6-2014

THE I: A CLIENT-BASED POINT-AND-CLICK PUZZLE GAME

Aldo Lewis
CSU, San Bernardino

Follow this and additional works at: https://scholarworks.lib.csusb.edu/etd

Part of the Computational Engineering Commons, Data Storage Systems Commons, and the Other Computer Engineering Commons

Recommended Citation
https://scholarworks.lib.csusb.edu/etd/74

This Project is brought to you for free and open access by the Office of Graduate Studies at CSUSB ScholarWorks. It has been accepted for inclusion in Electronic Theses, Projects, and Dissertations by an authorized administrator of CSUSB ScholarWorks. For more information, please contact scholarworks@csusb.edu.
THE I:
A CLIENT-BASED POINT-AND-CLICK PUZZLE GAME

A Project
Presented to the
Faculty of
California State University,
San Bernardino

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

by
Aldo Lewis
June 2014
THE I:
A CLIENT-BASED POINT-AND-CLICK PUZZLE GAME

A Project
Presented to the
Faculty of
California State University,
San Bernardino

by
Aldo Lewis
June 2014

Approved by:

David Turner, Committee Chair, Computer Science

Arturo Concepcion, Committee Member

Tong Yu, Committee Member
ABSTRACT

Given mobile devices’ weak computational power, game programmers must learn to create games with simple graphics that are engaging and easy to play. Though seldom created for phones and tablets, puzzle games are a perfect fit. In recent years, the genre has gained a following and even won some acclaim. Games like Myst, The Seventh Guest and Portal all engage gamers with challenging puzzles and then reward them with story components upon task fulfillment. Few such games have been created for mobile devices, in part due to the difficulty of developing for devices with different operating systems. Android, WebIOS and Windows Phone all have different software development kits that produce a final product incompatible with operating systems other than what it was developed for. One promising solution is to use browser technology to deliver games since all devices are geared to interact with the Web through browsers such as Internet Explorer, Mozilla Firefox, and Google Chrome. The aim of this project was to build a puzzle game that can be run on any digital device. The project can be accessed without any plug-ins and was created by using web technologies such as JQuery, Touch Punch, local storage, and WebGL. JQuery allows drag and drop functionality and Touch Punch allows the JQuery functionality intended for a mouse to work on a touch interface. Local storage provides storage on a user’s device, as opposed to a server, and WebGL enables graphics processing on a user’s tablet or phone through web commands.
TABLE OF CONTENTS

ABSTRACT .............................................................................................................................. iii
LIST OF FIGURES ................................................................................................................ ix
CHAPTER ONE: INTRODUCTION

- Background..................................................................................................................... 1
- Technology Overview .................................................................................................... 3
  - JavaScript .................................................................................................................. 3
  - JQuery .......................................................................................................................... 3
  - JQuery UI ................................................................................................................... 4
  - Touch Punch ............................................................................................................... 4
  - Local Storage ............................................................................................................ 5
  - GIMP .......................................................................................................................... 5
  - WebGL ....................................................................................................................... 5
- Project Purpose .............................................................................................................. 6
- Project Scope .................................................................................................................. 6
- Related Work ................................................................................................................ 6
- Definitions, Acronyms, and Abbreviations ................................................................... 7

CHAPTER TWO: GAME DESIGN

- Title ............................................................................................................................... 10
- Genre ............................................................................................................................. 10
- System Requirements .................................................................................................... 10
- Game Mechanics .......................................................................................................... 10
<table>
<thead>
<tr>
<th>Section</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>Level Object</td>
<td>38</td>
</tr>
<tr>
<td>Load Level</td>
<td>38</td>
</tr>
<tr>
<td>Start Game</td>
<td>39</td>
</tr>
<tr>
<td>Next Text</td>
<td>40</td>
</tr>
<tr>
<td>Local Storage</td>
<td>44</td>
</tr>
<tr>
<td>Entry</td>
<td>44</td>
</tr>
<tr>
<td>Index</td>
<td>44</td>
</tr>
<tr>
<td>Values</td>
<td>45</td>
</tr>
<tr>
<td>Storage</td>
<td>47</td>
</tr>
<tr>
<td>Stringify</td>
<td>47</td>
</tr>
<tr>
<td>Parse</td>
<td>48</td>
</tr>
<tr>
<td>Form</td>
<td>48</td>
</tr>
<tr>
<td>Table</td>
<td>51</td>
</tr>
<tr>
<td>Game Loop</td>
<td>58</td>
</tr>
<tr>
<td>Current Game</td>
<td>58</td>
</tr>
<tr>
<td>Level</td>
<td>59</td>
</tr>
<tr>
<td>Inventory</td>
<td>60</td>
</tr>
<tr>
<td>Game Progress</td>
<td>60</td>
</tr>
<tr>
<td>WebGL</td>
<td>61</td>
</tr>
<tr>
<td>Canvas</td>
<td>63</td>
</tr>
<tr>
<td>Shaders</td>
<td>64</td>
</tr>
<tr>
<td>Fragment</td>
<td>64</td>
</tr>
</tbody>
</table>
LIST OF FIGURES

