# Usability and Game Design: Improving the MITAR Game Editor

by

Robert F. Falconi

B.S, Computer Science, 2008

Massachusetts Institute of Technology



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

February 2010



Author_____

Department of Electrical Engineering and Computer Science
February 2, 2010

Certified by_____

Eric Klopfer
Associate Professor, Department of Urban Studies and Planning
Thesis Supervisor

Accepted by_____

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

Usability and Game Design: Improving the MITAR Game Editor

by

Robert F. Falconi

Submitted to the
Department of Electrical Engineering and Computer Science

February 2, 2010

In Partial Fulfillment of the Requirements for the Degree of
Master of Engineering in Electrical Engineering and Computer Science

## ABSTRACT

Creating MIT Augmented Realty (MITAR) games can be a daunting task. MITAR game designers require a usable game editor to simplify this process. The MITAR Game Editor was the first editor to provide game designers with the means to effectively create MITAR games, however, there were several areas that needed improvement. This motivated the development of several other incarnations of MITAR editor, each with its own unique usability strengths. However, these editors haven't seen much use by the game designers, and so much of the usability research that went into their development has gone to waste. In this project a new MITAR editor, the Full Editor, was developed. It combines the most usable portions of the newer editors together with the features of the original Game Editor into one game development solution. In addition, the Full Editor also introduces a new feature, the Flow View, which increases its usability further. Heuristic analysis and informal testing suggest that the Full Editor is a highly usable MITAR editor that will replace the Game Editor as the primary development platform for MITAR games.

Thesis Supervisor: Eric Klopfer
Title: Associate Professor, Department of Urban Studies and Planning

**Table of Contents**

# 1. Introduction

*In the year 2100, global climate change has become a serious problem for everyone. Unfortunately, it is already much too late to do anything about it. Or is it? Researchers at the TimeLab have discovered that a pivotal election held in Cambridge in 2008 could have changed the course of our history. As a TimeLab researcher, your goal is to go back in time and research laws that will have a positive effect on our environment today, so that they can be put on the 2008 ballot. It is important for these laws to have a positive impact on the environment, but that is not enough. They must also be popular enough to pass in the election. When your research is complete, you will propose the laws that you think should be placed on the ballot. You have been given the chance to battle the effects of global warming and to effect an incredible change on our global climate. So get out there and get to work, and good luck!*

This is the scenario presented to students playing "TimeLab 2100", an augmented reality game developed by the Scheller Teacher Education Program (2003). Armed with hand-held computers with built-in GPS functionality, the students are let loose on the MIT Campus. With the aid of these hand-held devices, the students research laws virtually and learn about climate change, global warming, and city governance. And while they play, they also explore a real world location and interact with each other.

This is just one example of an MIT Augmented Reality game. The Augmented Reality Group at the Scheller Teacher Education Program originally developed the software solutions for the creation of these games. Since then work on the MIT Augmented Reality project has been continued by developers at the Education Arcade. MIT Augmented Reality technologies have already been put to use in several successful educational applications, including "TimeLab

2100," described above. MIT Augmented Reality games are designed to provide teachers with an innovative way to present educational material to their students, as well as provide a fun, interactive way for students to learn.

# 2. Background

This chapter will present the background information upon which this project is based. The first section provides an introduction to the concept of Augmented Reality, and includes some examples of possible applications. The second section focuses on games which make use of Augmented Reality, such as MIT Augmented Reality (MITAR) games. The final section discusses the ways in which MITAR games are created, and provides the motivation for this project.

## 2.1 Augmented Reality

Reality is the world in which we live, the environment around us that exists in actuality. The reader may also be familiar with the concept of Virtual Reality (VR), in which the user is completely immersed in a computer-generated synthetic world (Milgram and Kishino 1994). In VR, information about the virtual world is provided to the user by the computer, and as the user interacts with the virtual world, the computer alters what is presented to the user accordingly. Between reality and VR lie all of the applications which combine elements of the real world with elements from a virtual one. Milgram and Kishino (1994) further divided these Mixed Reality (MR) applications into Augmented Reality (AR), in which virtual elements are added to a real environment, and Augmented Virtuality, in which elements of the real world are added to a synthetic environment. However, the term AR has found common use as a substitute for the more general term MR.

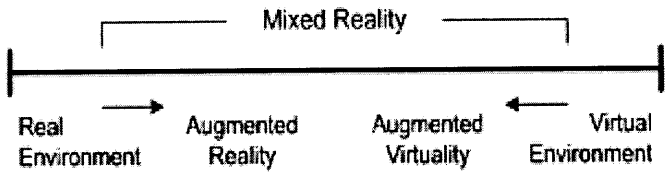


**Figure 2.1 Milgram and Kishino's Virtuality Continuum.** All MR applications, including AR, lie somewhere on the spectrum between realty and virtual reality. (Migram and Kishino 1994)

In his paper, "A Survey of Augmented Reality," Ronald Azuma (1997) defines AR as any technology that combines elements of the real world with computer-generated elements that exist only virtually. Additionally, these two elements must be combined in a way that they can be interacted with in real time. Azuma adds a third condition to his definition: the virtual elements should be rendered in 3-D when combined with the real elements. However, this makes the AR definition a little too restrictive and forces the exclusion of MITAR as an AR application. Therefore, this paper will ignore Azuma's final point about three dimensionality, favoring the broader AR definition that is implied without it.

It is the combination of real and virtual elements, and the many different ways in which this can be accomplished, that makes AR such a powerful and flexible technology. Real world elements provide grounding and familiarity, allowing the user to make use of the knowledge and skills acquired in the real world to interact with the AR application. These elements also provide the contextual information for the application. Virtual elements enhance the way the user perceives and interacts with the world around him, providing the user with information that is not available in the real world. The virtual elements also serve as the motivating force behind the applications, and this is especially true in the case of AR games. When digital and real world information are intertwined, the result is a new interactive experience would not be possible to

recreate in either the real or virtual world alone.

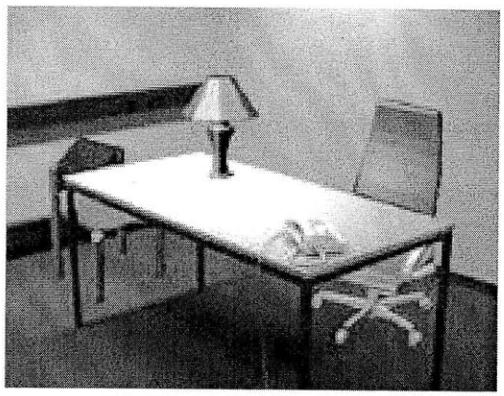

**Figure 2.2 Augmented Reality.** In this simple example of Augmented Reality, a virtual lamp and two virtual chairs are added to real footage of a table in a room.

AR is a flexible technology with a wide variety of applications ranging from simple to complex. Every football broadcast on television uses a simplistic AR system to overlay the yellow "first down" line on the field, for the benefit of the people watching from home (Bosnor 2010). That example is simplistic, but there are other more complex ones. At Columbia University, an architectural AR system was developed which superimposed a graphical representation of the hidden structural systems onto the walls of a given room in a building. This allowed workers to see and take these systems into account before they began renovations (Webster et al. 1996). AR has also been used in the medical field to combine virtual 3-D models of a patient's organs with the real view of the organs during surgery. This technique has already helped to guide surgeons during their procedures (Soler et al. 2009). AR was also instrumental in the filming of James Cameron's "Avatar," as Cameron developed a new AR camera to place his actors into the computer generated world of Pandora. This enabled the director to adjust

camera angles in real time rather than having to review them later on a computer, allowing Cameron to get just the shot he wanted (Thompson 2010).

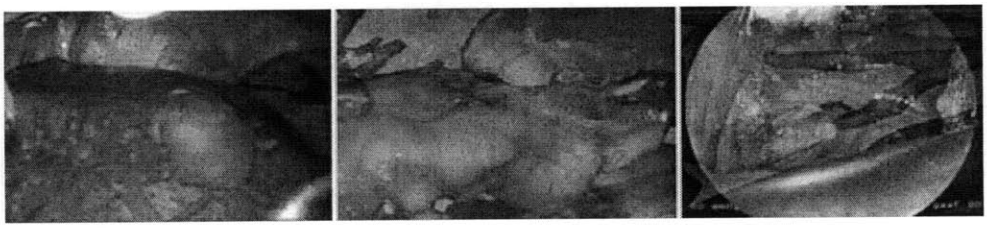

**Figure 2.3 Augmented Reality In Surgery.** An AR system being used in liver (left), pancreas (center), and parathyroid (right) video assisted surgery.

## 2.2 Augmented Reality Games

AR has a wide number of applications across many different fields. This project deals with AR as it applied to MITAR educational games. However, MITAR games are not the only games that make use of this technology. When discussing AR games we can divide the games into two classes, those games which are location-aware and those that are not. Sony's "Eye of Judgment" is a good example of an AR game that is not location-aware (Sony Entertainment America Inc 2007). "Eye of Judgment" is a card-based game in which players position their cards underneath a digital camera. The players then watch as the Playstation 3 displays not only the card but also an interactive, 3-D model of the card's character on the screen. MITAR games belong to the other class of AR games, those that are location-aware. These games make some use of the player's current position, tracking the players' movements though either GPS or some other means. Within the class of location-aware AR games, the games can again be divided into two different categories, those which are heavily augmented and those which are lightly augmented. Heavily augmented AR games are those games which provide the user with constant

access to the virtual elements of the game, usually through the use of a head-mounted-display. "ARQuake", an augmented reality version of the original computer game "Quake", is a heavily augmented AR game, superimposing the game's monsters onto real world locations to allow players to do battle in the real world (Thomas et al. 2000). Lightly augmented AR games refer to those games in which the player is not constantly exposed the virtual elements of the game. MITAR games fall into this category, as the virtual elements are present through the hand-held computing devices, so the can player choose when to access the information.

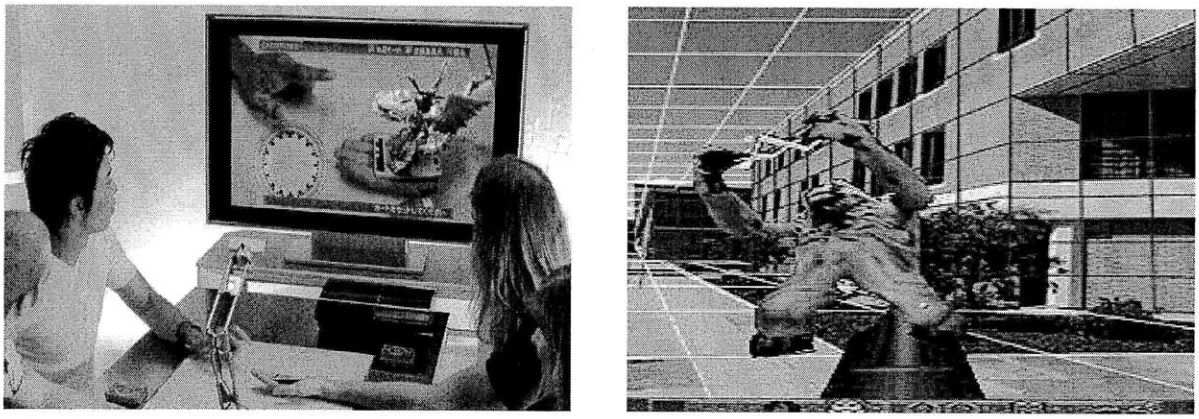

**Figure 2.4 AR Games.** On the left, players experiment with Sony's "Eye of Judgement". On the right, a game of "ARQuake" in action at the University of South Australia campus.

