certain *conditions* - for example, some game object may only be visible to players with a particular role. When a player gets close enough to a game object, it is *visited* and may cause *triggers* to occur. A *trigger* causes other game objects to become visible to the player. When an object is visited, the user may access that object's media - this is known as an *interview.* The *history* is the list of game objects that the player has already visited. Lastly, a *glyph* is a colored shape that represents the game object on the map.

3.2 Requirements

The goal for the iPhone implementation of the Outdoor AR game client was to provide a subset of the features available on the Windows Mobile game client. The iPhone client should be able to:

- Read the XML game file format outputted by the game editor
- Support multiple character roles
- Change player location by GPS or by user interaction
- Change the visibility state of objects as the game progresses
- Display game object media
- Display history

The above features were implemented in this thesis as well as:

- A game deployment system for obtaining new games
- A simple game save system for continuing saved games

3.3 System Architecture Overview

The reader should note that all concepts in this section adhere to the standard iPhone conventions. In particular, the notion of a *view* and a *view controller* may differ from what the reader is used to. In short, a view is responsible for drawing in some region and a view controller is responsible for managing views.

Much of the architecture was replicated from the existing Windows Mobile code. The main difference in the iPhone implementation is the choice of using Apple's Core Data framework to store game files in a SQLite database instead of using an XML file. The reason for this design choice is performance - the time taken to load the *Timelab 2100* game from a SQLite database is an order of magnitude smaller than that taken to load it from an XML file. The consequence of using Core Data is that the base class for all objects that can be parsed must inherit from the *NSManagedObject* class provided by Core Data.

Figures 3-1 and 3-2 show the class hierarchy for the implementation. Tables 3.1, 3.2, 3.3 and 3.4 summarize the purposes of the various classes in the architecture:

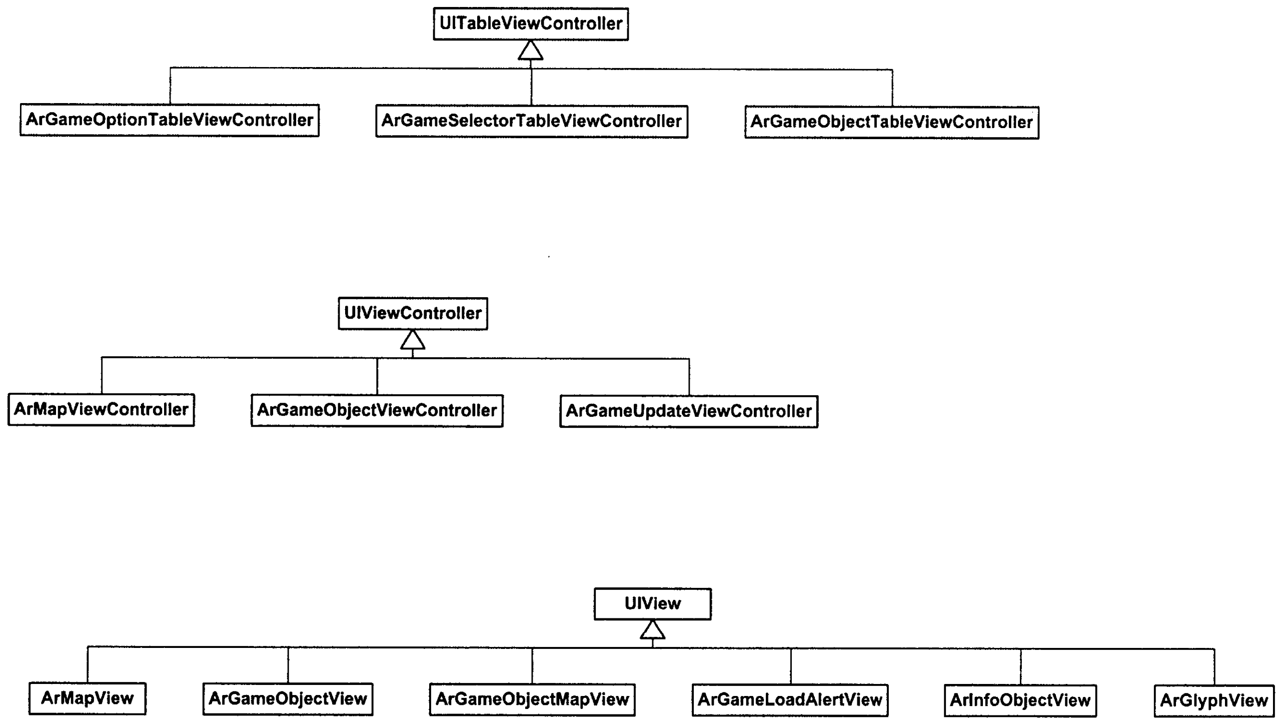


Figure 3-1: iPhone Client User Interface Class Hierarchy

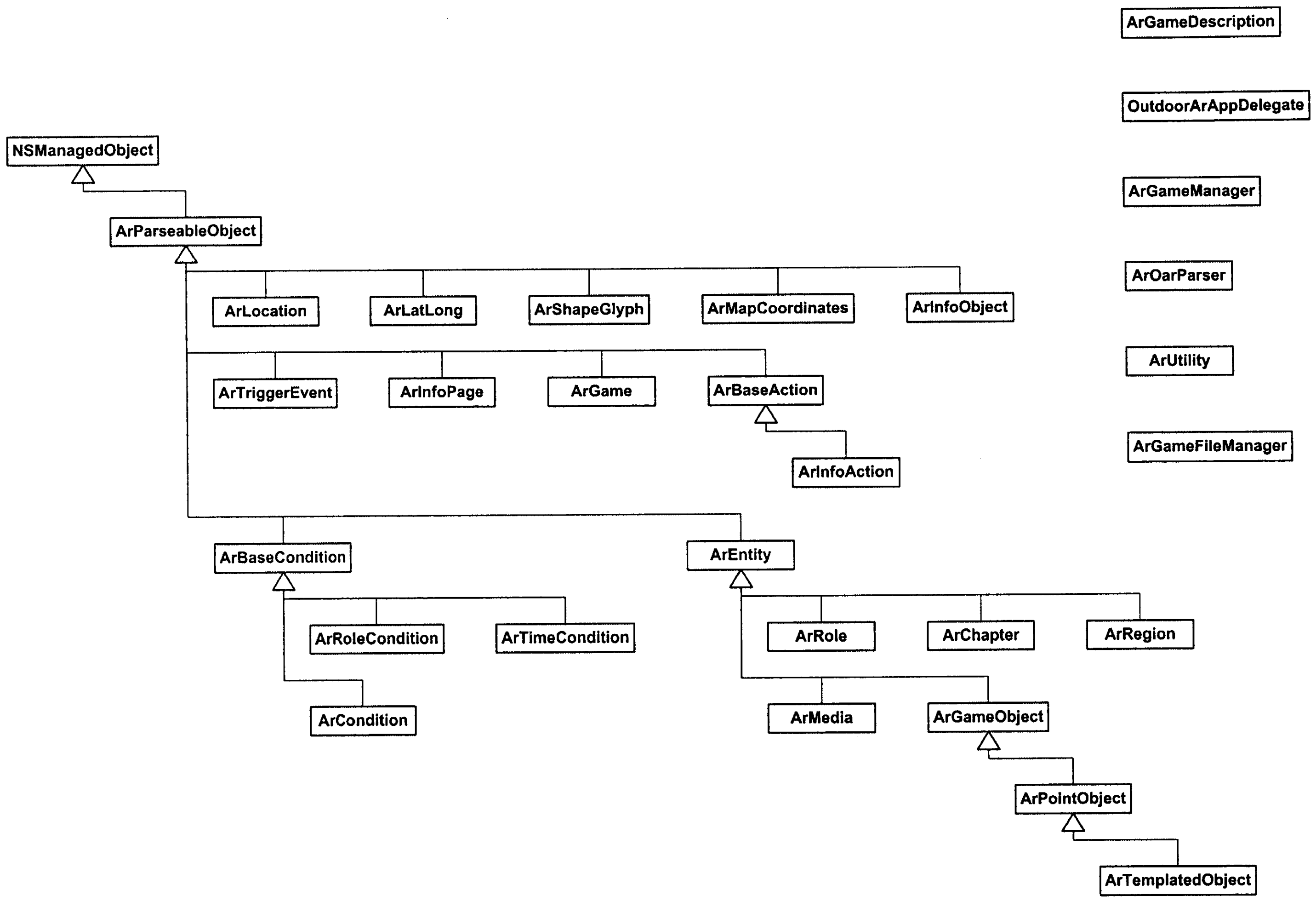


Figure 3-2: iPhone Client Backend Class Hierarchy

Class	**Description**
ArGameLoadAlertView	Asks user whether or not to load the saved game
ArGameObjectView	Displays an ArTemplatedObject
ArGameObjectMapView	Displays a game object on the map screen
ArGlyphView	Displays a glyph shape
ArInfoObjectView	Displays the contents of an ArInfoObject
ArMapView	Displays the map