Figure 1. Text Button Displaying Color Theme ............................................ 15
Figure 2. Atticus Drawing Illustrating Game Anime Style ............................ 16
Figure 3. Dark Background and Lighter Draggable ....................................... 17
Figure 4. Electric Blue Text Boxes and Buttons .......................................... 18
Figure 5. Deployment Workflow Diagram .................................................. 19
Figure 6. Dependency Diagram ................................................................... 21
Figure 7. Package Diagram .......................................................................... 25
Figure 8. Flowchart Diagram ...................................................................... 27
Figure 9. Photo Before Edge-Trace ............................................................. 82
Figure 10. Photo After Edge-Trace .............................................................. 82
Figure 11. Original Door .............................................................................. 84
Figure 12. Perspective-Shifted Door ............................................................. 84
Figure 13. Welcome Page ............................................................................. 85
Figure 14. Account Manager Page ............................................................... 86
Figure 15. First Story Screen ...................................................................... 87
Figure 16. First Story Screen After JQuery Transition ................................. 88
Figure 17. Hallway with Closed Door .......................................................... 89
Figure 18. Hallway with Open Door ............................................................ 89
Figure 19. Game Screen Upon Page Load ................................................... 90
Figure 20. Game Screen After Task Fulfillment .......................................... 91
Figure 21. Inventory Appears on Clicking Inventory Button ......................... 92
Figure 22. Inventory Item Loaded in New Page ........................................... 93
Figure 23. Initial Eye Position ................................................................. 94
Figure 24. Eye After Mouse Movement .................................................. 95
Figure 25. User Attempting to Skip to Higher Level ............................... 96
Figure 26. User Redirected to Right Level .............................................. 97
Figure 27. Incorrect Trinket Dropped on Droppable ............................... 98
Figure 28. Correct Trinket Drop Triggers Story Event ............................ 99
CHAPTER ONE
INTRODUCTION

Background

With a wide gamut of digital devices, a program’s effectiveness now relies less on harnessing the strengths of one dominant computing system, and more on ease of portability to all systems. Building something intended to run on an 8-core personal computer is drastically different than a program meant for a mobile device; therefore, the rise of mobile devices has paved the way for software to be run on these devices. Though these devices are complete computers unto themselves, they are relatively low performing when compared to traditional desktop and laptop computers. With the inherently weaker computational power of mobile devices, programmers must adjust to the limitations of these devices, and instead of intending their software to run on the latest and most powerful machines, programmers must learn to interact with mobile devices in a way that utilizes a device’s functionality and that provides mobile users with a unique experience.

One opportunity that the new mobile community provides programmers is the impetus to develop simple but addictive games. Whereas game consoles and PCs have high-end graphics games that require large teams to develop, cell phone users understand that their devices are not capable of such processor-intensive games, and thus they are content with products that are not as flashy.
What is most important then is ease of use, playability, and the ability to stop and continue game play at any point. The significance of this project is that it attempts to fulfill the need for fun and addictive games for mobile devices.

The average mobile game emphasizes simple graphics and game play, and typically some social components. Prime examples are games such as Farmville, Angry Birds, and Candy Crush Saga. However, there is space for other types of games to gain popularity. In particular, puzzle games have garnered a following, and even acclaim, in recent years. Games like Myst, The Seventh Guest and Portal all compel gamers with challenging puzzles and then reward them with story components upon fulfilling tasks. Those types of games, though, have been intended for PCs and gaming consoles, and there have been few games created for mobile devices. In particular, it is difficult to develop for devices with different operating systems, such as Android, WebOS and Windows Phone. One promising solution is to use browser technology to deliver games to various devices.

This need for simple games intended for the casual gamer has long been catered to by the flash player plug-in, but there is a recent push for browser universal capabilities, as shown by new html 5 standards and the recent transition to interactivity through JavaScript. The problem with plug-ins is that they take the user out of the browser experience, and such plug-in experiences can vary widely depending on users’ hardware, browser, or mobile phone type. In fact, iPhone does not even support the use of Flash, as the company claims flash
is an invasive development environment [1]. The aim of this project is to build a puzzle game that can be run on any device, but which avoids the traditional Flash solutions.

Technology Overview

JavaScript

Thanks to current development, users can now enjoy free dynamic interaction through JavaScript, which allows users to access new content with limited server interaction. These interactions can be triggered by various mouse events, such as mouseover,mousedown, and mouse drag [2]. The benefit of JavaScript is that it is an integrated part of DOM standards, and as such, is intended to run fairly uniformly on most browsers. There are still some differences between browsers —such as how to handle click events— but libraries built on top of JavaScript, such as JQuery, smooth those differences out.

JQuery

Though JavaScript is already standardized, JQuery is an open-source library that allows programmers to perform tasks more easily. Instead of worrying about scheduling various mouse events —worrying about what will fire first— JQuery allows us to hide most of this functionality and call simple functions such as drag and drop. JQuery also facilitates native JavaScript functionality by allowing both truncated and object-oriented support, which allows one to write cleaner code that runs uniformly on different systems. So instead of
document.getElementById("value"), the value is simply passed in during the declaration of the object: $('#value').