Each MITAR game is a location-aware, lightly augmented game. Each player is given a hand-held device and let loose at a real location, given free reign to roam within the borders of the on-screen map. Each of the hand-devices has GPS capability, allowing the game to track and display the player's movements on the map. As the player walks around in the real world, the player icon mirrors these movements on the map screen. This is only way to move player's character though the game. As the player moves about the game site, the player icon will bump

into icons representing the various objects in the game, granting the player access to the information they store. The player progresses through the game by visiting each of these objects in turn.

Originally, MITAR games were all played outside at a single, real world location. This has recently changed, and support has been added for multiple locations as well as indoor ones. The player can change locations by visiting portal objects, but game-play remains the same for all outdoor locations. When the player enters an indoor location, however, the GPS loses its effectiveness. Indoors, players are forced to tap game objects on the map screen to visit them, rather than moving to the appropriate location.

The physical game site does more than just provide a space for the player to move through. Game designers can choose to incorporate tangible information into the game; information that is available to the player only outside of the device. Allowing the player to uncover this information on site, rather than just presenting it through the hand-held device, can help to immerse the player in the game. In addition, the physical location also provides the context for the game, lending a sense of realism to the game.

At the start of a MITAR game, the player is given a role to play and a goal to achieve, in addition to the hand-held device. In this way MITAR is similar to more traditional role playing games (RPGs). In RPGs, players move their characters around a virtual world from one point of interest to the next (traditionally these are dungeons full of monsters). In the end, the player hopes to achieve some final goal (traditionally some sort of boss battle). In a MITAR game, the player also assumes some role dictated to him by the game. Then, the player directs his icon toward one of the game objects on the map by moving around the game site. When the player

reaches a particular game object, it is presented to him through the hand-held device. This is referred to as visiting the game object. The game objects can be non-player characters, in-game items, or even portals that give the player access to other maps. Each of these objects has its own name, picture and description. These objects may also present the player with important game information in the form of images, video and audio recordings, or written documents, all available for the player to review on his device. Additionally, visiting a game object may cause new game objects to appear or disappear, providing the player with clues about where to go next. Ultimately, the player will use the information he has acquired from each game object, combined with the tangible information gained on location, to complete the final goal.

"Environmental Detectives" (ED) is one example of a MITAR game (Klopfer 2008). It is an environmental mystery game, and the first AR game created by the MIT Teacher Education Program. In ED, players take on the role of environmental engineers investigating a toxic spill that occurred on the MIT campus. The player's goal is to determine the source of the spill and how far it has progressed, research the legal and health problems the spill poses, and make a recommendation for dealing with the spill. To accomplish these goals, players explore the MIT campus. They conduct interviews with experts and witnesses, and collect documents and soil samples, all through their hand-held devices. At the end of the game, the players get together and present their theories and recommendations to their peers, the other players. Along the way, the players are exposed to the field of environmental engineering and learn about scientific investigation.

## 2.3 MITAR Game Creation

In order to play a MITAR game, the player needs two things. First, the player needs the MITAR Game Engine. This program must be installed onto the hand-held device before it can be used to play a MITAR game. The Game Engine does not provide any of the specific data associated with any one MITAR game. It provides the functionality needed to run any MITAR game and the user interface needed to interact with it. Second, the player needs a MITAR game file. These files are specific to each game, and they contain most of the game's content, as well as the locations of all the game's associated media files. In essence, each MITAR game file provides the blueprint and building blocks that the MITAR Game Engine uses to put a specific MITAR game together.

While the MITAR Game Engine was provided by STEP, MITAR game files are not usually created by the MITAR developers. It is left up to the individual game designers to create their games and associated MITAR game files for themselves. This is the reason that the Education Arcade has developed and experimented with additional software solutions to help game designers to create MITAR game. These solutions include the MITAR Game Editor, which is currently the editor most often used to produce and edit MITAR game files, as well as several other similar programs like the Desktop Editor, Remote Editor, and Game Builder.

Originally, creating a MITAR game file required the creation of an XML (Extensible Markup Language) file by hand, in a standard text editor. This XML file had to be populated with all of the game's content. Once the XML file was complete, it could then be used with the MITAR Game Engine to run the game. Forcing game designers to use XML to make MITAR games presented a problem however, as many of these designers were not familiar with the workings of XML. Many times, the game designers would be unable create new games, or even

to add game additional content to existing games, without help. As a result, it would often fall on the MITAR developers to generate the XML for the designers. This was obviously not an ideal situation for either the MITAR developers or the game designers. To make matters worse, the process of populating the XML files by hand was long and tedious, and mistakes were commonly made. It soon became clear to the MITAR developers that new software solutions needed to be developed to facilitate MITAR game creation and abstract away the difficulties inherent in the processes. For this reason, the MITAR Game Editor was developed. The Game Editor provides a graphical user interface that designers can use to develop MITAR games without having to create or edit XML files directly. It contains tools that designers can use to help them find and upload maps, populate the game with objects, edit game content, and much more. The Game Editor hides all of the specifics of creating the XML file away from the designers, making it much easier for less technical users to create and edit MITAR games on their own.

Although the Game Editor does solve a number of problems, it also introduces new complexities. For example, it is only possible to view and edit content for one game object at a time when using the original version of the Game Editor. This is because the Game Editor uses modal dialog boxes to present content to designers. These dialog boxes can present the designer with only one game object at a time. The designer must first finishing editing the currently selected game object and close the dialog box before he moves on to editing the next game object. Another serious problem is the fact that the Game Editor completely lacks mobility. It is impossible to use the Game Editor to change an MITAR game while simultaneously testing it in the field. Additionally, the extensive toolkits offered by the Game Editor can be overwhelming

to new or young users. Not only do these features add levels of complexity to the process of MITAR game development, but they can also obscure the game's details, making it easy for game design bugs to be overlooked. In an attempt to address these issues, several newer versions of the Game Editor software were developed. These are the Desktop Editor, Remote Editor, and Game Builder.

The Desktop Editor was designed to address and solve many of the overall usability problems of the Game Editor (Wang 2008). For example, one of the goals of the Desktop Editor is to allow designers to view content associated with multiple game objects at one time. In the Desktop Editor, all of the game objects and the associated game content for a given chapter are made available through a table that resembles a spreadsheet. By presenting the information in this way, the Desktop Editor provides the designer with a view of the bigger picture of the game, as well as an easy way to edit multiple objects worth of game content and spot check for errors.

By design, every MITAR game is closely tied to the real world environment in which the game takes place. The designer is required to have an intimate knowledge and understanding of the game's location in order to produce a MITAR game that is truly effective. But because both the Game Editor and the Desktop Editor run on PC, all development is forced to happen indoors, wherever the development PC is located. The designer is therefore required to make frequent trips between the development lab and the game's real world location in order to incorporate new game objects or edit old ones based on discoveries made on-location. The Remote Editor was designed to solve this problem (Wang 2008). It is a mobile version of the Game Editor that runs on the same hand-held, GPS-enabled devices that run the Game Engine. The Remote Editor is designed for use in conjunction with the other editors, as its feature set is limited in comparison.

That being said, once one of the other editors is used to create an MITAR game file, that file can be loaded into the Remote Editor on location. Then the Remote Editor can be used to immediately make any necessary changes to the game file while the game is being tested in the field.

Although ideally MITAR games should be tested in the field, this is always the most practical choice, especially in the early stages of game development. The designer needs access to the powerful tools that are only provided by the PC game editors, but would also like to get a sense of how the game will look while it is being played on site. To this end, the MITAR Game Emulator was developed (Schellar Teacher Education Program 2007). The MITAR Game Emulator allows the designer to run the MITAR game on a PC while still designing it. Using the keyboard instead of GPS tracking to move the player icon, the designer can experience the game the way the player would, without having to leave the development PC. The MITAR Game Editor can also follow the game that is being played in the MITAR Emulator, showing the designer different map regions and selected objects as the designer plays through the game. That way, whenever a mistake is found, the MITAR Game Editor is already displaying the appropriate screens to allow for corrections to be made easily.

The Game Editor and Desktop Editor are both very powerful toolkits for professional game designers and computer scientists to use to create MITAR games. However, the complexities that emerge from the expansive list of features offered by these programs can be discouraging to new or younger users. In an effort to combat this problem, a new piece of software, the MITAR Game Builder, was developed. Game Builder is a version of the Game Editor aimed at both younger and inexperienced users. It has a much simpler user interface

which features large, clearly labeled buttons, no tabs, and no menus. The features and options that the Game Builder provides are a subset of those included in the Game Editor. That subset includes those features that the MITAR developers have determined to be the ones most important to and most frequently used in game design. By restricting itself to these features, the Game Builder solves many of the complexity issues present in the Game Editor. This makes it the ideal editor for young and inexperienced users. The Game Builder helps to spark children's interest in game design and provides an introductory platform for novice game designers to slowly accustom themselves to the concepts involved with designing MITAR games.

# 3. Problem Statement

The problem of how to best create a MITAR game file is not an easy one to solve. This is evident from the three different versions of the PC editor that are available: the Game Editor, the Desktop Editor, and the Game Builder. In addition, there is also the Remote Editor and Game Emulator, which serve to supplement these tools. My predecessor on the MITAR editor project, Tiffany Wang (2008), chose to focus on improving the usability of the Game Editor in her designs for the Desktop Editor and Remote Editor. My colleague, Chuan Zhang, helped develop the Game Builder to be a MITAR game file editor focused on ease of use for younger and novice users. Countless improvements and new features have been added to the Game Editor, making it an increasingly more powerful toolkit for MITAR game designers. But even with all of these available technologies, there are still many improvements that can be made from a usability standpoint. But before these improvements can be discussed, first we must establish what exactly is meant by the term "usability".

This chapter will present background for analyzing the problem as well as the proposed solution. The first section discusses the aspects of usability which serve a basis to analyze the various MITAR editors. The second section is a comparison of usability of the different MITAR editors, identifying some the usability strength and weakness inherent in each. Finally, the last section summarizes the problem this project endeavors to solve, and presents the proposed solution.