Table 3.1: UIView subclasses

Class	**Description**
ArGameObjectViewController	Controller for ArGameObjectView
ArGameOptionTableViewController	Presents game options
ArGameSelectorTableViewController	Presents available games
ArGameUpdateViewController	Presents available updates
ArMapViewController	Controller for ArMapView

Table 3.2: UIViewController subclasses

Class	**Description**
ArBaseAction	Base class for actions
ArBaseCondition	Base class for conditions
ArChapter	Stores information about a game chapter
ArCondition	Stores multiple conditions
ArEntity	Base class for id-based objects
ArGame	Stores information about the game
ArGameObject	Stores information about a game object
ArInfoAction	An action that represents an interview
ArInfoObject	Stores ArInfoPages
ArInfoPage	Stores interview information
ArLatLong	Stores latitude and longitude coordinates
ArLocation	Stores game coordinates
ArMapCoordinates	Stores latitude and longitude information for a map
ArMedia	Stores information about media files
ArPointObject	A game object located at a point
ArRegion	Stores information about a game region
ArRole	Stores information about a game role
ArRoleCondition	A condition that is met for a particular role
ArShapeGlyph	Stores information about glyph color
ArTemplatedObject	Stores information about a templated game object
ArTimeCondition	A condition that is met for a particular chapter
ArTriggerEvent	Represents an event that triggers visibility

Table 3.3: Data Container Classes

Class	**Description**
ArDownloadConnection	Used to asynchronously download a single file
ArGameDescription	Provides a short description of a game
ArGameFileManager	Manages the game files
ArGameManager	Manages the active game and retrieves GPS information
ArOarParser	Performs deserialization of the game file
ArParseableObject	Base class for all objects parseable from the game file
ArUtility	Utility class that provides convenience functions
OutdoorArAppDelegate	Application delegate for the client

Table 3.4: Other Classes

3.4 Game Deployment

As mentioned before, a new feature added to the iPhone client is the ability to retrieve new games that are available. This feature was not previously developed on the Windows Mobile client because games could be transfered directly to Windows Mobile devices via *ActiveSync*, whereby game files could be directly copied to the device's filesystem. However, no such mechanism exists for the iPhone, which does not expose its filesystem in the same manner as Windows Mobile devices. Hence, the development of a game deployment system is necessary.

Game deployment is done by maintaining an XML file at a predetermined remote location that stores information about the games available for download. We shall refer to this XML file as the remote *game list*. The children of the root element of the remote game list are game descriptions that specify name, game id, oar file, last modified and file url. Listing 3.1 shows an example of a remote game list, and Listing

3.2 shows its associated DTD.

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE ArGameList SYSTEM "outdoor_ar_gamelist.dtd">
<ArGameList>
<GameInfo>
  <Name>Timelab2100 Outdoor</Name>
  <GameId>aa343384-24e4-4742-9c65-f5217fadc3aa</GameId>
  <OarFile>timelab2100.oar</OarFile>
  <LastModified>6-27-2009 11:16 pm</LastModified>
  <FileUrl>http://education.mit.edu/timelab2100.zip</FileUrl>
</GameInfo>
<GameInfo>
  <Name>Timelab2100 Indoor</Name>
  <GameId>aa343384-24e4-4742-9c65-f5217fadc3ab</GameId>
  <OarFile>timelab2100_indoor.oar</OarFile>
  <LastModified>6-27-2009 11:16 pm</LastModified>
  <FileUrl>http://education.mit.edu/timelab2100_indoor.zip</FileUrl>
</GameInfo>
</ArGameList>
```

Listing 3.1: Remote Game List Example