**JQuery UI**

An advantage of JQueryUI, which is also open-source, is the various offshoot libraries that aid programmers in the task of sorting, resizing, and even drag and drop functionality [3,4]. Of course, all this was available before in JavaScript, but it was much harder to use. JQueryUI even allows us to check whether the item dropped is the one that we want to accept by setting an attribute called “scope.” These many functions allow rich interaction, which has users receiving feedback not just when clicking, but also when moving items around.

**Touch Punch**

Drag and drop movement is already instinctive to smart phone and tablet users, who do not have a mouse to click around, but that functionality is part of operating systems and not part of web technology, so to build a web game to be used on phones, we must use hacks such as the free library Touch Punch. The library replaces mouse events such as mouse move with touch move, mouse over with touch start, etc. Using Touch Punch allows us to write games that will run on various systems, instead of having to use specific kits to write games for Android, iOS, or Windows phone.
Local Storage

Another free technology that will allow us to provide functionality uniformly is local storage. For years, sites used cookies to store user information and persist data across pages because websites are intrinsically stateless, but such cookies can only maintain a small amount of information. Local storage allows us to store more information than normal cookies, at an average of five hundred megabytes, instead of four kilobytes for cookies [5]. Local storage is an html 5 specification, and thereby intended to run on all browsers equally.

GIMP

GIMP is a stand-alone open source graphics-editing program, and though it is not intrinsically linked to web technology, it is helpful for editing the images required for interactive web pages. Though css does allow the creation of graphics elements such as text boxes, gradients, etc., it is intended to be a positioning system, not a full-scale picture editor. So anything more difficult, such as the alteration of pictures and the creation of transparent pictures and layered scenes, must be created in a graphics editor. GIMP is relatively easy to use and open source.

WebGL

Another open source program that helps the game developer is openGL, and an affiliated library useful for the web programmer is webGL. WebGL adheres to most of the rules established by openGL, and even runs graphics on the client’s GPU. However, it is built on top of the HTML 5 canvas element, and
canvas is not universally supported, so by extension, neither is webGL. WebGL does run on most browsers, but not all; it has barely gained support in the most recent version of Internet Explorer. Since it barely runs on more powerful personal computers, support is even sparser on weaker tablets and phones. Therefore, the game does not rely much on webGL. What is included in the game is a preview of what is possible when universal support finally arrives.

Project Purpose

The intent of this project is to build a point-and-click story-driven puzzle game using client side web technology. The purpose of using web standards is to create a game that is relatively light weight, so it does not require specialized hardware or programming platforms. Furthermore, running on the Web assures a wide distribution for myriad systems.

Project Scope

This project will not be using any server programming or writing any native Android, IOS, or Windows Phone apps.

Related Work

This project is a new stand-alone project, but it builds upon the computer science department’s history of game development. A prototype of the project was conceptualized to fulfill the department’s game design class, CSE 440, in
collaboration with my class partner Mark Chapman. Though the proof of concept was originally constructed for the XNA developer’s club, and initial game design documents specified a game system with interactive controls like the Wii system, the concept can easily be ported to a browser. The project prototype simplifies the gameplay from 3D to 2D.

Definitions, Acronyms, and Abbreviations

2D

2-dimensional coordinate graphics system upon which location is dictated by the x and y-axis

3D

An abbreviation for three-dimensional computer graphics, which are graphics that use a three-dimensional representation of geometric data through the x, y, and z-axis

ANDROID

An operating system for smart phones and tablets built by Google

BROWSER

A program that enables communication through the Internet, typically through an http connection between client and server

COOKIES

Metadata stored on the client that allows programmers to track user Internet traffic
CSS
Cascading Style Sheets- allows programmers to specify the location and style of graphical elements on a web page

DOM
The Document Object Model- A child-parent structure representation of a web page document

GIMP
GNU Image Manipulation program, which provides tools to modify existing images

GRAPHICS PIPELINE
Term describing the steps 3D graphics undergo to become 2D rasterized images

GUI
Graphical User Interface- allows the user to interact with a program through an intuitive visual system, as opposed to commands given through a prompt

HTML
Hypertext markup language- used to structure web pages

IOS
Operating system for smart phones and tablets built by Apple

JAVASCRIPT
A web language intended to dynamically alter static html elements

JQUERY
An object-oriented library built on top of JavaScript

JQUERYUI

Built on top of JQUERY. It is intended to allow move and drop functionality to elements

NPC

Non-Playable Character- typically, a character in a video game that provides players with game information or quests

OPENGL

An open source graphics library language that allows rendering of graphics on the graphics card

SHADER

A program that manipulates individual graphics elements such as a vertex or pixel

TOUCH PUNCH

A library that transforms mouse events into touch events compatible with portable devices

WebGL

A web standard implementation of the popular openGL library
CHAPTER TWO
GAME DESIGN

Title
The title of the game is The i.

Genre
The genres of the game are puzzle and adventure.

System Requirements
The game is intended for all devices. It is planned to run on all personal computer operating systems, such as Windows, Linux, Mac OS, etc. It is also compatible with mobile phone and tablet operating systems such as Android, WebOS and Windows Phone.

Game Mechanics
The primary game mode will focus on exploring and traversing a 2D world filled with ancient technology. Players will interact with all the elements in the world by using the drag-and-drop functionality. Players can select objects and manipulate them with a click, and then drag objects to different places as needed.
The current version of the game can be completed in between fifteen to thirty minutes, but the envisioned full version should take at least four or five hours.