## 3.1 Usability

The International Standards Organization (ISO) defines usability as "the extent to which

a product can be used by specified users to achieve specified goals with effectiveness, efficiency and satisfaction in a specified context of use" (ISO 9241-11). In *A Practical Guide to Usability Testing*, Redish and Dumas (1999) assert that "usability means that the *people who use the product* can do so *quickly and easily* to accomplish *their own tasks.*" And according to Steve Krug (2000), usability simply means that "that a person of average ability and experience can use the thing for its intended purpose without getting hopelessly frustrated." What this all means is that there are as many different definitions of what exactly usability is as there are people who write about the subject. However, it is generally accepted that there are five major attributes through which it is possible to measure usability. These five attributes are *learnability, memorability, efficiency, errors*, and *satisfaction* (Neilson 2003).

### *3.1.1 Learnability and Memorability*

An application has good learnability when on average new users are able to use the the product to accomplish simple tasks the very first time they encounter it. Similarly, an application has good memorability when users can return to the application after a period of not using it and still be able to accomplish tasks with ease. Making improvements to the learnability and memorability of an application can help new and returning users to get work done quickly and effectively, minimizing time wasted familiarizing oneself to the interface. Learnability and memorability are similar concepts, so many of the same aspects contribute to both the learnability and memorability of a particular interface. One such aspect is good use of affordances. Norman defines affordance as the "actual and perceived properties of a thing" (1988). If the user believes that a particular control should do something, then that is most

likely what the control should do. For example, if a control looks like a button, it has a perceived affordance that it can be pushed and something will happen. If the user cannot push the button, or if he pushes it and nothing happens, the user will become confused. In addition to good affordances, a learnable and memorable interface will have high visibility. This simply means that the relevant controls are obvious to the user or easily found. Finally, an interface should ensure it is controls are consistent both internally and externally. Internally, similar controls should behave in the same way, and controls that look different should behave differently. Externally, the interface itself should appear and behave similarly to other interfaces with the same purpose. When an interface is consistent, the user can apply knowledge gained from using other applications or from using other parts of the interface itself, which eases the learning process (Miller 2009).

### *3.1.2 Efficiency*

Efficiency refers to the speed at which experienced users can use an application to perform their tasks. There are many ways to increase the efficiency of an interface. First of all, in interfaces that use a mouse a primary mode of interaction, the mouse movements themselves can be a source of inefficiency. Mouse movements are governed by Fitt's law, which basically states that the further away from the starting point or the smaller the target of a mouse movement is, the slower the movement will be. Therefore, interface efficiency can be improved through the use of larger controls, or by grouping together controls that are likely to be used in succession. Mouse movements can also be kept to a minimum through the use of keyboard shortcuts and context menus. The use of common default options for certain controls and the saving of

recently used commands in a history can also increase the efficiency of an interface. Finally, an efficient interface will strive to anticipate the needs of the user at any given moment. For example, if the user were to suddenly realize that he needs to correct some mistake based on the information currently displayed by the interface, he should not have to switch to some other part of the interface to make the correction. The interface should present all the information and controls the user needs to complete his tasks in one place (Miller 2009).

### *3.1.3 Errors*

In a usability context, "errors" doesn't simply refer to the number of mistakes the user makes while attempting to accomplish a task. It also refers to the frequency, severity, and ease with which these errors can be detected and corrected. Errors can also be good indicator of problems with the other usability attributes. For example, confusing interfaces with poor learnability properties can lead to high instances of error when new users attempt to perform a task. The majority of the errors that occur when using an interface can be divided into slips and lapses. A slip is an error in execution, such as when the user sets out to do one task but accidentally completes another, usually because the tasks are similar in nature. Users are especially prone to these errors when doing repetitive tasks. A lapse is an error in memory, such as when the user omits a portion of the task or even the task entirely. These errors can be commonly attributed to a lack of attention from the user. While every interface should endeavor to prevent the user from making any mistakes at all, the sad truth is that this is impossible. Therefore wherever mistakes are likely to be made, the interface should be designed so that errors are immediately presented to the user, or at least designed so that the user has an easy

means of error checking. In addition, the interface should provide the means for easily correcting the error once it has been identified. One example of a simple but powerful error correcting measure is the commonly employed "undo / redo" menu option (Miller 2009).

### *3.1.4 Satisfaction*

Satisfaction is the hardest of the usability attributes to define, despite its deceptively straightforward description. Put simply, satisfaction refers to the amount that the user enjoys using an interface. It is closely related to all of the other usability attributes in that interfaces that behave poorly with respect to the other attributes are likely to result in low satisfaction. There are many proposed ways of measuring the satisfaction a user feels when working with a particular interface, but in the end, the surest way to increase user satisfaction is to design the interface with the other usability attributes in mind (Miller 2009).

## 3.2 MITAR Editor Usability Analysis

### *3.2.1 Game Editor*

The Game Editor is the application most MITAR game developers currently use to create their games. Since its creation, the Game Editor has seen the most iterations, bug fixes, and feature additions out of all the other editors. It has been tested over and over again, both in the laboratory as well as in the field. Of the editors discussed, it is also the only one with the complete set of features. All of these things are likely to contribute to the user's overall satisfaction with the Game Editor.

In terms of learnability, the Game Editor is well documented and most of its features are

supplemented with tool tips, which can help to guide new users as they explore interface in an attempt to create games. However, the sheer number of options in the Game Editor can be daunting to a new user, especially one who has never designed an AR game before and does not understand all that it entails. It also cannot be assumed that the new user will choose to refer to the manual. Even in the case where the user does read the documentation, younger users are not likely to be able to understand it completely as it was written with an older, more technically savvy user in mind. Additionally, there are some controls that could potentially confuse a new user. The buttons for adding new non-player characters (NPCs), objects, and portals could cause a problem, as the Game Editor does not provide the user with much feedback after clicking them until the mouse cursor is moved over the game map.

When analyzed from an efficiency standpoint, there are several things that the MITAR game editor does right. The Game Editor does make use of keyboard shortcuts for many commonly used features such as saving the game. There is also a history that stores the most recently loaded MITAR game files for easy loading. Additionally, when the user adds a new game object, the Game Editor fills in all of the data the object needs to function correctly by default. This way, even if the user does not edit the object and populate it with real game content, the object will still function correctly and the game will run on the Game Engine. Despite these features, there are also a few efficiency problems that need attention. The largest of these problems is caused by the Game Editor's use of modal dialog boxes as the primary means of editing a game object's content. A modal dialog box restricts the freedom of the user to interact with an interface. As long as the dialog box is displayed, the rest of the interface will not respond to any input. Essentially, modal dialog boxes exist to force the user to deal with

whatever is contained within the dialog box before he can move on to the next task. In some interfaces this behavior is desirable, as it can prevent inconsistencies from appearing in the application data. However, in the context of the Game Editor, these dialog boxes simply inhibit the user from being able to edit more than game object at time. Depending on the number of game objects that need to be placed in a given game, this method of content population can become a serious bottleneck.

As mentioned before, the MITAR Game Editor has seen extensive use since its creation, and many possible errors are prevented by the interface itself. For example, if there is no game object selected, the controls inside of the Object Properties Panel are disabled, as attempting to edit information for a non-existent game object would definitely crash the application. Additionally, the Game Editor also implements undo / redo functionality so that the user can quickly reverse any unwanted changes made to the game file. However the users of the Game Editor are still prone to making two common errors. The first occurs in game content population. As mentioned above in the efficiency section, the process of adding game content to the objects is restricted by the modal dialog boxes. When more then one object needs to be added at a time, the process can become tedious and repetitive. This leads to slips where game content is copied incorrectly and lapses where game content is accidentally omitted. The second common error occurs when setting game flow properties. Game flow is the way in which the game progresses from object to another. Each object has visibility properties that need to be set correctly for the object to be made available to the player, and these options vary depending on the role, chapter, and team settings. There are also settings that can cause visiting a particular object to trigger other objects to appear or disappear. The way in which the game designer

chooses to set these properties completely determines the way that the game flows from beginning to end, and setting them correctly is an essential part of game development. However, these visibility and trigger settings can only be modified through the Game Editors modal dialog boxes, and as a result the process of setting game flow properties is susceptible to the same lapses and slips that effect content population. Game designers can trigger objects to disappear when they are meant to appear, or trigger the wrong object at the wrong time. In the worst case, some objects might never get triggered to appear at all, and these objects are effectively left out of the game completely. Game flow errors in particular are hard to find as all the objects appear on the map in Game Editor, even though they might not appear when the game is run in the engine. And although triggers are indicated by lines connecting the objects on the map, it is not possible to tell from looking at the map what kind of trigger each line represents. With enough triggers in a game, the map itself becomes a mess of icons and lines that is no use to a game developer looking for game flow errors.

### *3.2.2 Desktop Editor*

The Desktop Editor is a re-imagining of the Game Editor. In her design for the Desktop Editor, Tiffany Wang identified many of the usability issues that were present in the Game Editor at the time, and set out to correct them. The Desktop Editor employed a number of improvements to boost its overall usability in comparison to the Game Editor. The wide range of the changes included the implementation of a start wizard to guide users through the process of creating a new game, the use of more explicit tool tips, and clearer error messages (Wang 2008).

The biggest usability improvements, however, are found in the areas of efficiency and

errors. The feature responsible for these improvements is the game content table. This table is similar to a spreadsheet, both in appearance and function. Each of the game objects takes up a section of the table, and all of the object's associated game content is displayed within its specified rows. The content table provides more than just a view of the game objects and content, however; it provides a convenient way for designers to make changes to this content. Editing is accomplished through the use of a side properties panel which changes depending on the selected cell in table. By circumventing the need for constant opening and closing of modal dialog boxes, the content table provides a way for designers to quickly and efficiently add and edit game objects and content. This is especially true in the case where many new objects need to be added and edited at once. Additionally, having all of the content available in one place makes it easy for designers to check for any mistakes or omissions in the game content. When mistakes are found, the table can be used immediately to make corrections (2008).

But despite the many ways that the Desktop Editor improved on the Game Editor's original design, there are still problems. Just like the Game Editor, the Desktop Editor does not offer a solution to the problem designers face with regard to game flow errors. Additionally, Desktop Editor had done away with the idea of editing content with the modal dialog box altogether, but there were still some users that expected to be able to use this feature. These users were not satisfied with the inability to choose the editing method for themselves. But ultimately, there was really only one reason that the Desktop Editor did not replace Game Editor. The biggest issue with Desktop Editor was that it fell behind the Game Editor in terms of its feature set. As the Desktop Editor was being developed, bug fixes, new features, and general improvements were being made to the Game Editor. The Desktop Editor had been developed as

a completely separate application from the Game Editor. This meant that there was no easy way to incorporate these new Game Editor changes into the Desktop Editor. In order for Desktop Editor to be put to use, all of the work would have to be repeated to add all of these new features. The Desktop Editor would have to be retested, and the users who had already learned to use the Game Editor would have to be taught to use the new interface, or at least take the time to teach it to themselves. These reasons worked against Desktop Editor in the end, and as the Game Editor continued to evolve, the Desktop Editor was left behind, and its usability improvements went unused.