```
<!ELEMENT ArGameList (GameInfo)*>
<!ELEMENT GameInfo (Name,GameId,OarFile,LastModified,FileUrl)>
<!ELEMENT Name (#PCDATA)>
<!ELEMENT GameId (#PCDATA)>
<!ELEMENT OarFile (#PCDATA)>
<!ELEMENT LastModified (#PCDATA)>
<!ELEMENT FileUrl (#PCDATA)>
```

Listing 3.2: Game List DTD

Observe that the file url points to a zip file. A game consists of a *oar* file containing game information and also all the various media files used by the game, which can include audio files, video files and documents in the form of HTML and plain text. We choose to distribute a game via a single archive instead of many individual files to avoid the overhead of having to create a new connection for each file. This strategy is particularly effective when a game contains many files. For example, the *Timelab 2100* game has 224 files and the total time taken to download its files individually is an order of magnitude greater than that taken to download and unarchive the single zip file.

After a given game is downloaded and unarchived, it is deserialized and converted into a Core Data storage format. Next, it is added to the local game list file which stores descriptions of all the games available on the local filesystem. The local game list is in the same format as the remote game list and conforms to the same DTD. The local storage is managed by the *ArGameFileManager* class.

3.5 Game Deserialization

As mentioned in the previous section, a game's information is contained in an *oar* file. This file is really an XML file generated by using the .NET XML serialization library. Since we cannot utilize .NET methods on the iPhone, it is necessary to implement a parser that will read that XML and create the appropriate objects.

The parser is implemented in a manner that is class independent. Individual serializable classes do not have to provide specialized parsing code - rather, they will inherit the appropriate methods from the class *ArParseableObject* which parse the game file's XML correctly. This behavior is achieved by using game file format information stored in the application bundle's property list, the *NSXMLParser* class provided by the iPhone SDK, and reflection.

The game file format information mentioned above is a mapping from the element names in the game file to the actual names of classes and fields of the data structures that those elements represent. The OarParserConfig property of the property list contains a list of two dictionaries called ElementToClass and ClassConfig. The ElementToClass dictionary stores a map between element names and class names. The ClassConfig dictionary stores a map from class name to a list containing a dictionary called FieldInfo and a list called IgnoredElements. The FieldInfo dictionary stores a map from element name to field name and field type, whereas the IgnoredElements list stores the list of elements that should be ignored for the class.

Given a class name, we can use reflection to dynamically create an instance of that class. Given the field name and type, we can dynamically obtain the method used to set the field of that class. Hence, reflection coupled with the game file format information allows us to translate the element names in the game file to instances of the objects they represent.

The NSXMLParser is event based, which means that we are notified when new tags are encountered and it is our responsibility to provide callbacks to handle each event type. In particular, we need to provide a delegate that implements methods to call when the start of an element is found, characters are found, and when the end of an element is found. These methods are implemented in the *ArParseableObject* class and the parser's delegate is always set to be some instance of an *ArParseableObject*. Also, an *ArParseableObject* keeps two important fields - its parent delegate (the object that previously set this object to be the parser's delegate) and a string buffer.

When the start of an element is found, we instantiate the appropriate class associated with that element and set that object to be the parser's delegate. Also we set the parent delegate of the newly instantiated object to be the current object. Lastly, we set the relevant field of the current object to be the newly instantiated object. However, if the class is actually a primitive type then we just set our string buffer to be empty without changing delegates.

Upon finding characters, we add those characters to our string buffer.

When the end of an element is found, we look up the appropriate class associated with that element. If it is a primitive type we convert the contents of our string buffer to that type and we set the appropriate field of the current object. If the type is the same as that of the current object it must be the end tag for this object, so we set the parser's delegate to be the current object's parent delegate.

The special case that occurs is the start and end of serialization whereby we make use of the fact that the root element of the game file XML is an instance of *ArGame*. The parser's delegate is first set to be an instance of *ArGame* which does not have a parent delegate.

3.6 Gameplay

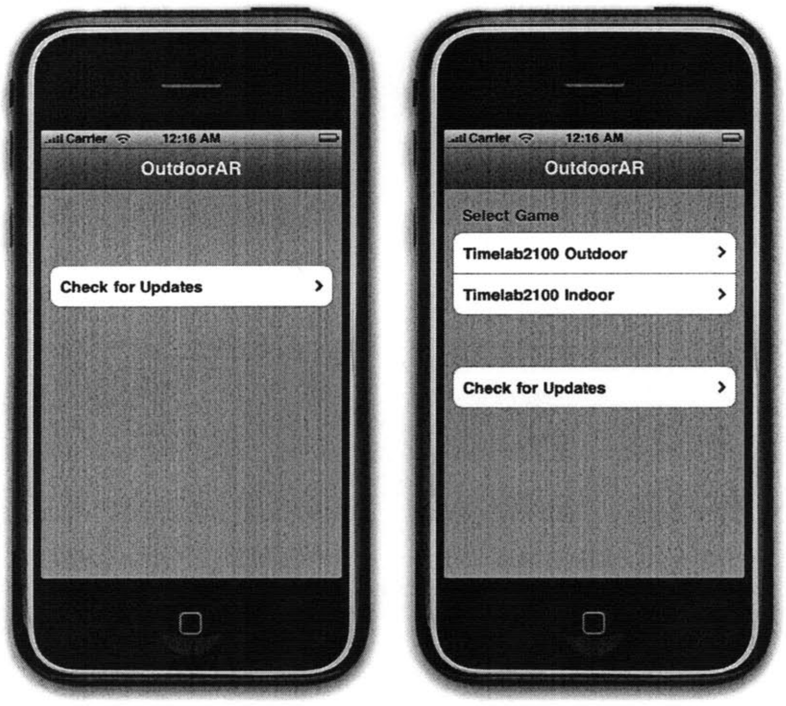
Gameplay is driven by GPS update events. The *ArGameManager* is responsible for subscribing to those events and to update the current location. Whenever the location is updated, the player's distance to all game objects is calculated. If the player is close enough to a game object, it is triggered.

3.7 User Interface

This section describes the various screens that are presented to the user.

3.7.1 Game Selection

During game selection, the user is presented with the list of games that have been downloaded to the iPhone. Also, there is an option to check for updates which checks the remote game list for new games. Figure 3-3 shows this screen.

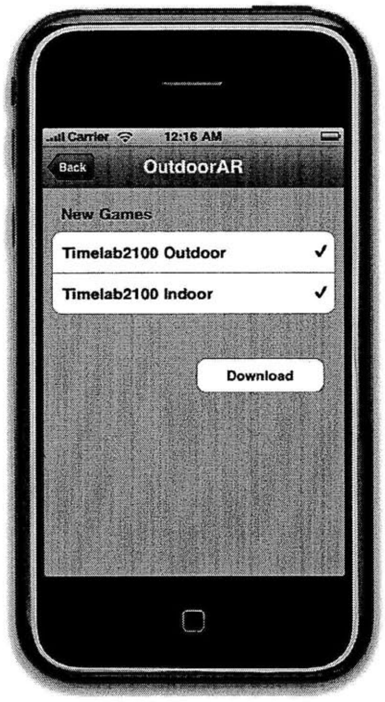


(a) No Games Available (b) Two Games Available

Figure 3-3: Game Selection View

3.7.2 Game Update

The game update screen presents the user with available new games and update progress. The user may check select some number of these for download. Figure 3-4 shows this screen. When the update is complete an alert is presented to the user, as shown in Figure 3-5.

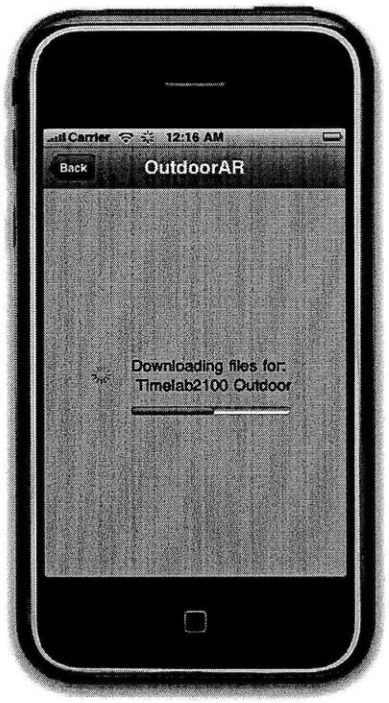


New Games

Timelab2100 Outdoor ✓

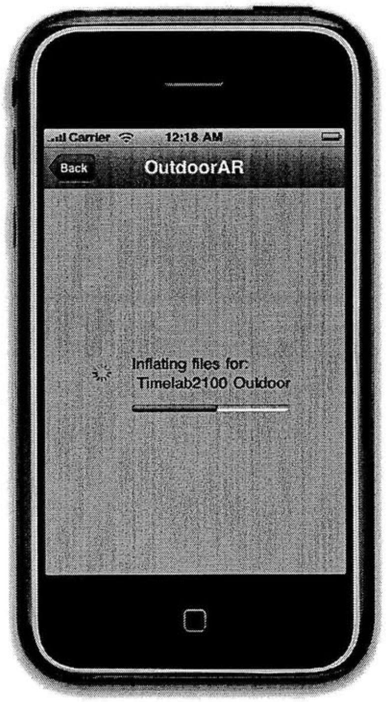
Timelab2100 Indoor ✓

(a) Two Games Available



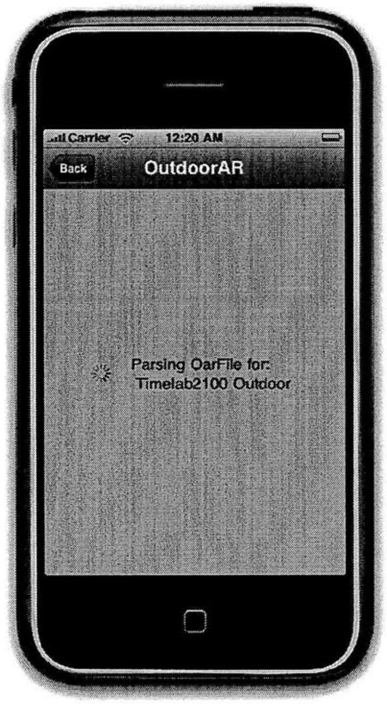
Downloading files for:
Timelab2100 Outdoor

(b) Download in Progress



Inflating files for:
Timelab2100 Outdoor

(c) Inflate in Progress



Parsing OarFile for:
Timelab2100 Outdoor

(d) Parse in Progress

Figure 3-4: Game Update View

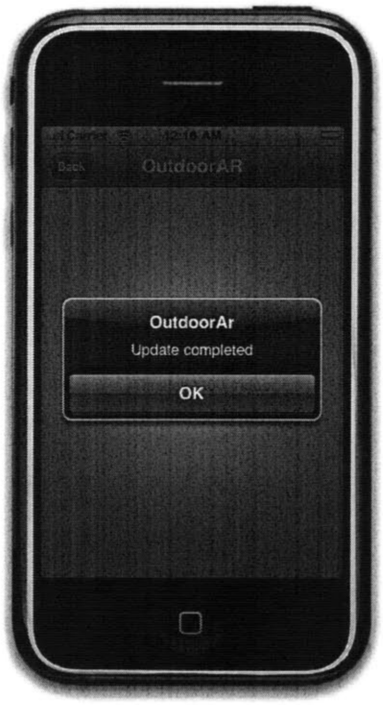
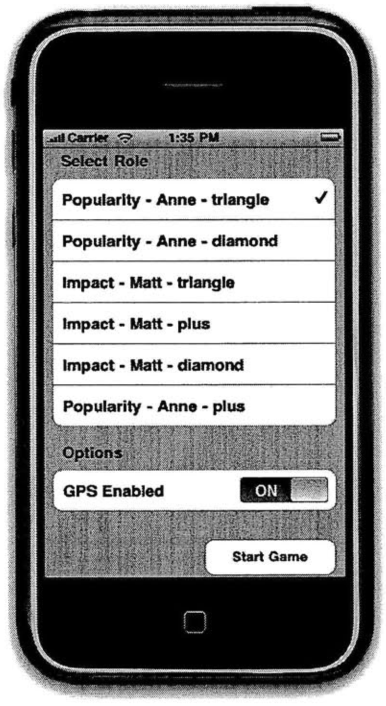


Figure 3-5: Game Update Alert View

3.7.3 Game Options

The game options view presents the user with the options associated with the game. The options available in this client's implementation are role selection and gps mode. Figure 3-6 shows the view in question.



Select Role

Popularity - Anne - triangle ✓

Popularity - Anne - diamond

Impact - Matt - triangle

Impact - Matt - plus

Impact - Matt - diamond

Popularity - Anne - plus

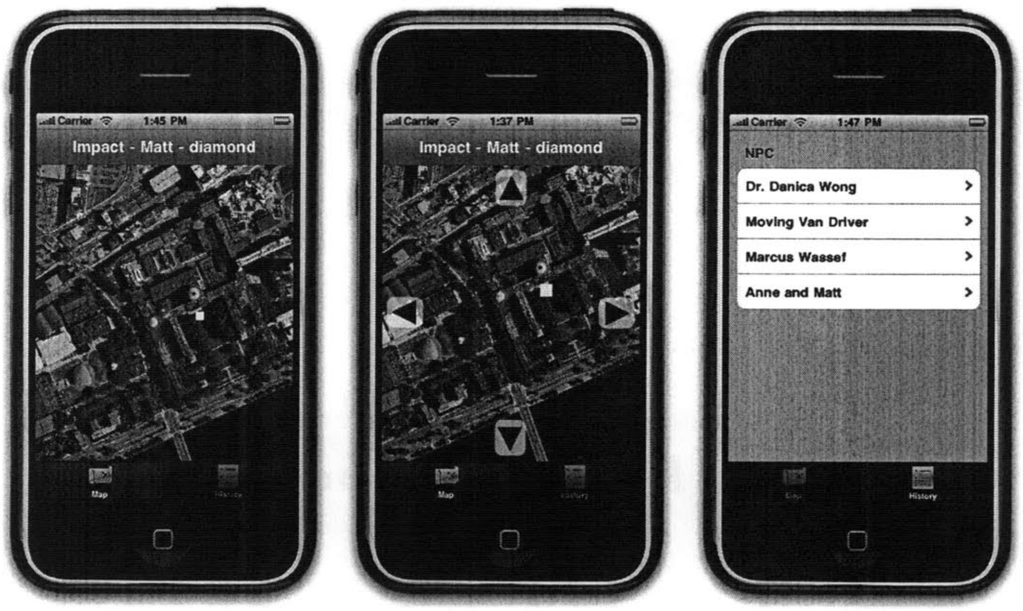
Options

GPS Enabled ON

Figure 3-6: Game Options View

3.7.4 Primary Game View

The primary game view comprises of a tab bar that switches between the map view and the history view, as shown in Figure 3-7.



(a) Map View - GPS (b) Map View - Manual (c) History View

Figure 3-7: Primary Game View

Map View

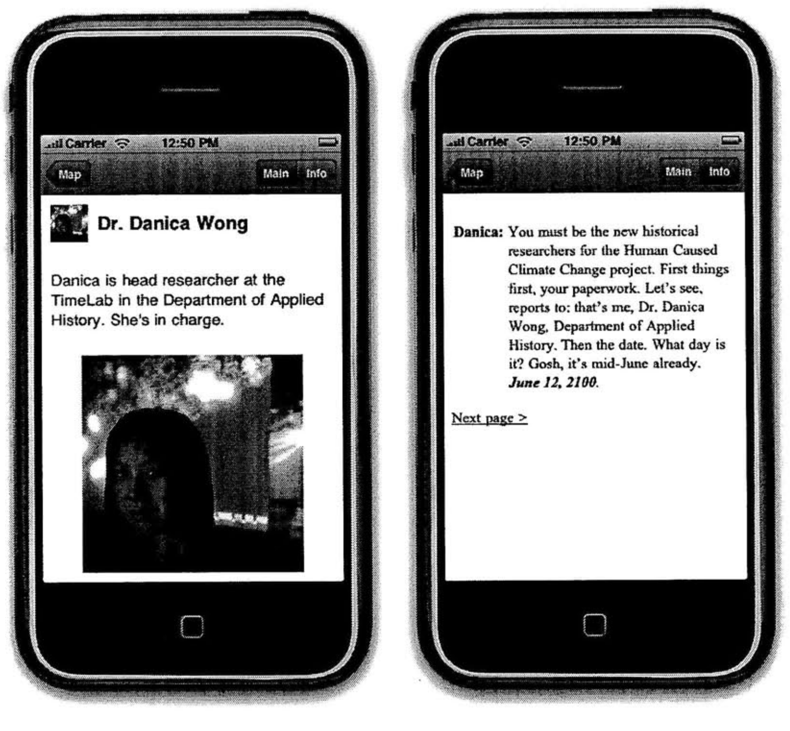