Game Story

Background

In the future, mankind begins to see the Internet as an all-encompassing repository of knowledge. Every man, woman, and child has a chip implanted in his or her brain that allows the upload of every waking memory, thought, and experience. The hope of this project is to both democratize knowledge and to allow humans to be capable of creating genuinely new ideas.

Unfortunately, such omniscience and super-connectivity is too much for humans to handle. Private residences become technological opium dens as people become addicted to the lives and experiences of others. After all, why live one’s life, when one can live the life of a rockstar? Even if people manage to evade the pointlessness of addiction, their bodies still suffer from years of misuse—obesity and heart disease plague the human blobs of the future. Those who are able to contribute to society find themselves inundated with the increasing amount of knowledge, and many become insane. All the while every thought and experience is documented.
Eventually people begin to accept the consequence of linking to the network. The few survivors capable of disconnecting from the Web are incapable of prospering in a world without technology and are cast back into a stone age.

A thousand years later, we find that mankind has begun to harness old technologies, but they are still in a dark age. People know little of their ancestors’ reliance on computers, and the only memories that remain are in superstitious admonitions. Religious leaders preach that an evil spirit lives within the old technology, and that the technological empires of the past should be avoided.

As a child, the main character, Atticus, lives near the technological ruins of a prominent city. The ruins are labeled as a restricted zone, but over the years security becomes lax.

Through a dare, Atticus finds himself in the city, which is like a maze because it was catered for interconnectivity instead of human transversal. As he gropes through the dark and deserted alleys, he finds himself falling and breaking his leg. Sitting there, hopeless, and unable to find an exit, the old technologies reboot and show him a holographic memory. The memory details a nano-technological gun that heals his broken leg.

Traumatized, he suppresses the incident.

Years later, a plague sweeps his village. He watches helplessly as families are decimated, but it is not until the disease reaches his doorstep that his repressed memories resurface. When his wife and best friend become ill, all his fears and superstitions are swept away, and he considers a journey into the
forbidden ruins. Surely, some of the technology must be of use to him in fighting the plague.

Gameplay Storyline

Upon entering the maze, Atticus finds himself trapped. The entrance shuts behind him and only select doors open, one at a time, leading further into the dungeon through what seems like a predetermined path.

Atticus’ hope is ignited as he meets the i, and the intelligence promises to help in finding the cure. Soon, though, the protagonist discovers that the i is not as reliable as hoped since the consciousness begins to splinter into its individual consciousnesses. First, the i finds itself turning into Aurora, a young girl whose physical life has long ended but whose memories live, and then a moment later, the i morphs into Aurora’s boyfriend Riley. Atticus begins to understand the fickleness of the i but pushes on.

Game Controls and User Interface

Users interact with the game through two interfaces: the selection screen and the scene. The selection screen is used only to start a game or continue a new game. Users will only be required to type information on this screen, and they will click on buttons to move to another page or submit info. For the most part, the user will interact with the scene, which consists of a textured background for each room and clickable objects in the forefront.
The way a user interacts with the two interfaces depends on what type of device they are using. On devices with a mouse, such as personal computers, the user will select items on the screen by left clicking. Movement around the game space and dragging are accomplished by moving the mouse. Entering text is completed by typing on a keyboard. On touch-based interfaces such as mobile phones and tablets, it is different. Touching the screen will allow users to select items, and moving one’s finger across the screen will allow dragging items. If users have a keyboard plugged into the mobile device, they may type with it. Otherwise, when the user touches a text input field, the phone or tablet’s native touch keyboard will allow the user to type.

Replayability

The driving factor in the game’s replayability is the story. The current version has no branching options, but the envisioned final will have the player choosing different paths, allegiances, and endings.

Sequel

There is no sequel planned for the game. Because the game so closely follows the unfurling of the i, there will be little left to explore in the future. However, if the gameplay proves addictive enough, it could be used for a game that takes place in another world.
Asset List

The game requires 2D assets to be used as backgrounds, buttons, text boxes, characters, and draggable items. The backgrounds are of fixed size, typically eight hundred by four hundred pixels, in order to fill the average user’s browser window. As shown in figure 1, buttons and text boxes are rectangular, with a black and electric blue gradient. The colors were chosen to reflect the game’s artistic theme: black for the dark maze and blue for the electrical life that inhabits the game world:

Figure 1: Text Button Displaying Color Theme

Characters are hand drawn anime style, which is a staple of story driven games, such as role-playing games. Unlike draggable items, which are created by altering images through the use of graphics editing software, characters should seem lively and vibrant. An example is that of Atticus, which may be seen in figure 2:
Figure 2: Atticus Drawing Illustrating Game Anime Style. Hand-drawing by Kimberly Cooprider

A full display of all the game elements at play may be seen in chapter six. For example, figure 13 displays a game level with a draggable metal spider. As may be noted, most of the colors are dark, with a dark draggable spider and dark background.
Figure 14 portrays story delivery after completing a challenge. Here, one may see the anime style characters, the black and electric blue gradient text backgrounds and buttons, and the user interface. In order to move to the next bit of text, the user must press the buttons to navigate forward or backward. Figures 13 and 14 together illustrate another user interface feature, the ability to drag and drop items. The first figure shows the screen before the draggable has been dragged to its destination, and the second figure shows the effect after dragging. The user interface is designed so that players have a simple drag and drop interface.
Figure 4: Electric Blue Text Boxes and Buttons. Hand Drawings by Kimberly Cooprider
CHAPTER THREE
SYSTEM ARCHITECTURE

Deployment Workflow

The client only interacts with one server, and that is for the singular purpose of retrieving new web pages. Any other interaction occurs on the client, be it on a pc or laptop, or on a smart phone or tablet. Figure 5 illustrates this.