### *3.2.3 Game Builder*

The Game Builder was designed to have better learnability properties than the Game Editor. Its decreased feature set helps to eliminate some of the complexities that are inherent in MITAR game design with the Game Editor. As a result the Game Builder is a more accessible design platform for beginner users, especially when these new users are younger. It contains no menus, and most of its controls are large, clearly labeled buttons. These buttons provide instant feedback when pressed, so the user is never surprised by ambiguous control behavior. Additionally, all of the controls are made clearly visible with the interface, which makes it easy for the user to identify the controls that are needed to preform a particular task.

The Game Builder is not a completely stand-alone product, as it can only build a MITAR game up from an existing game template. The game template contains some of the more complicated game settings that need to be set for a MITAR game to function, but were too complicated to expect younger users to set themselves. For example, the game template already

contains all of the necessary map settings. Since game templates must be exported from the Game Editor, the Game Builder cannot function completely independently. However, once a template is provided, fully functioning MITAR games can be produced with the Game Builder.

### *3.2.4 Remote Editor*

The Remote Editor was developed as a way to address the efficiency problems caused by the fact that the Game Editor is not mobile. Being that MITAR games have such strong ties to their real world locations, it is inconvenient and inefficient to force game designers to travel back and forth between the game site and the development location just to test and correct the game. The Remote Editor solves this problem by providing the developer with a simplified editor that works in the field. This way designers can immediately make changes when they test the games on site, instead of having to remember the changes and make them later in the Game Editor.

Remote Editor is similar to Game Builder in that it is not meant as a stand-alone product. The Remote Editor is not feature complete, as it was only meant to allow designers to make quick, simple changes in the field. More complicated changes necessitate the use of the Game Editor. In addition, the Remote Editor cannot create new game files, it can only edit existing ones. The Remote Editor is meant to be a supplement to the Game Editor rather than a replacement.

## 3.3 Problem and Proposed Solution

Each of the MITAR editors has its own advantages in terms of its feature set and its overall usability. Despite this fact, the Game Editor is currently the MITAR editor most

frequently used for game development. As a result, game designers are not able to take advantage of all the research that was previously done to improve the Game Editor's usability. Part of the problem lies in the fact that there are multiple editors, and that not all of the editors can be used interchangeably.

The proposed solution was the development of a new MITAR game editor, the Full Editor. When the problem is the existence of too many editors, it might seem counterintuitive to solve the problem with another editor. However, the Full Editor was not designed as a completely new application. Instead, it was built upon the existing framework of the Game Editor, designed from the very beginning to eventually replace the current version. The Full Editor combines the most usable features of the Game Editor and Desktop Editors together into one package. As the Game Builder and Remote Editor were already serving as supplements to the Game Editor rather than alternatives, they were not considered in the Full Editor design. However, effort was made to ensure these applications would continue to work correctly with the Full Editor. Beyond simply incorporating the most usable elements of the Desktop and Game Editors, the Full Editor also includes a novel solution to the problem of identifying and correcting game flow errors. This was an important problem to solve, as it was an issue prevalent in all previous incarnations of the MITAR editor. Ultimately, the Full Editor is a much more usable version of the Game Editor, combining into one interface the best features of the Desktop Editor and Game Editor with original usability solutions.

# 4. Design

This chapter describes the design process behind the development of the Full Editor. As the Full Editor incorporates existing elements from both the Game Editor and the Desktop Editor, the bulk of the design effort was expended on the new Flow View. As the name implies, Flow View is a new feature designed to address game flow errors. The following sections detail the steps involved in conceiving the Flow View, highlighting the various iterations it went through along the way.

## 4.1 Flow View Requirements

The Flow View was conceived as a tool to provide MITAR game designers with a means to examine their game's general progression. This progression is dictated by way that game objects appear and disappear from the game map. Not all game objects are presented to every player at one time. The game objects that a player sees at the beginning of the game are dependent on the role the player has assumed. When the game features different teams, the team a player belongs to also has an effect on whether or not a particular game object is visible. Game objects also appear and disappear as chapters change throughout the game. Certain objects will only appear after the player finds a special hint either somewhere else within the game or even hidden at the game site. These hints are simple number sequences referred to as *clue codes*. Clue-coded game objects will appear only after the player has found the code and entered it into his hand-held device correctly. Finally, the act of visiting visible game objects will cause other objects to appear and disappear. A *trigger* is game rule that causes one object to appear when a specific object game object is visited. We will refer to the object that causes the trigger to fire as

the firing object. An *anti-trigger* cause a game object to disappear when the firing object is visited. In addition to these two rules, there is another rule that can cause an object to be automatically visited some time after the firing object is visited by the player. These objects are referred to as *delayed objects*, a reference to the time delay between the visitation of the firing object and the automatic visitation of the target object.

Keeping in mind the game flow dynamics within a given MITAR game, the requirements for a effective Flow View emerged. Most importantly, the Flow View needed to provide a clear, concise visualization of the game objects and the various rules that governed their appearance and disappearance. This visualization is what game designers would use to analyze the game flow and check for slips and lapses in the game flow design. The Flow View also needed to provide the designer with a view of the overall flow of the entire game, while still allowing him to focus in on specific role and chapter settings. Finally, in order for the Flow View to be of most use to designers, it needed to provide them with a quick, efficient way to correct game flow errors as they were found.

## 4.2 Design Sketches

Drawing sketches of the proposed components is often the first step in the design of a usable computer interface. These sketches can be completed quickly and easily, and as a result the interface designer can explore several different options with very little commitment to any of them. Sketching leaves the interface designer open to changes and improvements on the interface design, rather than locking him into his original design. This is how the design process began for the Flow View; with a number of sketches that were presented to the MITAR

developers.

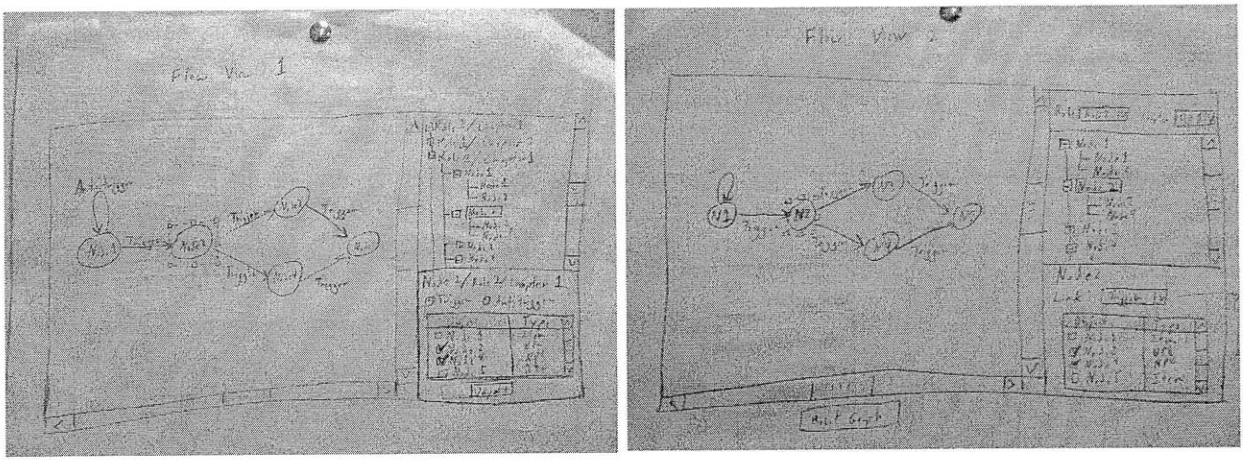

**Figure 4.1 Flow View Sketches.** Two early sketches depicting possible designs for the Flow View. Flow View Design 1 is on the left and Flow View Design 2 is on the right.

It was decided very early on in the design process that the easiest way to for the Flow View to visualize the game flow would be make use of flow chart diagrams. Individual game objects could be simply represented by the nodes of the graph. The various triggering rules (triggers, anti-triggers, delayed objects) lent themselves nicely to representation by arrows that point from the firing object to target object. The flow diagram would provide the game designer with an intuitive way to trace the game flow from one object to the next. For this reason, the flow diagram very quickly became a permanent part of the Flow View design.

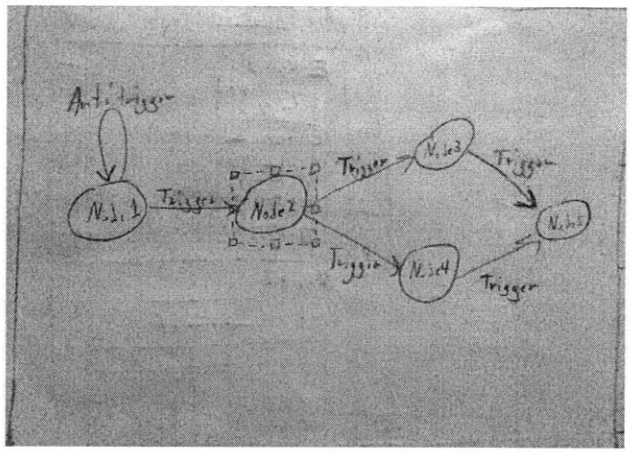


**Figure 4.2 Flow Chart Sketch.** Sketch depicting the early design for the flow diagram

It was clear from the start that the flow diagram would become an integral part of the Flow View. The right designs for the rest of the Flow View were not as clear in the beginning. For example, Figure 3.3 depicts two possible designs for the Flow View side panel. The top of the side panel was meant to provide the game designer with an overview of the triggering rules. The bottom portion would contain the controls that the game designer would use to make changes to these rules. Flow View Design 1 (FVD1) and Flow View Design 2 (FVD2) were conceived with different implementations of this side panel. In FVD1, the trigger relationships are grouped together by role and chapter in the overview section, and so the user has access to the triggering rules for all role/chapter combinations at once. In FVD2, only one role/chapter combination is visible at a time, and the currently selected role and chapter are controlled via combo boxes positioned at the top of the side panel. There are also other differences in the way the user edits the triggering rules. In FVD1, changes do not take effect until the user clicks on the update button. These same changes occur automatically in FVD2.