The map view is scrollable and zoomable, and displays the player's current position as a blue dot, similar to that used in the *Maps* application on the iPhone. The player's current role is displayed on the navigation bar. Also, it displays the visitable game objects on the map as glyph shapes. When a player gets close enough to a glyph, it enlarges to make it apparent to the player that the location is nearby and visitable. Lastly, in the case where GPS is not enabled, directional buttons are provided which can be tapped to manually change the player's location.

History View

The history view displays a list of the game objects that the player has previously visited. The player may review the information associated with those objects again by tapping the relevant object names.

3.7.5 Game Object View

The game object view displays the name and description of the game object, as well as the information associated with interviewing the object. A segmented control controls what is currently shown to the player. Note that the information is stored as HTML, so we are actually rendering HTML using the *UIWebView* class on the information page. Figure 3-8 shows this view.



Dr. Danica Wong

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

Danica: You must be the new historical researchers for the Human Caused Climate Change project. First things first, your paperwork. Let's see, reports to: that's me, Dr. Danica Wong, Department of Applied History. Then the date. What day is it? Gosh, it's mid-June already. ***June 12, 2100.***

Next page >

(a) Name and Description (b) Info

Figure 3-8: Game Object View

3.8 Lessons Learned

The primary difficulty encountered while porting the game client to the iPhone was the replication of the previous client's framework. While it would be possible to simply perform a direct copy of the system architecture to obtain all of its strengths, doing so would also mean that we inherit all of its weaknesses. These weaknesses exist not only because the original client was not written with cross platform compatibility in mind, but also because the game code has undergone numerous iterations in which different paradigms were adopted, resulting in a rather confusing system.

Rather than spending time to compensate for those weaknesses and developing workarounds, it is better to spend some time to rethink the existing architecture. For example, in the case of game deserialization it is better to create our own protocol for serializing objects rather than having to parse XML generated by the .NET XML serialization library. Given that the architecture requires some amount of redesign to better support cross platform development, it is also an opportune time to unify all of the ideas and experiences gained from the previous iterations of the game client and integrate that into a cohesive architecture.

With the experience gained from having to develop an iPhone implementation of a game client, I designed a new architecture that better supports cross platform development, and also addresses some issues with the current architecture. This architecture is outlined in Chapter 4.

THIS PAGE INTENTIONALLY LEFT BLANK

Chapter 4

The MIT Augmented Reality Framework

In this chapter we will describe a new framework for augmented reality games. This framework has been implemented in part in Java, but as of the time of writing is not fully implemented in either the Windows Mobile client or the iPhone client. As such, there has been limited testing for considerations such as performance, but no major changes to the described framework are anticipated.

4.1 Motivation

The main purpose for developing a new core framework is to create an architecture that is easily portable to other platforms, while stripping away legacy code from our system. At the same time, we want to develop a single design specification that will be adhered to for all platforms since such a document does not currently exist.

4.2 Challenges

The major challenge in creating cross platform support is the maintenance of the project's code base. We cannot assume that the code for the different platforms can be written in the same language and compiled for the correct platform. This point should be clear by observing that the iPhone SDK mandates the use of Objective-C, Windows Mobile mandates the use of C# and Android mandates the use of Java. We should thus assume that to some extent we will have to maintain separate code for each of the platforms we support. Hence, we are faced with the task of simply keeping the maintenance work for our supported platforms to a bare minimum, especially when we wish to add new functionality.

The other challenge in this endeavor is the format, generation and deployment of the developed games. The same game file format should be readable by all platforms, the generation of the game files should account for platform specific settings and the method by which games are distributed to each platform should be user-friendly.

Lastly, given that there has been a lot of development on the WIndows Mobile platform already which has been released to the public, we also need to address the question of backwards compatibility and also the question of how to minimize the impact of the framework migration with respect to preserving as much of the current ideas and concepts as possible.

4.3 Goals

The new framework should meet the following requirements:

1. Flexible and Extensible
2. Decoupled from User Interface

Flexibility and extensibility is stressed in this design because we wish to avoid as much as possible having to change any core functionality. A large change in one platform translates to a large change in other platforms, which causes much unnecessary work if it can be avoided. It is much preferable to allow for new features to be added simply by extending the existing framework. However, the cost of adding flexibility and extensibility to any given framework is that of complexity. We have to ensure that the system is still easy to use despite any added complexity. Furthermore, we need to ensure that the system performance does not suffer due to the new design.

It is important that we implement this framework in a manner that is decoupled from the user interface because it is a definite fact that the user interface code for all platform implementations will be different. As such, the backend cannot rely in any manner on the particular version of the user interface that is being used. Therefore, the game client needs to be driven by classes that have already been defined in the backend. There should be almost no game logic in the user interface code. Rather, the user interface is used to interact with the backend.

4.4 System Architecture

4.4.1 Game Ticks

The new system is driven not only by GPS updates, but also by what we term *game ticks*. Game ticks model the passage of time in this system. The reason for including game ticks in the new system is so that we can implement state specific behavior, where state can change as time passes. For example, game ticks allow us to implement basic artificial intelligence (AI) for NPCs using finite state machines, which is discussed in a later section. It also enables us to maintain a set of global events that will fire when certain conditions are met.

The above features were requested in some fashion by the users of the current client which is why they are being implemented in this iteration. They could have been implemented in a way that does not require game ticks. We would instead make use of separate threads that update the relevant subsystems at some fixed interval. The problem with using multiple threads in that fashion is that we are not guaranteed that all objects are updated at the same time. By using game ticks, a single thread drives all the objects that require updates, which mean that their state is updated at roughly the same time and is consistent with one another.

The class responsible for generating game ticks is *ArGameManager*. Classes that need to receive game ticks should implement the *ArGameTickListener* interface, which is shown in Listing 4.1.