Figure 5: Deployment Workflow Diagram
When a client wants to play the game, they send an http request to the server for the content. Using the http connection, the server then sends the content for the page back to the client, with all the necessary libraries to run the game on the client's browser. All other interaction for a given page occurs on the client, and that is the reason there are no other http requests flowing from the client to the browser. In a typical application, we might see an http request for each interaction: when a user clicks on a button to get more content, when a user submits account information, etc. However, in this project, there is only one interaction with the server per page, be it either for a mobile device or traditional computer.

This server independence is mainly accomplished by direct script links in the header section of html pages. These links reference external libraries stored in a separate folder, some of which work by themselves, and some of which reference other libraries in that same folder. Figure 6 uses the file movingEye.html to illustrate the dependencies since it is one of the files that uses all of the external libraries. The solid lines indicate that the html file relies on a library –in this case all five libraries, as all those libraries are included in the header of the file. The order that the javascript libraries are called in the header depends on their own dependencies on one another, and these interdependencies are denoted by dashed lines. For instance, touch punch depends on jquery-ui-1.8.23.custom.min.js, which in turn depends on jquery-1.7.2.min.js. Another interdependence exists between the WebGL libraries and
jquery. The file `jquery-1.7.2.min.js` must be called first, then `glMatrix-0.9.5`, then `webgl-util.js`; hence, the dependence arrows flow from `webgl-util.js` to `glMatrix-0.9.5` to `jquery-1.7.2.min.js`.

![Dependency Diagram](image)

Figure 6: Dependency Diagram

All of these external libraries are intended to run dynamically on the browser without the server, so they are all written in JavaScript.
JQuery Libraries

From the first arrival at the game’s first web page, welcome.html, the native functionality of the browser is supplemented by external libraries – the first of which is the JQuery library. JQuery is intended to facilitate functions already present in JavaScript, and the first library, jquery-1.7.2.min.js, has the broadest applications since it deals with creating JQuery objects, so it is the first to be called:

```
<head>
  <script src="script/external_libraries/jquery-1.7.2.min.js"></script>
</head>
```

The second library, jquery-ui-1.8.23.custom.min.js, is not needed until the beginning of the game, when the user will need to be able to drag and drop items.

```
<script type="text/javascript" src="script/external_libraries/jquery-1.7.2.min.js"></script>
<script type="text/javascript" src="script/external_libraries/jquery-ui-1.8.23.custom.min.js"></script>
```

Touch Punch

After the JQuery functionality has been established, we may call the Touch Punch library:

```
<script type="text/javascript" src="script/external_libraries/tpunch.js"></script>
```
In the same header where the root JQuery file and the JQuery UI file were called, the next line calls the Touch Punch library. This library relies on the widget function of the JQuery library, so it is called after the first two libraries have already been called.

**WebGL Libraries**

Another library required for the project is WebGL, which allows the creation of 3D graphics. Unfortunately, calling the webgl-utils.js is not enough. Traditional OpenGL has built-in matrix manipulation libraries, but WebGL does not. Because of the heavy need for matrices in 3D programming, we need to call the external file glMatrix-0.9.5.min.js to deal with the matrices. The matrix functions need to exist first, so we call the matrix library first, and then WebGL:

```html
<script src="script/external_libraries/jquery-1.7.2.min.js"></script>
<script type="text/javascript" src="script/external_libraries/glMatrix-0.9.5.min.js"></script>
<script type="text/javascript" src="script/external_libraries/webgl-utils.js"></script>
<script type="text/javascript" src="script/external_libraries/jquery-ui-1.8.23.custom.min.js"></script>
<script type="text/javascript" src="script/external_libraries/tpunch.js"></script>
```
Local Storage Access

Local storage does not require a call to an external library, as it is a standard feature of html 5 enabled browsers. That is why there are no header declarations that refer to this functionality. Just as one is able to declare an html tag or a javascript variable natively for browser programming, one is able to use local storage from the beginning.

Package Diagram

The interrelationship between the code and assets is illustrated in figure 7. The entrance into the program is through the html files for each level. The html depends on the css to structure items, the images file for assets, and the javascript files for any functionality. As noted in the diagram the image folder is further subdivided into separate sections for the different types of images loaded, and the script package includes the javascript files needed for game execution, including the game levels and the functionality that makes them possible, which will be discussed in the next chapter’s Javascript, JQuery UI and Touch Punch section. The javascript package also includes the external libraries needed for running the WebGL sample, and they will be discussed in the WebGL section of chapter four. Furthermore, even though it is not represented in the figure, because it does not require calling external libraries or resources, local storage is a vital component in allowing the game to run, including the gameloop file stored
in the game controller package, and that will be discussed in the Local Storage section of the next chapter.