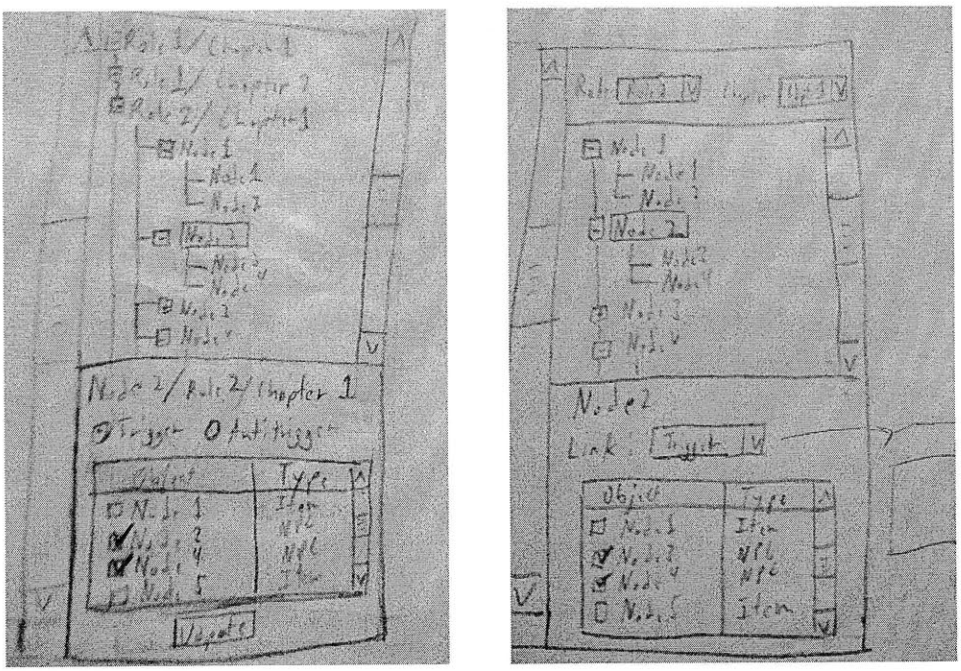

**Figure 4.3 Side Panel Sketches.** The side panel on the left belongs to Flow View Design 1, while the side panel on the right belongs to Flow View Design 2.

Several other design sketches were created during this stage of the design process. These sketches were concerned with various design issues, such as the best way to display clue-coded objects within the flow diagram, or how the Flow View should handle the case where the triggering rules for all role/chapter combinations need to be displayed at once. Usability evaluations on each of the sketches were performed and the other MITAR developers were consulted, and it was eventually decided that FVD2 was a good initial design for the Flow View. To confirm this, mock-ups of how FVD2 would display an existing game were drawn. When this was done, it was time to move on to the first prototyping stage.

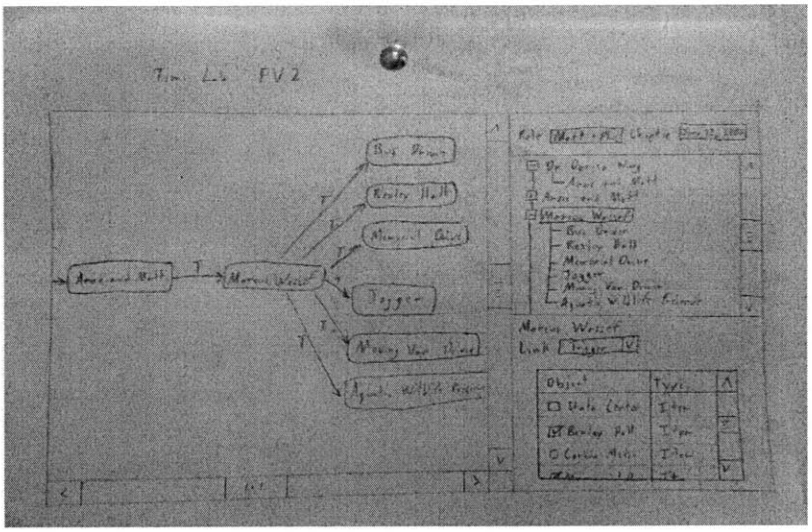

**Figure 4.4 Time Lab 2100 Flow View Mock Up.** This sketch depicts "Time Lab 2100" the way it would be displayed in Flow View Design 2.

## 4.3 Paper Prototypes

Next came the creation of Flow View prototypes. Prototypes are faster and easier to build than fully functional interfaces, so they were the next logical step in the design process. A paper prototype can help interface designers to get an idea of how well their interface behaves without the time investment that comes with writing code. Similar to a design sketch, paper prototypes can involve a lot of drawing of components. The big difference, however, is that paper prototypes can be interacted with, while sketches cannot. In a paper prototype, the interface designer simulates the intended functionality of the interface, filling in for the computer. A little piece of paper with menu items written in can be used to represent a context menu. One sketch can be switched out for another to represent the changing of a tab selection. In this way, paper prototypes provide a easy way for interface designers to test out an interface before actually implementing it.

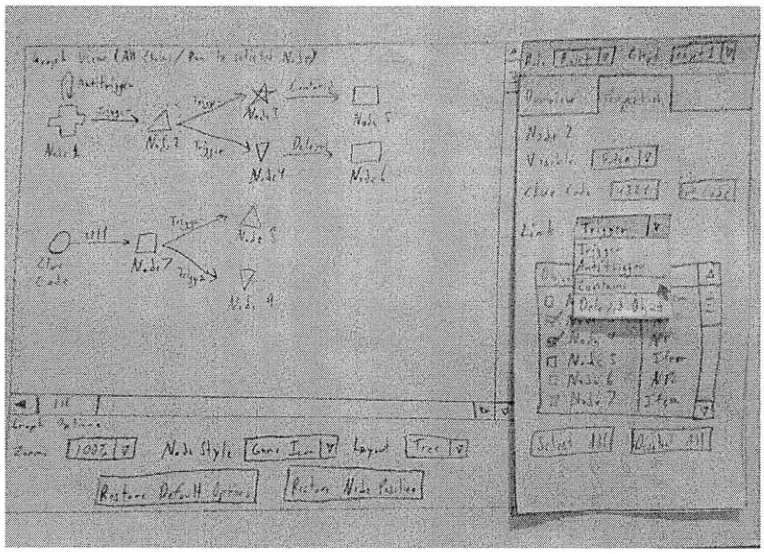

**Figure 4.5 Flow View Paper Prototype.** Paper prototype of the Flow View. The selection of a tab or combo box can be simulating using other pieces of paper, as shown.

Several changes were made to the Flow View in transitioning from the design sketches to the paper prototype. Several new controls were added to the flow diagram, to make it easier for game designers to interact with it. Included among these options were controls for the zoom level, the type of node displayed, and the diagram layout. Buttons to restore these flow diagram options to their default settings was also added. Additionally, there was a button to restore the flow diagram's nodes to their original positions, as by this time it was decided that the game designer would be given the ability to move the individual nodes around within the flow diagram.

The other big change was the separation of the original side panel into two different panels contained within a tab control. The overview panel would contain the same tree-list overview of the triggering rules, but in addition it would also store a list of objects that were inaccessible in the game. The inaccessible list would allow designers to quickly see which

objects were accidentally left out of the game flow, which would in turn cut down on the total number of lapse errors. The other panel was the properties panel, and it would contain all controls needed to edit the visibility and trigger rules of a given object. In addition to controlling visibility, the game designer would also be able to use this panel to make an object clue-coded. Finally, the properties panel would list of check boxes for the game designer to use to add or delete triggers, anti-triggers, or delay rules depending on which triggering rule was selected.

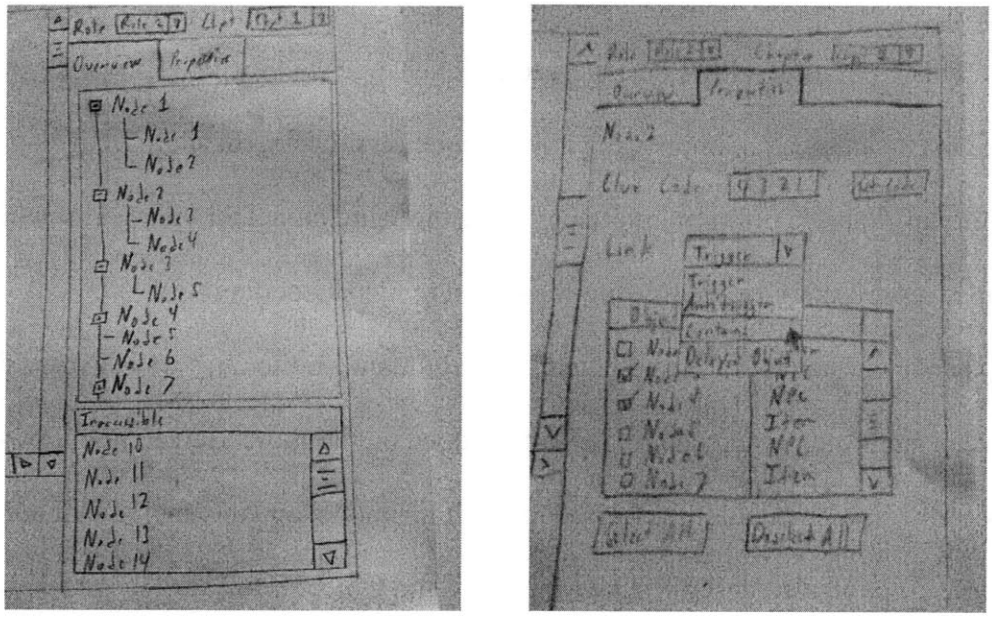

**Figure 4.6 Overview and Properties Panels .** The overview panel, on the left, and the properties panel, on the right, replace the side panel from Flow View design sketches.

# 5. Implementation

In application design, it is good practice to divide the implementation into two parts, the model and the interface. The model contains all of the code that runs in the background. It stores all the information that the application needs to run. The model also contains all of the code needed to make changes to the application data. The interface contains all the visual components that display the application data to the user. Its also has all of the controls that allow the user to interact with the data, and logic that determines how the application should react to user input. This separation allows designers to make changes to the model and interface code separately, which can greatly simplify the implementation process. In addition, it also allows for the possibility of multiple different interfaces using the same model, or vice-versa, which is convenient as it prevents time from being wasted on redundant coding.

This chapter describes the implementation of the Full Editor. As was the case during the design step, most of the implementation effort centered on the Flow View. The following sections describe implementation processes for both the model and the interface of the Flow View. The last section also describes the Map View and Object Info View, the other two major components of the MITAR Full Editor.

## 5.1 Flow View Model

The overall model that drives the Full Editor is a collection of classes contained with a code library known as the CommonLib. The model on which the Flow View depends is separate from the CommonLib and has seen two iterations. Originally, the Flow View used what will be referred to as the graph model. This model depended on two main object classes, the Graph and

the Flow. The Graph served as storage for Graph Nodes and Graph Edges. For each game object, the Graph contained a corresponding Graph Node. Each of these nodes contained a copy of the important information of its associated game object, such as the game object's id number or the role / chapter conditions under which it would be visible. For every triggering rule, there was also existed a Graph Edge within the Graph. Similar to the Graph Nodes, these Graph Edges contained copies of the important information of the associated triggering rule, such as the id numbers of the firing and targeted objects, as well as the type of triggering rule, and so on. The Graph was the sum of all the objects and triggering rules within the game and it was not dependent on any one particular role / chapter combination. In addition to its lists of Graph Nodes and Graph Edges, the Graph contained a method to create its nodes and edges from the classes within CommonLib and a method to produce Flow objects.