```
public interface ArGameTickListener {
        public void tick(long t);
}
```

Listing 4.1: ArGameTickListener interface

Game ticks are propagated throughout the system in an indirect manner. We make use of the fact that the *ArGame* class is used as a container for all the other components of a game. Whenever a component that implements *ArGameTickListener* is added to the game, the *ArGame* should also add it to a list of *ArGameTickListener* objects. Now, we have *ArGame* implement the *ArGameTickListener* interface. On each game tick, it calls the tick function of each object in its list of *ArGameTick-Listener* objects. If one of the objects in that list has as a field another object that implements the interface, it should propagate the tick to that object when its own tick is called (in addition to doing whatever it does on each game tick). Lastly, each time the *ArGameManager* generates a tick, it just calls the tick function of the active instance of *ArGame*.

We perform the propagation in this manner instead of having classes directly subscribe to the game manager to avoid a strong coupling between the listeners and the game manager.

4.4.2 Communication between User Interface and Model

The question that arises now is how the user interface is notified when something changes in our backend model. This is solved in this system by the use of a mediator class which we will call *ArEventManager* along with an interface called *ArEventListener*.

The *ArEventListener* interface defines a single method called *onEventFire* that takes an *Object* and an *ArEventType* as its arguments. The *ArEventManager* class is a singleton that maintains a mapping between a particular event type as defined in the *ArEventType* enumeration and a list of *ArEventListener* objects that are interested in that event type. It has *subscribe* and *unsubcribe* methods that take as arguments an *ArEventListener* and an *ArEventType*. It also has a *fireEvent* method that takes as arguments an *Object* and an *ArEventType*. When the fireEvent method is called, all the listeners for the provided event are notified and passed the same object and event type.

When something of interest changes in the model, the *ArEventManager*'s fireEvent method is called with the appropriate arguments. For example, when an NPC has changed its location, the fireEvent method is called with the NPC game object and *ArEventType.LOCATION_UPDATE* as arguments. The user interface component responsible for drawing the NPC should implement the *ArEventListener* interface and subscribe to the location update event type. Then, it will be notified when the location has changed and will be able to redraw itself in the right position.

In this system, some care must be taken when subscribing to events to avoid performance issues. Consider the scenario where the game map is represented by some user interface component, and the game objects on the map are represented by some other user interface component that is a child of the map component in the component hierarchy. If the game object map components each subscribe to location update events, we'd have a problem on our hands.

Imagine that there are a hundred such game object map components. Say one of the underlying game objects moved. The *ArEventManager* now calls the *onEventFire* method of all of the hundred game object map components, but only one of them actually moves. We have now 99 wasted function calls. Instead, the map component should be the one to subscribe to the event manager. When the location change event occurs, the map component is notified. It then uses the object argument of the *onEventFire* method to retrieve the associated map component. That component's *onEventFire* method is now called, and the game object component redraws itself.

This system is sufficient because in practice there will be few listeners but many event sources. The user interface components that subscribe for a particular event should know how to handle that event correctly and are passed sufficient information to do so because the originator of the event is provided. Some other examples of changes that cause an event to propagate through the event manager are visibility changes, region changes, game victory, game failure and players visiting game objects.

4.4.3 Entities

Game entities inherit from the abstract *ArEntity* class. Entities are id based objects in the game, not to be confused with non-player characters and items. For example, an instance of *ArMedia* is considered to be an entity. Ids are unique to each instance of an *ArEntity* - no two entities should share the same id. The primary use of entity ids is to differentiate between two equal *ArEntity* subclasses. For example, two monsters could have the exact same parameters, and can only be differentiated by their unique ids.

4.4.4 Conditions

Conditions are objects that check if some game condition has been met by a player and subclass the abstract class *ArCondition*, which is shown in Listing 4.2.