Figure 7: Package Diagram
CHAPTER FOUR
SOFTWARE COMPONENTS

Software Component Flowchart

The following chapter provides an outline for the different software components required to run the game. It is organized to provide an understanding of how each component functions separately since detailing the back and forth transactions between parts might prove confusing. For example, the driving game loop object is created equally using JQuery and Local Storage, and it is constantly called throughout game play, so addressing its different uses at different times would have the reader jumping back and forth between concepts. Nevertheless, in the next page is included a workflow diagram that shows the general execution of the game.
As figure 8 shows, after the game is started, the next step is game account creation, which involves initializing the table that shows the different game accounts and the form that allows users to create one account. After a created account is select for use, the user begins the game proper, and this is where the game loop comes into play. As long as the user is playing, each page
will run through a loop in which the system checks users’ progress, and the current level is saved to their account, along with whatever inventory might be stored. Once user account concerns are completed, the functionality of individual levels is initialized. First, the elements needed are created using load level, and the interaction for those levels is defined in start game. What to do once the specific game level task is fulfilled is specified in the next text function, which then forwards the user to a new level, and this loop proceeds as users run through the different game levels. The loop ends when the user has finished all the levels; then, they may proceed to the WebGL sample. This final level requires a number of steps, starting from initializing the shaders and buffers, handling mouse events, loading textures, and drawing the scene.

JavaScript, JQuery UI and Touch Punch

JavaScript Object

JavaScript is very generous in that a programmer does not have to set the type of item to be stored. An integer, character, string, array, and object will all be set with the var keyword [6]. This is especially helpful with our gameLoop object.

Gameloop Object

Because JavaScript is not intrinsically object oriented, we can mimic the nature of object-oriented languages by making member variables and methods fields of the gameLoop object. The point of the gameLoop object is to keep track
of user progress through the game, so we call the method checkGameProgress(), which plays traffic cop by sending users to the right level.

Check Game Progress. Of course, in order to do this, we need to determine who is playing, which is performed by calling the function setCurrentPlayer. This function retrieves the current player from local storage and saves it to the field currentPlayer. The specifics are detailed in the chapter on local storage, but what can be clarified here is that we want to keep track of the data related to the current player, so we add a field named currentPlayer to the gameLoop object. It is initially set to the empty string:

currentPlayer: "",

Once we find the current Player, we set it. Remember that this current player field is an object unto itself. It contains identifying information such as the player’s name, game progress information such as what level the player is on and what story markers have been fulfilled, and any inventory the user may be carrying. Once the player object is in storage, most of the work will consist of reading information from the object, as writes only happen when something significant occurs.

Set Current Level. The first significant task performed is retrieving the player’s level:

var currentLevel = gameLoop.currentPlayer.level;

This numeric value will be used to iterate through the map containing the relative paths to different levels. The map is stored as a field of the game loop object:
gameProgressionMap:


In order to find the absolute path, we retrieve the value of the host’s url. In this case, the server is at http://www.aldolewis.com:

base: "http://www.aldolewis.com/TheiProject/",

We append the server string to the relative path string and set them as the current game level url:

var currentLevelURL =

gameLoop.base+gameLoop.gameProgressionMap[currentLevel];

Put together, we know where the player should be. For example, if the player has just begun the game after creating their account, they will be in the first level and will be directed to the beginning story screen:


If, for whatever reason, the user is not at the right URL, we iterate through the game progression map, checking the current URL against the URL of a given level. If a user is at an earlier level, we allow them to remain, as a user may return to earlier levels. However, it is a problem if they try to travel to a higher level. Therefore, when we are looping through the progression map, if the current URL matches a level, we see if the id of that URL is greater than the current player’s level. If that is the case, the player is trying to skip, so we redirect the player to the level they should be in:
if (document.URL != currentLevelURL)
{
    for (var i=0; i<gameLoop.gameProgressionMap.length; i++)
    {
        currentLevelURL = gameLoop.base + gameLoop.gameProgressionMap[i];
        if (document.URL == currentLevelURL)
        {
            if (i > currentLevel)
            {
                window.location.replace(gameLoop.gameProgressionMap[currentLevel]);
            }
        }
    }
    break;
}

Set Up Inventory. After the user is in the right level, we can begin setting up the inventory:

gameLoop.setUpInventory();

We do not want the inventory to load the moment the user enters the level, so we bind a click handler to the inventory button, allowing the user to bring up the inventory at will:

$(".inventoryButton").click(function()

We only want one inventory up at a time once the event handler fires, so we check that one has not yet been created:

if (gameLoop.inventoryNotYetCreated)
If it has not, we may proceed and create it. The first step is to set our
inventory creation flag to false:

```javascript
gameLoop.inventoryNotYetCreated = false;
```