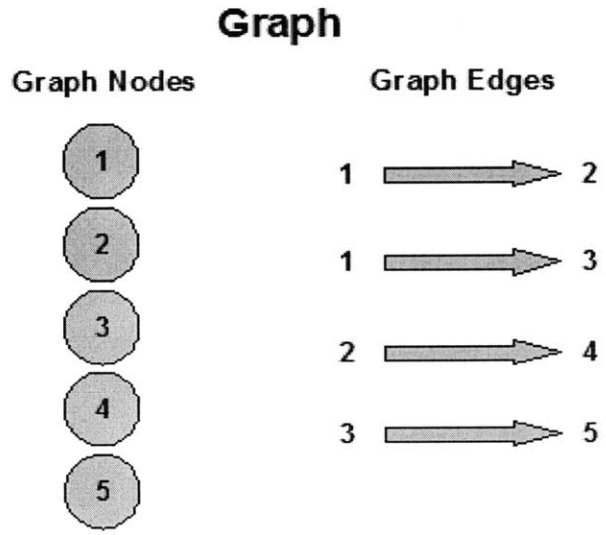


**Figure 5.1 Graph.** This is a representation of the Graph object. Graph Nodes and Graph Edges were stored separately as there was no need to traverse from one Graph Node to another.

Flows were the objects that the Flow View interface actually relied upon. Flow objects

were meant to be like slices of the graph, containing only the information specific to a particular role / chapter combination. Flows contained a list of Flow Nodes. These nodes contained similar information to their counterparts in the Graph, with one exception. For every triggering rule that applied to the Flow's specific role / chapter combination, the Flow Nodes for the firing and targeted game objects contained pointers to each other. This pointers were sorted by the type of triggering rule, which meant there was no need for specific edge objects in the Flow. The Flows also contained a list of root Flow Nodes. These roots were nodes that were initially visible in the role / chapter. By starting with a root node and following the pointers for each triggering rule, one could create a tree of all the game objects that were accessible, as well as identify those that were not.

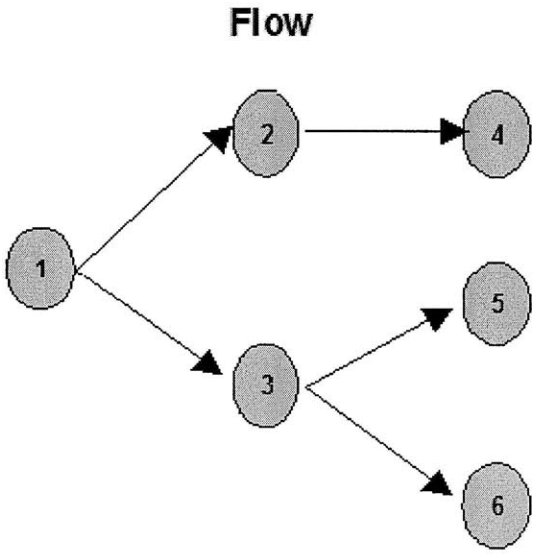


**Figure 5.2 Flow.** This is a representation of the Flow object. Each circle represents a Flow Node. In this example, node 1 is the only root. Flow Nodes can be traced from the root on to form a tree that describes the game flow.

The graph model was implemented for the purpose of providing an additional layer of

abstraction between the classes in the CommonLib and the Flow View interface. In the event that changes would have to be made to the CommonLib, only the Graph object would be affected and need modification. As long as it still produced that same Flow objects, the Flow View interface would still be able to function correctly. However, following a code review of the graph model, it was realized that changes to CommonLib were not made very often and so additional layers of abstraction were unnecessary. In addition, there were problems that could not be easily solved within framework of the graph model. For instance, whenever a user would make changes to the game flow, those changes would only be present in the Flow object. At some point, those changes need to be absorbed into the Graph, and from the Graph they would need to be passed on to the classes in CommonLib. The problem of when and how to do this was not a simple one. It would be much easier to have the changes directly reflected in the CommonLib classes as they were being made. This idea is what motivated the creation of the current Flow View model, the flow model.

Within the flow model framework, the Graph object no longer exists. The Graph's main purpose was to provide an extra layer of abstraction, but that was no longer an issue. The Graph was therefore replaced by an object called the Flow Factory. The Flow Factory does not store any data the way the Graph did. Its sole purpose is to create a Flow object using data from the CommonLib classes when passed a specific role / chapter combination. The Flow object remained pretty much the same as in the graph model, storing Flow Nodes and roots. The biggest changes were made to the Flow Nodes. Flow Nodes no longer contained copies of the information from within the game objects. Instead they now have direct access to the game object's data fields. This means that when changes are made to the Flow Nodes, these changes

are immediately available from inside the CommonLib classes. Additionally, instead of storing pointers to other nodes to represent triggering rules, the Flow Nodes now contained pointers a new object, the Flow Edge. These Flow Edges had direct access to all the data fields of the associated triggering rule, and also stored pointers to the firing and target Flow Nodes. The Flow Edges ensure that changes made to triggering rules are instantly reflected within the CommonLib classes. They also maintain the ability to traverse the tree of accessible Flow Nodes in a given role / chapter combination, starting from any of the root nodes.

## 5.2 Flow View Interface

The Flow View interface consists of the three major components: the Flow Diagram, the Flow Overview Panel, and the Flow Properties Panel. These three components are positioned together on on a single tab in the Full Editor. This section will describe the features included in each of these components.

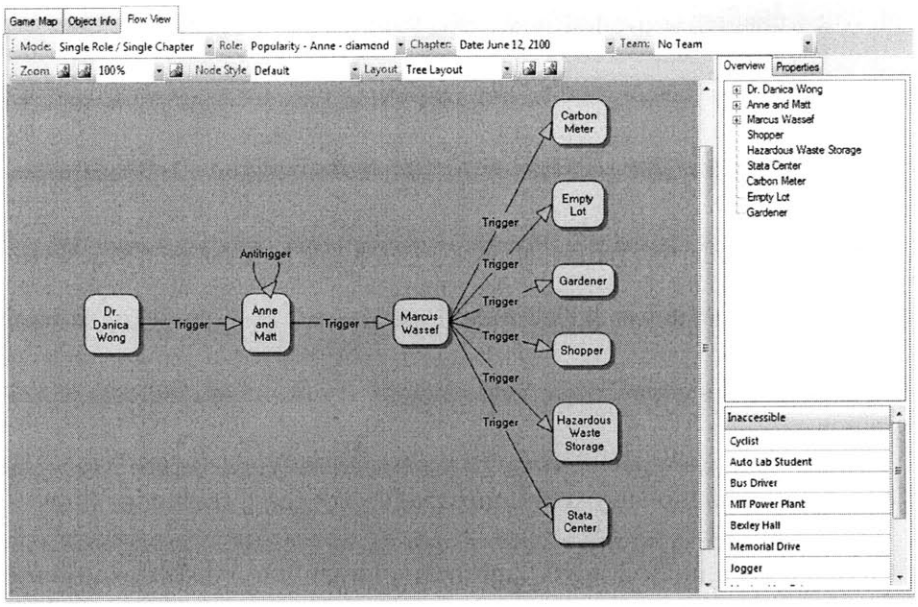


**Figure 5.3 Flow View.** The final implementation of the Flow View. Most prominently featured is the Flow Diagram in the center. Off on the right side, a tab control contains the Flow Overview Panel and the Flow Properties Panel.

Before moving on to three major components, the top-most tool bar warrants some discussion. By using the role, chapter, and team combo boxes contained therein, the game designer can control exactly which combination of the role / chapter and team settings the Flow View is displaying. In addition, this is where the designer can change the Flow View display mode from "Single Role / Single Chapter" (single mode) to "All Roles / All Chapters" (all mode). In single mode, Flow View is dependent on the selected role / chapter combination. The things the Flow View displays and any changes made concern only the selected role and chapter. In contrast, when the Flow View is put into all mode, individual roles and chapters are no longer selectable. The Flow View will display the combination of information from across all the roles and chapters. For example, every triggering rule will be displayed in the Flow Diagram, regardless of its role / chapter conditions. All mode also causes any changes to the game flow to

be applied to all role / chapter combinations.

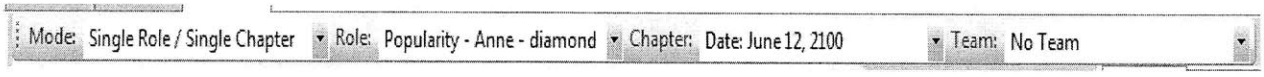


**Figure 5.4 Mode Tool Bar.** This tool bar is to switch back and forth between a single and all mode. It also allows the game designer to the set role, chapter and team selection for the entire Flow View.

### *5.2.1 Diagram View*

The Flow Diagram is the largest and most visible of the three major Flow View components. As in the design, the Flow Diagram implementation displays a flow chart representation of the game flow. Each node in the tree represents a game object, and the name of the game object is visible within the node. The arrows stand in for the triggering rules and are labeled with the triggering rule type. The only exception is for the arrows that represent a clue-code relationship, as these are labeled with the actual numeric clue code. The Flow Diagram can display more than one tree of nodes at a time, as it is possible to have completely separate paths though a given game. At the head of each tree is a root node which can either be a game object that is visible initially visible to the role / chapter combination, or the Clue Code object. The Clue Code object is not an object that is accessible in any of the games; it simply serves as the root node for all the game objects that are made visible via clue code. The Flow Diagram is the component which provides the game designer with the ability to review the flow of his game and also to perform spot checks for triggering rules that are set incorrectly.

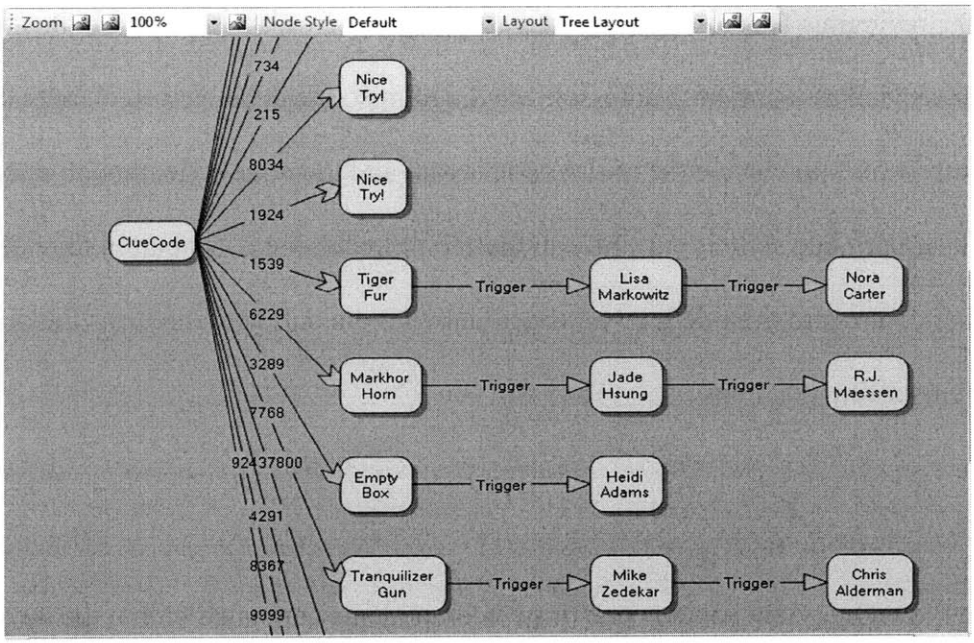


**Figure 5.5 Flow Diagram.** The "Zoo Scene Investigators" (ZSI) MITAR game is displayed in the Flow Diagram. ZSI contains a large number of clue-coded objects, some of which can be seen here along with the actual clue codes.