```
public abstract class ArCondition extends ArSerializable {

  protected ArCondition();

  public abstract boolean isConditionMet(ArPlayer player);
}
```

Listing 4.2: ArCondition class

Some convenience classes are provided in the framework that provide common functionality. For example, *ArTrueCondition* implements the abstract method by always returning true. Similarly, *ArFalseCondition* implements the abstract method by always returning false. The other two convenience classes of note are *ArOrCondition* and *ArAndCondition*, each of which maintain a list of *ArCondition* objects. As their names imply, the *ArOrCondition*'s method will return true if any of its contained conditions returns true, and the *ArAndCondition*'s method will return true if all of its contained conditions returns true.

Other conditions of note that are provided by this framework are *ArRoleCondition*, *ArChapterCondition* and *ArProbabilityCondition*. An *ArRoleCondition* object stores a reference to an *ArRole* object and its condition is met when the player's role is equal to the stored role. Likewise an *ArChapterCondition* stores a reference to an *ArChapter* and its condition is met when the active chapter is equal to the stored chapter. On the other hand, an *ArProbabilityCondition* stores a float p and its condition is met with probability p. These various classes demonstrate how flexible a condition can actually be. Also, these classes are lightweight since only references or primitive types are ever stored.

4.4.5 Actions

Actions are ways in which a player may interact with some game object, and inherit from the *ArAction* class. An action has a name, description and some condition that must be met before a player can execute it. The name refers to what is displayed to the user when the actions are listed, and the description describes the effect of the action. Some actions may be performed repeatedly, others may only be performed once. Once an action has been completed, it cannot be executed again unless the repeatable flag of the action is set to true.

An example of an action is an *ArInterviewAction*. Its default name and description are 'Interview' and the empty string respectively. This action is repeatable because after interviewing an NPC, we may wish to revisit the NPC to interview it again. Although the execution condition may default to being an *ArTrueCondition*, it could also be that the the action is only executable if the player is of a particular role. An *ArInterviewAction* also contains the information that is displayed to the player during the interview, which is some form of media.

4.4.6 Events

In a previous section we defined *ArEventManager* and *ArEventListener*. We are now going to define another class called *ArEvent*, which can be thought of as a container for some event type. An *ArEvent* is similar to an *ArAction*. It has a condition that must be met before it can occur and may occur repeatedly. We say that the *ArEvent* is a container because the event type will not be fired before the *ArEvent*'s condition is met.

To illustrate, we shall consider how to fire a game won event when the game is won by achieving some condition. Where should that condition be checked? Although we could hardcode that into the game manager such that it checks that condition each game tick, it is a rather inelegant solution. What we do instead is subclass *ArEvent* to create *ArWinEvent*. The stored event type of an *ArWinEvent* object is *ArEventType.WIN*, and the event condition is the win condition. This *ArWinEvent* is stored in a list in an *ArGame* object called the global event list. On each game tick, the objects in that list are checked to see if they can be fired. If so, then they are. Once the win condition is met, the *ArWinEvent*'s fireEvent method is called, which then calls the *ArEventManager*'s fireEvent method, which then notifies the appropriate listeners and the game ends with a victory.

4.4.7 States

AI is implemented by constructing a hierarchical finite state machine using two classes - *ArState* and *ArStateMachine*, which both implement the *ArGameTickListener* interface. An *ArStateMachine* contains a set of *ArState* objects and keeps track of the current state. On each game tick, the state machine forwards the tick to the current state. Hence, the behavior exhibited by the state machine on each game tick depends on the behavior of the current state. State transitions are handled by the *ArState* objects themselves. When a state transition occurs, the state object calls the *setCurrentState* method of its *ArStateMachine*.

A new state machine with a given set of states should be implemented for each behavior desired. However, it is possible to reuse existing state machines. To do so, we create a new state that maintains a reference to not only the parent state machine, but also to the state machine whose behavior we are reusing. When the parent state machine calls the state's tick, the state forwards that tick to its enclosed state machine (provided no state transitions occur). In this manner, the new state emulates the behavior of its enclosed state machine.

For example, say we have a state machine that causes an NPC to patrol between certain waypoints, but will cause the NPC to stop if a player is nearby. The two states that belong to this state machine are *Patrol* and *Idle*. During the *Patrol* state a state transition occurs to the *Idle* state if a player is nearby. During the *Idle* state a state transition occurs to the *Patrol* state if a player is not nearby.

Now say we want to create another state machine whereby in addition to patrolling and stopping for players, the NPC should attack any nearby enemies. We don't want to redefine new *Patrol* and *Idle* classes for this new state machine. Instead, we enclose the previous machine in a new state called *PatrolStateMachineState*. If this were the sole state of our new state machine, it would behave exactly like the previous state machine. However, we are adding a new *Attack* state. In the *PatrolStateMachineState*'s tick function, we transition to the attack state if an enemy is nearby. In the *Attack* state's tick function, we transition to the *PatrolStateMachine* if no enemies are nearby. Hence, our new state machine reused our old state machine to create a combination of behavior.

4.4.8 Map Coordinates

An *ArMapCoordinates* object stores xy-coordinates as well as a region.

4.4.9 Players

A player object contains data about the player's team, role, current location and history.

4.4.10 Media

Media objects inherit from *ArEntity* and contain information about some media file. This media may be an image, HTML, audio or video file. They are marked with the content type and also the local file path.

4.4.11 Regions

An *ArRegion* stores information about a given region. This information includes the image path and dimensions for the image representing the region, the latitude and longitude coordinates for the boundaries of that region, the game coordinate bounds, the game objects in that region and a flag to indicate whether or not the region is indoors.

4.4.12 Game

An *ArGame* is a container for all the elements of a game. These elements include the entities, global events, roles, chapters, players, regions, game name, game id and game description.

4.4.13 Location

Locations on the map are represented by subclasses of *ArLocation*, not to be confused with *ArMapCoordinates*. Locations store some set of *ArMapCoordinates* which can be queried to see if the player is in its hotspot or nudge zone. They can also be queried for map coordinates, though what is returned may vary depending on the purpose of the subclass.

The simplest example of an *ArLocation* is an *ArPointLocation*, whose hotspot and nudge zones are simply circles of some radius around the point it represents. The returned map coordinates is simply the coordinates of the point. A location could also be area based. For example, *ArPolygonalLocation* represents a location described by a polygon with more than 2 vertices, whose hotspot and nudge zones correspond to area enclosed by the polygon. In this case, we could define the returned map coordinates to be the centroid of the polygon. For convenience, we also define *ArLocationEverywhere* and *ArLocationNowhere* as sentinels for those cases.

4.4.14 Game Objects

Game objects are objects that the player can interact with in the game and are instances of *ArGameObject*. Game objects contain the following information:

1. A location that represents where the object is located
2. A condition that must be met for the object to be unlocked
3. A condition that must be met for the object to be visible
4. An optional additional condition for the object to be visitable

5. An optional additional condition for the object to be nudgeable

6. A list of actions that can be performed on or with the object

7. A state machine that handles the object's state

8. A property list that contains information used by the UI drawing code

We now define clearly the concepts of *Visibility*, *Visitability* and *Nudgeability*. When an object is visible, it is viewable to the player and meets its visibility condition. When an object is visitable by a player, the player must both be in the hotspot zone of the object's location and must meet the additional visit condition. Similarly, when an object is nudgeable by a player, the player must both be in the nudge zone of the object's location and must meet the additional nudge condition. Some invariants must always hold - a visible object must be visitable and a visitable object must be nudgeable. Note that the converse does not hold in both cases. A visitable object might be invisible, and a nudgeable object might not be visitable. As an example, a treasure chest is invisible, but we may stumble across it.

Visibility

On the previous iteration of the Outdoor AR client, visibility was implemented as an amalgam of a visible flag and visibility conditions, as opposed to only using the visibility condition. We will now discuss the merits and pitfalls of the two approaches.

In the approach where we use the combination of the flag and the condition, the flag has three states - ForceVisible, ForceHidden and Default. The default flag specifies to check the visibility condition object, and the other flags specifies to force visibility or invisibility. The force visibility flags are used when an object causes another object to become visible. This approach saves a good amount of function calls since we need only compare the fields, and not evaluate conditions to check visibility. However, it leads us to question why we use a visibility condition at all, if we merely toggle a flag. in fact, we could simply always ignore the visibility condition, and set the visibility flag as required.

The issue with using a flag is that we can get ourselves into a situation where it is unclear whether or not a object should be visible given that triggering an object changes the visibility state of other objects. Consider the following scenario:

- Visiting A causes B1 and B2 to become visible.
- Visiting B1 causes D to become visible.
- Visiting B2 causes C to be come visible.
- Visiting C causes D to become invisible.
- A and D are initially visible.

The result of visiting A,B1,B2,C and of visiting A,B2,C,B1 are not in fact the same. What we probably wanted to say is that D is visible as long as C has not been visited, which can be easily represented as a condition. We could in fact use the default visibility flag which uses the condition flag, but then now we'd have a group of objects whereby it quickly gets confusing which object is causing what to become visible.

The alternative approach simply uses condition objects for checking visibility all the time. The conditions can be arbitrarily complex - the condition could be that the player has visited objects A and B, but not C. However not each condition object needs contain the entire history of the player's visitations. For example, if B is visible only if A is visible, and C is visible only if B is visible, then it is implicit that A must be visited for a player to visit C. One could argue that in this situation we need to remember what the player has visited, but the user interface is already keeping track of this because visited objects are drawn differently.

The downside of this approach is the added function calls and the danger of many condition objects floating around in memory. For example, if a hundred objects are visible only if object A has been visited, then we might have a nasty situation where we have a hundred equivalent condition objects. For this reason, conditions should be implemented using the flyweight design pattern, where we would share the one condition object that represents the condition that A has been visited. This strategy will drastically reduce the memory footprint of using condition objects.

Note however in this approach we can now do away with visibility events. Visiting an object will now not cause others to be visible directly - rather, visiting an object causes the visibility condition of other objects to be met. From the user's point of view however, it is still pretty much the same thing.

Lastly, we address the issue of when visibility is checked. This new model of approaching visibility allows for visibility to be dynamic - it is no longer the case that visibility can only change after visiting a game object. Visibility is now checked periodically during a game tick. A game object checks its visibility during a game tick if a certain amount of time has passed since the last visibility check. If the visibility changed, a visibility change event is fired. Note that this duration does not have to be fixed - it can be a random number in some range. The performance implication of checking visibility periodically is as of yet unknown and may be platform independent. However, it is not expected a large performance hit will result with the strategy outlined above.

Locked Objects

Until an object's password condition is met, the user can only view it's name and description but may not otherwise interact with it. Password conditions are special in the sense that passwords are dependent on the game state. For example, we may have a different password for each player role to access the same object. We are not restricted to having such a role condition - we may also want to add other conditions such as being in a particular chapter, and the player having visited a particular object beforehand. All of this information is written into a single *ArPasswordCondition* object that is then assigned to the relevant game object.

There are two types of locked objects - those that appear on the map that the user is able to visit, and those that do not appear on the map but will appear when the user enters a code on some other screen. Dealing with the former case is simple enough - the password condition is simply checked when the player visits the object in question. Dealing with the latter however is tricky for the reason that we may want behaviors other than visiting objects after typing in a particular code. For that reason, we support the latter type of game objects via some indirection.

The solution is to use an *ArVisitEvent* which maintains a reference to the game object in question and has as its event condition the relevant password condition. This event is added to the global event list and is set to be repeatable. Once the user enters a matching password the event is fired. Note that it may be possible that more than one event may be fired as a result of entering one code - the user interface should deal with this as appropriate.

Game Object Property List

This list is a map between property names and property values, and is primarily used to store user interface information. Two game objects of the same class may need to be displayed differently. They may have different shapes for example. The user interface code knows about the right keys, and looks up the right values. To draw a game object, it would look up the shape property and draw the right shape.

Subclassing ArGameObject

ArGameObject is marked final and not subclassable. Any new capability that is desired for a game object should be added directly into the class. Note that a capability is different from a behavior. An example of a capability is having a game object being able to contain other game objects, as in the case of a bag. An example of a behavior is a game object patrolling between waypoints. As seen before, behavior is implemented by state machines and can vary. However, the capabilities of a game object is always fixed.

4.4.15 Game File Format

The game file will be an XML document whose root element has a child representing the serialized form of an *ArGame* instance. This is sufficient because all game information should be contained in the *ArGame* object. A DTD is not required, since badly formed XML will simply cause the deserialization process to fail.

4.5 Serialization and Deserialization Protocol

As mentioned before, all implementations of the MIT AR client on the various platforms need to implement the same serialization and deserialization protocol in order for them to read the same game files. For this reason, we need to design our own protocol instead of relying on language specific serialization. Clearly, it would be inconvenient to deserialize a serialized Java object in say Objective-C. One might argue that it saves work to simply adhere to one language's serialization protocol. If we do so, we need only worry about implementing serialization and deserialization methods on all platforms except the platform whose language was chosen. However, this idea is dangerous because then we are forced to implement virtually all functionality described by that protocol in question, which is a large task and much more than what is actually needed. Furthermore, we would then have to make changes to our own implementation each time that particular protocol is modified, which may happen as the language evolves over time.

All the objects stored in the game file should extend the *ArSerializable* class, which provides an interface for the serializing the object's value to and from either an XML string or a byte stream. Subclasses must meet the following requirements:

A subclass must have a visible default public constructor It is possible that we will not have all the information required to directly fill all fields of the object in question immediately upon deserialization, depending on the type of XML parser that is being used. It is unreasonable to use a DOM based parser because of memory issues so it is likely that a stream-based or an event-based parser will be used in most platform implementations. We wish to first instantiate a blank copy of an object, and fill in the requisite fields as we encounter them during the deserialization process.

A subclass must have getters and setters for all fields to be serialized Not all of the fields for a class may have public access so getters and setters are required for these. The naming convention for getters and setters should adhere to the standard convention of getFieldName() and setFieldName(). Boolean fields should have a getter names isFieldName(). The reason for the standardized naming convention is to enable support for reflection on platforms where it is available. Also, fields that should not be serialized must be marked in some fashion - this can be done in an external file by providing a map between classes and unserialized fields, or simply by adding a transient keyword if it is available.

4.5.1 Serialization

The serialization process is relatively straightforward. The only complexity during serialization is dealing with object references, which need to be preserved. There may also be cycles in the object graph, and care needs to be taken not to end up in an infinite loop during serialization. To solve this problem, we assign a unique id to each object during serialization and store the object to id pair in a map. Before attempting to serialize an object, we check that map to see if an id has already been assigned. If so, then we do not attempt to serialize the object again, but rather we write a reference representation instead of the full object representation. Primitive types will not be assigned ids. Here we will only describe XML serialization, since the byte stream serialization protocol is an feature for future implementations. Listing 4.3 shows a template for the serialization format.