The Flow View does more than just simply display the game nodes and triggering rules. The game designer can also interact with the Flow Diagram in a number of ways. First of all, each of the nodes can be selected and moved around the diagram by the designer. If the designer finds that moving the nodes around has made the diagram too hard to read, there is a button on the Flow Diagram tool bar, which sits just above the diagram, which returns the nodes to their original positions. In addition, the game designer can left click any node to bring up a context menu. This menu allows the designer to bring up the original game properties model dialog box, which he can then use to make changes to the game object. Double clicking any node has the same effect. Once the changes have been made and the dialog box is closed, these changes are immediately reflected in the Flow Diagram, as well as throughout the rest of the Flow View. In addition to these direct diagram interactions, the designer can also use the Flow Diagram tool bar

to control the diagram. This tool bar includes controls to increase and decrease the zoom level of the diagram, as well as a button to auto-size the diagram so it can be viewed all at once. There are also combo boxes to change the initial node layout and the node style, though only the default rectangular node style is currently supported. Finally, there is a button to restore the other tool bar controls to their default values, along with the button that returns the diagram nodes to their starting positions.

### *5.2.2 Flow Overview Panel*

The Flow Overview Panel is one of two components contained within the tab control to the right of the Flow Diagram. The top portion provides the designer with a tree-list overview of all the game objects which are accessible given the current role / chapter selection. Every game object that appears in the Flow Diagram also has a corresponding entry in the tree-list overview. Expanding any one of these entries reveals a list of the game object which it triggers. This can be useful for the designer to quickly look up all the objects targeted in the triggering rules of any one object. The bottom portion of the Overview Panel contains the Inaccessible Object List. This list displays all of the objects that are not a part of the game flow for the selected role / chapter combination. When the Flow View is set to all mode, the Inaccessible Object List will display all the game objects that are not visible in any of the role / chapter combinations, and so they are left out of the game flow completely. Since it is pointless for a game to contain objects that can never be visited under any circumstance, the Inaccessible Object List provides an easy method to check for lapse errors when used in this way. Even when the Flow View is in single mode, the Inaccessible Object List is still a useful tool for spotting errors in the game flow of

specific role / chapter combinations.

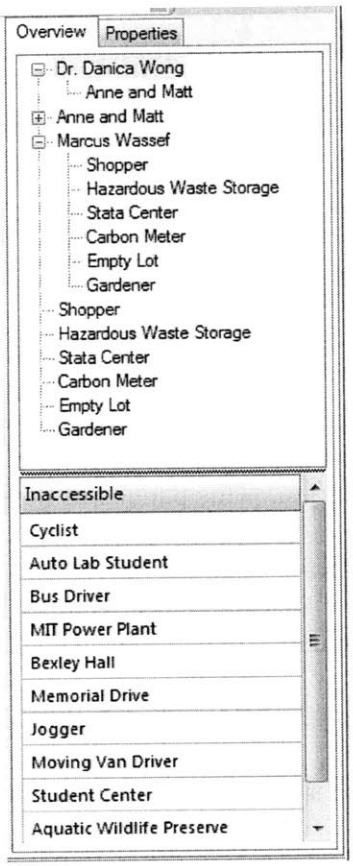


**Figure 5.6 Flow Overview Panel.** "TimeLab 2100" displayed in the Flow Overview. Here the objects targeted by Dr. Danica Wong and Marcus Wassef are displayed. In addition, the Inaccessible Objects List is filled with objects that won't be available to the player under the current role / chapter conditions.

There are a couple of simple ways in which the designer can interact with the Flow Overview Panel. First, the game designer can select an object from either the tree-list or the Inaccessible Object List. This causes the corresponding object to be selected in one or both of the Flow Diagram and the Flow Properties Panel (depending on whether or not the selected object was accessible, there may or may not be a node within the Flow Diagram to highlight). All three of the major Flow View components share this feature; selecting one component causes

the object to be selected in other components. However, it is easiest to take advantage of this feature in the Flow Overview Panel, as it is here that all of the game objects are listed concisely. The Flow Overview Panel also includes a context menu that can be used to bring up the properties dialog box for the selected object, should the designer desire to make changes to the game object in that way.

### *5.2.3 Flow Properties Panel*

The Flow Properties Panel is the second of the two components contained within the tab control to the right of the Flow Diagram. This panel was designed to provide quick, efficient access to those game object properties that directly affect the game flow. This panel contains controls that allow the designer to change a game object's visibility, set a clue-code, or add and remove triggering rules. Of course, game designers could still choose to use the game object property dialog boxes to make these same changes, but the Flow Property Panel is meant to offer a more efficient alternative. The combo box at the top of the panel displays the currently selected game object, the object that will be affected if any changes are made. This combo box contains all of the game objects, so the designer can change the selected game object without having to use the Flow Diagram or Flow Overview Panel. Below the combo box is a message box which displays a reminder of what mode the Flow View is currently in, and describes how changes made in the Flow Properties Panel will affect the overall game flow. The next two controls are used to change the visibility of a game object and to set or remove a clue code from the object, respectively. The final control is a combo box coupled with a grid view filled with checkable game objects. This is where the designer makes changes to the triggering rules. The

combo box is used to select the desired triggering rule (either trigger, anti-trigger, or delayed), and then the rules can be added or deleted by checking or un-checking the box next to the desired target object.

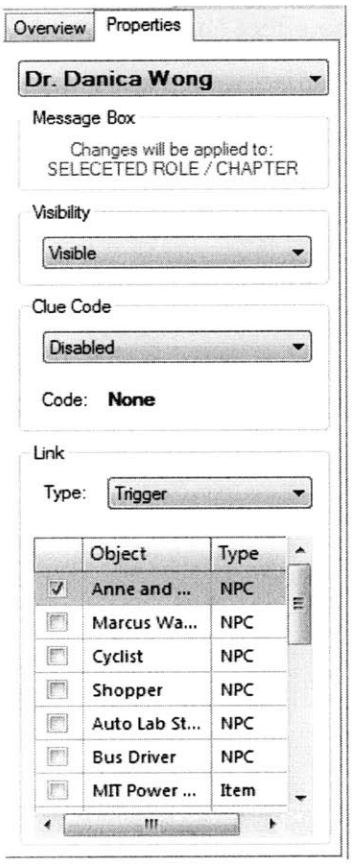


**Figure 5.7 Flow Properties Panel.** The Flow Properties Panel displaying the game object properties of an NPC, Dr. Danica Wong, from "TimeLab 2100".

There are some significant changes that occur in the Flow Properties Panel when the designer switches from single mode to all mode. As mentioned above, the message box will change to warn the designer that any changes will be applied to the game object in all possible role / chapter combinations. In addition, if the selected object is visible in some role / chapter

combinations but not others, the visibility combo box will display "Sometimes Visible" to indicate this. For any triggering rules that are not specifically applied across all roles and chapters, a small square will appear inside the check box next to the target object. Any target objects with checks next to them indicate triggering rules that were specifically applied to all roles and chapters. All mode has no effect on the clue code controls, as clue codes are not role / chapter specific.

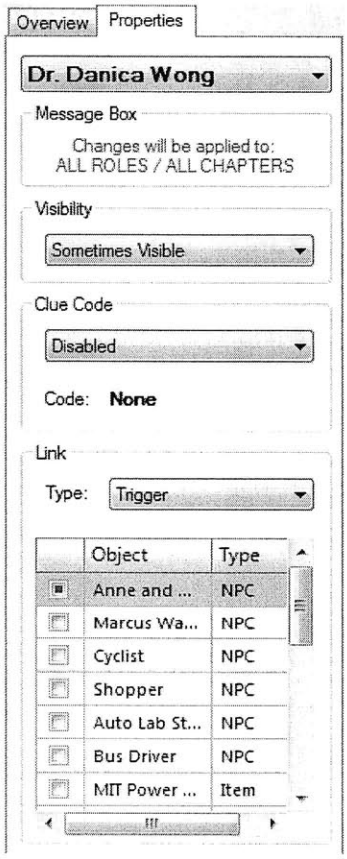


**Figure 5.8 Flow Properties Panel, All Mode.** The Flow Properties Panel displaying the game object properties of an NPC, Dr. Danica Wong, with all mode on.

## 5.3 Full Editor

The Full Editor is split up up into three separate views: the Map View, the Object Info View, and the Flow View. The Flow View was already described in detail in the sections above. This section discusses the rest of the the Full Editor, the Map View and the Object Info View, the components that were borrowed from the Game Editor and the Desktop Editor respectively.

The Object Info View contains the content table and its controls, as designed and implemented by Tiffany Wang for the Desktop Editor. The content table itself resembles a spreadsheet, with two or more rows of space devoted to each game object. The left half of the content table displays simple game object data that remains the same across all role / chapter combinations, like the object's name and description. The right half of the table displays the game object's information pages and stored documents. Because pages and documents are role / chapter specific, the content table contains a column for each of the game's roles. The selected chapter can be changed using a combo box in the tool bar that sits above the content table. This arrangement allows pages and documents to be added to the game efficiently, as the game designer can modify these properties for all roles at once, given a particular chapter selection. Edits are made to the game object properties through the use of a panel which sits off to the right side of the content table. This panel changes to reflect the game properties displayed in the cell that is currently selected on the content table, providing controls specific to those properties. The Object Info View also features a tool bar that contains controls to add and remove game objects, add and remove information pages, and apply content changes across all roles or all chapters.