```
<$Class id='$id'>
  <$Field [[id= '$id'] ref='$refId']>$Value</$Field>
  ...
</$Class>
```

Listing 4.3: XML Serialization Format Template

The serialized XML string of an object will begin with a tag containing its class name. Inherited classes shall write the most specific class, and not provide names of superclasses. The attributes that may be present in the start tag are *id* and *ref.* A *ref* attribute signifies that the object has already been previously serialized and is the same as the object represented by the node with the *id* attribute of the same value.

Serializable fields of the object are represented by child elements of the object's node. The element names are simply the field names themselves. If the field value is an object, the element will have either an *id* or *ref* attribute, and will have child elements representing its fields if any. Lists are dealt with in a special way - the child elements of a list node are named by their class type. Note that the class type for each field an be inferred from the class type of its enclosing class, which is why it is not included in the XML. Also, fields whose values are null are ignored.

We will now discuss an example to demonstrate id usage. Say we have an object of type Foo with a field named bar of type Bar and an object of type Bar with a field named foo of type Foo, and these objects point to each other. The result of serializing the object of type Foo is shown in Listing 4.4:

```
<Foo id='0'>
  <bar id='1'>
    <foo ref='0'/>
  </bar>
</Foo>
```

Listing 4.4: Circular Reference Example

Note how the cycle is broken, and note how the reference representation contains only a reference attribute with no child elements.

4.5.2 Deserialization

Deserialization occurs in the exact reverse of serialization - we store a map from id to object. If the id is present in the map we retrieve the object from the map. Otherwise, we instantiate a new object of the specified class and deserialize the object fully. For the example in Listing 4.4, we'd encounter a Foo type, and note that we have not seen an object for id 0. We then instantiate an empty Foo object and store it in our map. Now we encounter a Bar object that belongs to the field named bar in Foo with id 1. We instantiate an empty Bar object and store it in our map. Now we encounter a Foo object again, but this time the id is in our map, so we simply set the field named foo to be the object mapped to by the id 0. Note that that Foo object is not fully initialized yet at this point of time. Now we are done deserializing the Bar object, so we set the field of the Foo object. Lastly, we encounter the Foo end tag so deserialization is complete.

One scenario that may happen upon deserialization is that we may be attempting to deserialize an old version of the serialized output. For example, say that a class Foo had a field bar before, but in a newer version of the class the field bar has been removed. When we attempt to deserialize this old file, we will encounter a child element named bar, but this is no longer in the class. In scenarios such as these, the deserializer will provide a warning, ignore the said element and its children, but continue parsing the rest of the file. In the converse scenario where there are fewer child elements than there are serializable fields in the class, the fields that were not found are assumed to take default values as defined by the default constructor.

4.5.3 Differences between the old format and the current format

The old serialization format was simply the default XML output of the .NET serialization library. While this format is similar to the one outlined in previous sections, it has some differences. It does not support object references, and led to some amount of object clutter since the same object could be serialized twice and after deserialization we obtain two equal objects instead of the one object that existed before. Also, empty fields were serialized when these could simply be ignored. Lastly, derived types have tags that contain both the name of the derived type and the base type, when this is not required. To summarize, the new serialization format supports object references and has less string clutter, leading to smaller game files and faster deserialization.

4.6 User Interface Specifications

It is not possible to fully describe how the user interface code should be implemented because it is highly platform specific. For example, drawing and repainting are completely different when interacting with the UIKit on the iPhone, and with Swing on Java. However, we will sketch out the behavior of the basic components of the UI system in the following subsections. This specification is not exhaustive and can be changed to accommodate the specific platform implementation.

4.6.1 Primary User Interface Component

The main user interface component contains a tab controller that allows the user to switch between the main map and the history list. Other tabs may be present depending on the type of game that is being played. For example, if a game has global codes, there will be a tab containing a keypad where the user can enter codes.

4.6.2 Map Component

The map component is responsible for constructing the appropriate views for all the game objects in the game and placing them at the right coordinates. It should subscribe to the events for visibility change, player visits, location changes and region changes.

4.6.3 Game Object Map Component

This component aggregates a game object component, drawing the object based on information in the game object's property list. This component is a child of the map component in the component hierarchy.

4.6.4 Player Component

This component aggregates an *ArPlayer* instance and draws it on the map. It is a child of the map component in the component hierarchy.

4.6.5 Game Object Component

This component is used when a game object is visited by a player. It displays the name, description, and actions for that object. It should be able to change itself in the right way depending on what action is selected. For example, when a interview action is selected, it displays the interview content for the object.

4.6.6 History Component

The history view wraps around an *ArPlayer* instance and shows the player's history.

4.7 Backwards Compatibility

Unfortunately, the new framework is sufficiently different that a direct translation between the old and the new models will not be easy. In particular, reading old game files is not possible in this current framework. To provide support for old games, a compatibility layer will have to be introduced at the deserialization stage that will translate old game files correctly in the new framework. Such a layer can indeed be constructed because the old features of the games are still preserved in this new framework. The mapping between the two different sets of code is for the most part one to one, but there are cases where some complications will arise. For example, since we are using references instead of object ids in the new system, we will need to convert all the object ids to references when attempting to convert an old game file to the new format.

However, the actual user interface does not have to change. The underlying model has changed but all the concepts are preserved. The user interface will still look the same to players, but the user interface code now interacts differently with the backend.

4.8 Summary of Changes

The main changes to the framework are listed below:

1. Game file format

2. Dynamic visibility

3. Customizable conditions, events and actions

4. AI for game objects

5. Decoupling of user interface and model

4.9 Examples of Possible New Features

In this section, we will describe new functionality that is enabled by this framework.

4.9.1 Location Based Visibility

Upon entering a certain area of a map, certain objects may be shown or hidden. This can be easily implemented by first subclassing *ArCondition* to create a class *ArLocationCondition* which contains *ArPolygonalLocation* object. The condition should return true if the player is in the hotspot of that location, and false otherwise. An object then sets this *ArLocationCondition* as its visibility condition.

4.9.2 Visiting Hidden Objects

This is a trivial result of the new approach to handling visibility.

4.9.3 Moving Objects

This is a trivial result of using our state machine.

4.9.4 Location Based Bump Probability

A player's location in a certain area of a map results in some likelihood of visiting certain objects. We implement this behavior for each object as follows - we first set the location of the object to be *ArLocationEverywhere* such that the player is always in its hotspot and nudge zone. Then, we set the additional visit and nudge condition of the object to be an instance of an *ArAndCondition* containing an *ArProbabilityCondition* and an *ArLocationCondition* whose condition is met when the player is in the hotspot of the area in question.

4.9.5 Game Over on Visiting Certain Objects

We implement this by subclassing *ArEvent* to create an *ArGameOverEvent.* The condition for that event will then be that the player has visited the object in question. The user interface should listen to the *ArGameOverEvent* and respond as required.

4.9.6 Conditional Interaction with Game Objects

An example of such a feature would be interacting with a dragon NPC. If the player possesses a sword, then the player may choose to defeat the dragon. Upon defeating the dragon, there is now an option to grab the treasure that the dragon is guarding. This behavior can be implemented by first subclassing *ArAction* to create *ArSimple-Action.* We create two instances of this action - the first action's condition is set to be a subclass of *ArCondition* called *ArInventoryCondition*, which checks to see if a given object is in the player's inventory. The second action's condition is set to be a subclass of *ArCondition* called *ArActionCompletedCondition* which aggregrates the first action and whose condition is met when the first action has been completed.

4.9.7 Customizable Sound for Triggering

This is implemented by checking the appropriate property of the property list object that each game object contains.

THIS PAGE INTENTIONALLY LEFT BLANK

Chapter 5

Conclusion and Future Work

In this thesis I started out with implementing a basic game client for the iPhone that supported a subset of the functionality of the Windows Mobile client. At the same time, I defined a means for deploying games via an archived format and a remote game list.

Based on the experience of developing that client, I designed a new architecture that allows greater flexibility and extensibility, while providing several new features. At the same time, this architecture should be easily portable to other platforms because it is user interface independent. The work in migrating to different platforms will primarily be that of designing the particular user interface for that platform.

The new framework has certain implications for the game editor. Although the framework itself is flexible, the game editor should not expose that flexibility to the game designer. For example, the game editor is not capable of creating new actions - it can simply present the game designer a list of actions that can be taken, as well as the parameters of that action. Likewise, the game designer is only presented with a set of events that can be used.

The next steps for the project involve migrating the current code base to the new framework. A branch of the project should be created whereby the new framework will be developed and integrated with existing code. This is relevant only to the Windows Mobile code - the iPhone client can simply be reimplemented given that much of the work on it has been preliminary. Upon completion, this branch can then be merged with the main development of the project.

The first step of the development should be to write the backend code for the .NET platform, which is shared by both the game editor and the game client. Once that is done, the game editor's user interface code should be updated to interact with the new backend. At this point, we will have a game editor capable of creating games in the new file format. This will be useful in testing the revised implementation of the game client which reads that new file format. Next, we update the user interface code of the game client similarly to interact with the new backend. We should now have a functional version of the game client on the Windows Mobile platform that utilizes the new framework. On the iPhone side of things, development will occur similarly - the backend will be completed first, and then the user interface code updated.

Once implementation is complete, some tests should be conducted to measure the performance of this new design. Adjustments to the framework should be unnecessary provided the right optimizations have been made. However, in the unlikely event that the performance is still poor, the manner in which game ticks are handled should be rethought, since that is where most of the processing time is spent. Lastly, the new game features that can now be implemented with this framework can be added to the games.

Bibliography

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

[2] Eric Klopfer. "Augmented Learning." The MIT Press, 2008.

[3] Eric Klopfer, Judy Perry, Kurt Squire and Ming-Fong Jan. "Collaborative Learning through Augmented Reality Role Playing." *Proceedings of Th 2005 Conference on Computer Support For Collaborative Learning: Learning 2005: the Next 10 Years!* (Taipei, Taiwan, May 30 - June 04, 2005). Computer Support for Collaborative Learning. International Society of the Learning Sciences, 311-315.

[4] Eric Klopfer, Kurt Squire and Henry Jenkins. "Environmental Detectives: PDAs as a window into a virtual simulated world." *Wireless and Mobile Technologies in Education, 2002. Proceedings. IEEE International Workshop on* pp. 95-98, 2002

[5] Benjamin Arthur Schmeckpeper. "Outdoor Augmented Reality n-th Game Editing Suite." Massachusetts Institute of Technology, 2007.