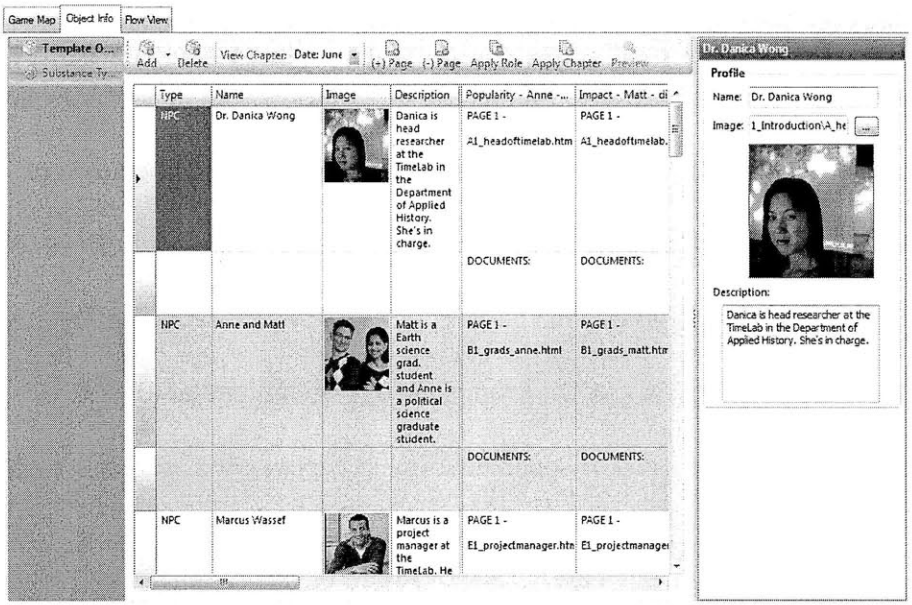


**Figure 5.9 Object Info View.** The Info View displaying the properties of all the game objects in "TimeLab 2100".

The Map View encompasses all of the same controls and displays found within the Game Editor. The prominently featured game map, the various tool bars, and the tab controlled side panel are all the same as their counter parts in the Game Editor. The Full Editor contains the complete set of features and includes of all the latest bug fixes and improvements made to the Game Editor. The Map View's control layout is exactly same as the Game Editor's, so game designers who switch from the Game Editor to the Full Editor will find that they are already familiar with the Map View. This gives the Full Editor an advantage in terms of learnability and memorability, as game designers can apply the skills they learned using the Game Editor directly to the Map View in the Full Editor. There is one addition to the Map View that is not included in the Game Editor. In the lower right corner of the Map View sits the Stray Objects box. This control stores any new game objects created in the Object Info View. Since the game designer cannot use the Object Info View to specify where on the game map new game objects should be

located, these game objects are instead displayed within the Stray Objects box. By selecting one of these stray objects and clicking the "Add to Map" button, the game designer can move the object out of the Stray Object box and place it in the center of the game map. From there, the game designer can move the object to the desired location on the map.

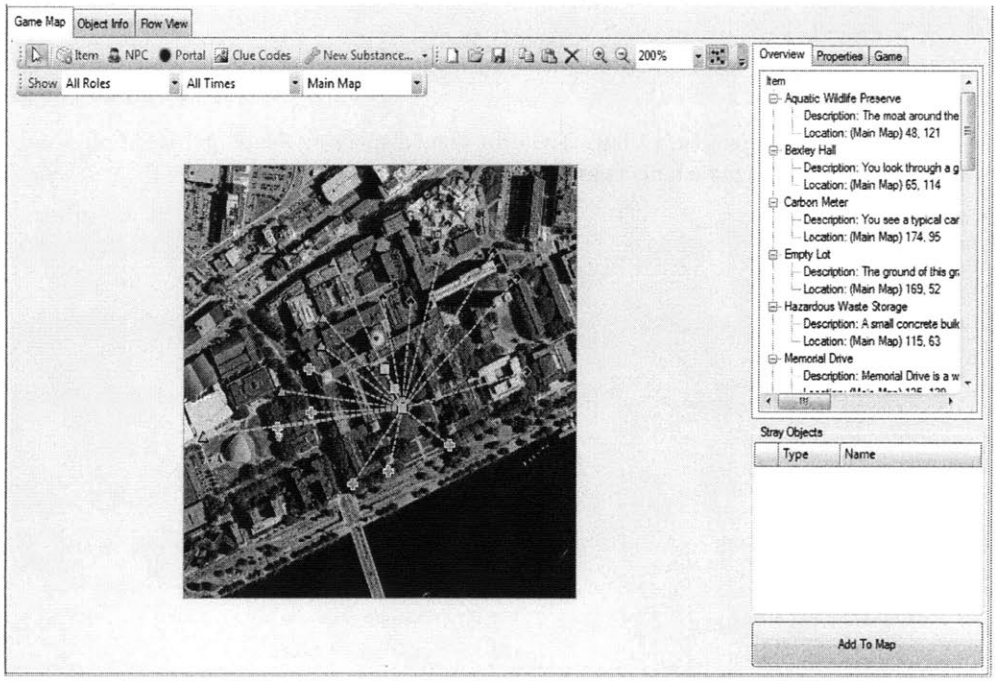


**Figure 5.10 Map View.** The Map View displaying "TimeLab 2100".

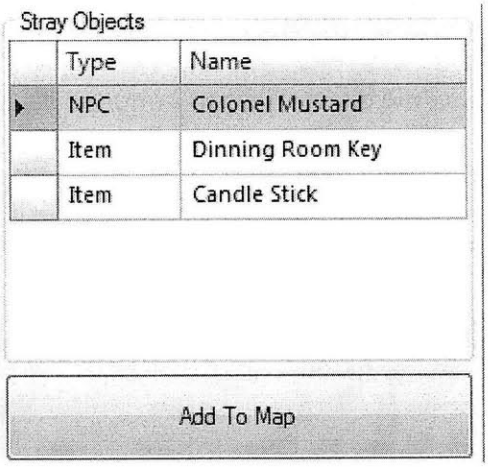


**Figure 5.11 Object Info View.** The Info View displaying the properties of all the game objects in "TimeLab 2100".

# 6. Conclusion

## 6.1 Future Work

Although the MITAR Full Editor is already a powerful and flexible tool for MITAR game development, this is not to say it could not benefit from further work in the future. For example, the Map View is currently the only place where a portal object can be added to a game. Time constraints

prevented the inclusion of portal objects within the Object Info View. However, game designers would be able to fully maximize their use of the Object Info View if portal objects, along with any other new objects added to the Map View in the future, are added to the Object Info View as well. The Flow View would also benefit greatly from the ability to change the node and arrow styles. Using different styles would make the Flow Diagram easier to scan for individual game objects and triggering rules. Finally, the undo / redo functionality needs to be expanded so that it affects all three of the Full Editor views. At the moment, undo / redo only functions within the Map View. It is a waste to provide undo / redo functionality for one part of an application and not another, especially since it is such a powerful feature for correcting errors.

Although heuristic analysis and informal tests all suggest that the Full Editor is indeed a much better MITAR development platform in terms of usability, the Full Editor would benefit from more formal usability tests. In one such experiment, the subjects would be asked to create a new game using either the Game Editor or the Full Editor. The subjects would be given a script detailing all of the game objects, content, and triggering rules to put into the game. The experimenter would record time it takes each subject to finish creating the game. In end, the game creation times for the Game Editor and Full Editor would be compared to each other, to

decide which editor was more efficient. In addition to recording the time, the experimenter note the number of lapses and slips that went unnoticed by the subjects in each of the editors. This would provide a good measure of how each editor behaves with respect to error facilitation. One final experiment would involve the experimenter setting up a MITAR game with a number of errors deliberately put into it. The experimenter would then ask the subjects to search for these errors, using either the Game Editor or the Full Editor. The experimenter would then record the time required to find all the errors, or the total number of errors found. These statistics could then be used to evaluate which of the two editors is a more efficient tool for error spotting and correction.

## 6.2 Final Thoughts

The ultimate goal this project was to provide MITAR game designers with a more usable development tool. The Full Editor that resulted from this project is the combination of the most usable elements of the other MITAR editors. The implementation of the Flow View improves the usability of the Full Editor even further by providing the game designer new insights into the game flow that were previously unavailable to them. In addition, the Full Editor maintains compatibility with the other applications that support MITAR game development, the Game Builder, the Remote Editor and the Game Emulator. And although the Full Editor is far from perfect, and will likely go through many changes and iterations as time goes on, heuristic analysis and informal testing suggest that the Full Editor has indeed fulfilled the requirements set out for this project. The Full Editor will serve MITAR game designers well as they set out to create the next generation of Augmented Reality games.

# 7. References


- Azuma, R. 1997. A Survey of Augmented Reality. *Presence: Teleoperators and Virtual Environments 6*, no. 4, pp. 355-385.

- Bonsor, K. 2010. How Augmented Reality Works. How Stuff Works, Inc. Available at <http://computer.howstuffworks.com/augmented-reality1.htm> (accessed January 24, 2010).

- Dumas, J. S., and J. C. Redish. (1999). *A Practical Guide to Usability Testing* (Revised Edition). Exeter, England: Intellect Ltd.

- Klopfer, E. 2008. *Augmented Learning: Research and Design of Mobile Education Games*. Cambridge, MA: The MIT Press.

- Krug, S. (2000). *Don't Make Me Think: A Common Sense Approach to Web Usability.* Indianapolis, IN: New Rider Publishing.

- Milgram, P., and F. Kishino. 1994. A taxonomy of mixed reality visual displays. *IEICE Transactions on Information and Systems,* E77-D, no. 12, pp. 1321-1329.

- Miller, R. (2009). Unpublished usability lecture notes. Available at <http://stellar.mit.edu/S/course/6/sp09/6.831/courseMaterial/topics/topic2/readings/text1/text> (accessed January 27, 2007).

- Nielsen, J. (2003). *Usability 101: Introduction to Usability.* useit.com. Available at < http://www.useit.com/alertbox/20030825.html> (accessed January 26, 2010).

- Norman, D. (1988). *Design of Everyday Things.* New York, NY: Basic Books.

- Schellar Teacher Education Program. 2003. TimeLab. *AR Games Projects*. Available at <http://education.mit.edu/drupal/ar/projects> (accessed January 25, 2010).

- Schellar Teacher Education Program. 2007. Getting Started Designing & Playing AR Games. *MIT Augmented Reality Documentation v 6.0*. Unpublished document. Cambridge, MA. Massachusetts Institute of Technology.

- Soler, L., Nicolau, S., Hostettler, A., Fasquel, J. B., Agnus, V., Charnoz, A., Moreau, J., Dallemagne, B., Mutter, D., and J. Marescaux. 2009. Virtual Reality and Augmented Reality Applied to Endoscopic and Notes Procedures. *World Congress on Medical Physics and Biomedical Engineering 12,* no. 6, pp. 362-365.

- Sony Entertainment America Inc. 2007. PLAYSTATION 3 Brings Collectable Trading

Card Games To Life In THE EYE OF JUDGMENT. Sony Press Release. Available at <http://www.us.playstation.com/News/PressReleases/430> (accessed January 25, 2010).

- Thomas, B., Close, B., Donoghue, J., Squires, J., Bondi, P. D., Morris, M., and W. Piekarski. 2000. ARQuake: An Outdoor / Indoor Augmented Reality First Person Application. *Proceedings of the Fourth International Symposium on Wearable Computers* (ISWC'00) pp. 139-146.

- Thompson, A. 2010. How James Cameron's Innovative New 3D Tech Created *Avatar.* PopularMechanics. Available at <http://www.popularmechanics.com/technology/industry/4339455.html?page=1> (accessed January 24, 2010).

- Wang, T. 2008. *Case For Usability: Designing Outdoor Augmented Reality Games.* Masters of Engineering Thesis. Cambridge, MA. Massachusetts Institute of Technology.

- Webster, A., Feiner, S., MacIntyre, B., Massie, W., and T. Krueger. Augmented Reality in Architectural Construction, Inspection, and Renovation. *Computing in Civil Engineering,* pp. 913-919.