The next step is to create the actual inventory. Because we are appending to the
tree-structure DOM, we have to find a parent to attach our child node to. In this
case, we will pick the inventory button to attach to. The new child will have the id
“inventory” and will include the path to the image intended to represent the
inventory. For artistic purposes, it is preferable to represent inventory as a picture
rather than a css-created box:

```javascript
$('.inventoryButton').append('<p id="inventory"><img
src="images/draggable/InventoryFinalGrid.jpg" style="border-
style:none;"></p>');</
```

Since this is going to be an inventory, we want users to be able to drag it around
the screen so they can situate it where it won’t interfere with selection of other
items on the screen, and we also want it droppable since items will be placed in
it. Thankfully, Jquery UI handles this functionality:

```javascript
$('#inventory').draggable();
$('#inventory').droppable(

In terms of usage, dragging the inventory is simple: the user selects the item and
places it where they want it to go. Unfortunately, handling dropping on the
inventory is more complex. For one, we want to know what is being dropped on
the inventory and whether to accept it, and the scope attribute allows us to
accomplish the task. As an example, we can give a draggable item a specific
name, e.g., “important,” to signify it can be dropped, and we then assign an accompanying identifier to the droppable to denote what type of item to accept. This ensures that we accept only items that can be stored and not any given draggable item:

```javascript
scope: "sortable",
```

Once the inventory has been given a scope, we may dictate what to do upon a legitimate drop, which is accomplished through the callback drop function:

```javascript
drop: function (event, ui)
```

In this function, we first find the id of the draggable item that has been dropped:

```javascript
var droppedItemId = ui.draggable.attr("id");
```

Now that we know what item has been dropped, we will have to decide whether this item will be the first item to populate the inventory, or whether it will be replacing an already stored item. For the former case, we check that there is not yet an item in the inventory, and if that condition is true, we store the item dropped into the inventory.

```javascript
if(!gameLoop.currentPlayer.inventory)
{
   gameLoop.storeInventory(droppedItemId);
}
```

Storing the inventory requires two writes, one to the object in memory and one to storage. The one to storage is explained in the local storage chapter, but the one in memory only requires storing the id we passed in:

```javascript
gameLoop.currentPlayer.inventory = itemID;
```
In order to avoid having duplicate copies of an item—one dropped onto the inventory from the page and one loaded from the inventory—we only allow the one loaded from inventory to exist. Thus, we erase the dropped item:

```javascript
$(ui.draggable).remove();
```

We then load the one from inventory:

```javascript
gameLoop.loadInventory();
```

Loading from inventory technically requires creating the item anew. When we store an item, all that is stored is the id of the item, so we use that seed to create the final product.

After initializing the drop function, we want to load the inventory. To do so, we first check whether the item to be loaded already exists on the page:

```javascript
if (gameLoop.checkIfItemExists())
```

The list of storable items is small, so we define an array with the possibilities:

```javascript
var possibleDraggableItems =
  ["chair","mechSpider","wrench","trinket","falseTrinket1","falseTrinket2"];
```

We iterate through that list and check if any of the items already exist on the page by calling the `getById` JavaScript function. If the item exists, the function will return true, and we will pass on the message in our own `checkIfItemExists` function:

```javascript
for (var i=0;i<possibleDraggableItems.length;i++)
{
    if (document.getElementById(possibleDraggableItems[i]))
        return true;
```
Once that condition is true, we store that result into the gameloop object for easy reference:

```javascript
gameLoop.duplicateItemOnPage = true;
```

Determining if there is a duplicate on the page is important because JQuery functionality will only work on one item with a given id; a duplicate item would sit on the page and do nothing. In order to distinguish duplicates, one will be given the default name (e.g. “chair”), and the copy will have an extension (e.g. “chaircopy”). After this, we can proceed to load the inventory.

In the last function, when we checked for a duplicate, we set the field `duplicateItemOnPage` to true if the duplicate was already a storable item on the page. This helps if we find that the item to be loaded happens to already exist. If it does not, we would append the empty string to the item, leaving the name untouched:

```javascript
var draggableNameAddition = "";
```

However, if there were a copy, we would add the string “copy” to it.

```javascript
if (gameLoop.duplicateItemOnPage)
{
    draggableNameAddition = "copy";
}
```

Because we append the copy addition to the item name, it is possible that the item we are loading was already a copy, either because a copy was stored earlier when the user performed actions on another page, or through a
progression of stores on the current page. Therefore, when we fetch the name of
the item stored in inventory, we check whether it has the name copy by
searching for the regular expression “copy.” If it is a copy, we want to separate
the string “copy” for now and add it later, after the exact creation of the item. The
reason for this is that the cases we run through will only check for the root name
of an item and create that item. That is, “chair” and “chaircopy” will both generate
the same item; they will just have different names. For example, to load a chair:

```javascript
    case "chair":
        $(".inventoryButton").append(''<p id="chair + draggableNameAddition +"
"<img src="images/draggable/standalonechair.png" style="border-
style:none;"></p>'

$("#chair"+draggableNameAddition).draggable(
{
    scope: "important"
});

break;
```

Notice that when we added the draggable name addition to the chair, if the chair
was the original, it had the empty string added, to equal “chair,” and if it was a
copy, it had “copy” added for a resulting “chaircopy.” All other items are created
in the same manner, and each is given an appropriate scope. Chair is given the
scope “important” because it will trigger a story event. However, items that do not
trigger story events, such as “falsetrinket,” will not receive a scope:

```javascript
    case "falseTrinket1":
        $(".inventoryButton").append(''<p id="falseTrinket1 +
```
A final use of JQuery in our game loop object occurs when we set the player's level after they pass through a doorway that has been newly opened following the completion of a story event. Take the example of the second hallway, where players will only be allowed to progress through the door once they have met “i” for the first time. We check if they have, and if that condition is true, we create the image of an open door that acts as a hyperlink. When the user clicks on the link, their level is set to the suggested level by calling the function setLevel:

```javascript
checkFirstEncounter: function()
{

    if(gameLoop.currentPlayer.storyProgressMarker == "iMet")
    {

        $('.frontDoor').html('<a href="javascript:gameLoop.setLevel(5)"
        style="text-decoration:none"><img src="images/background/frontOpenDoorFinal.jpg" border="0"></a>');
    }
}
```

The way the setLevel function works is by checking the player's level, as found in the game loop object in memory. If the stored level is lower, the player has probably not progressed further, so the level is set to the higher one. In this code
sample, case five. If the player’s level is higher, or equal, then the player has already progressed further in the game, so we leave that recorded higher level.

We do not want to overwrite with the lower level:

```javascript
if(levelNumber>gameLoop.currentPlayer.level)
{
    gameLoop.currentPlayer.level = levelNumber;
}
window.location.replace(gameLoop.gameProgressionMap[levelNumber]);
```

**Level Object**

Besides being useful in creating the gameloop object that keeps track of user progress, JQuery can also be used for the actual game levels, where we can update the page according to user interaction. The first step is to simply load the level.

**Load Level.** In the html, we provide very little data about the item on the page since that data is provided through a JavaScript variable. In the load level function, we load the items onto the web page as follows;

loadLevel: `function()
{
    var level = '<div id="chair" class="ui-widget-content"><img src="images/draggable/standalonechair.png" /></div><div id="droppable" class="ui-widget-content"></div>';
    $('".thisRoom") .html(level);
}`
Start Game. At this point in the code, the items are only static images; to make them interactive, we call the next function, start game:

```javascript
levelOne.startGame();
```

In the function, we make the chair draggable so the user can move the chair to a given spot on the page, and we do so by declaring a droppable section. This section has the same scope, “important,” and has a drop function that activates when the user drops the right object. In the case of the first level, the user is told to move a chair to a certain location so that they can access a part of the level. When the chair is dropped, it will trigger a traditional 2D dialogue scene that consists of pictures of the two characters talking and a text box:

```javascript
$("#droppable").droppable({
  scope: "important",
  drop: function (event, ui) {
    $('.thei').html('<img src="images/artwork/theiTotemFinal.png" style="border-style:none;">');
    $('.atticus').html('<img src="images/artwork/atticusFinal.png" style="border-style:none;">');
    $('.textbox').html('<img src="images/background/textBackground.jpg" style="border-style:none;">');
    $('#backButton').html('<img src="images/buttons/FinalBackButton.jpg" style="border-style:none;">');
    $('#nextButton').html('<img src="images/buttons/FinalNextButton.jpg" style="border-style:none;">');
  }
});
```
The text is loaded from a field in the level object:

```javascript
$(".textboxText").html(levelOne.firstText);
```

In this case, it calls the text:

```javascript
firstText: "<h3>i</h3><p>Are you... organic?</p>",
```

**Next Text.** After the initial text, the function nextText is called, which binds a click function to the back and next buttons, allowing the user to scroll through the next and last text:

```javascript
levelOne.nextText();
```

The way the nextText function works is that it increments the index field—which is a field of the level one object—every time the user clicks the next button. This rewrites the html of the textboxText JQuery object. When the player has scrolled through all the text, they have finished the first encounter with the i and can proceed to the next task. Therefore, the story market is updated and the user is redirected to the next page, which in this case happens to be the hallway that led to the first game level. The hallway previously had a closed door, but now that the user has met the i, the door will be open:

```javascript
$("#nextButton").click(function()
{
    levelOne.index++;
    switch (levelOne.index)
    {
        case 2:
```
The index is set to one originally and increments up to five in this game level, with each numbered case representing dialogue text and case four being the last text case; case five redirects the user. That is, case one gives the first text, case two gives the second text, etc. To make sure that the functionality works equally in the back button function as it did in the next button function, we assign cases one through four to the corresponding first through fourth text, but, because we are decrementing the index, instead of incrementing, it would be possible to reach case zero when the user is in case one and hits the back button. Therefore, in the back button function, we plan for the user to reach case zero, and when that happens, the index is automatically set back to one, leaving the user with the case one text – the first text:

```javascript
$("#backButton").click(function()
{
    levelOne.index--; 
    switch (levelOne.index)
    {
```
case 0:
    levelOne.index=1;
    break;
case 1:
    $('.textboxText').html(levelOne.firstText);
    break;
}

Most of the game levels have similar structure, with some minor differences needed for specific functionality. For example, in game.js, the user drops the draggable item in the right position and receives static text, but in game2.js, a bit of animation is required. The user is required to move a spider-like contraption, and when the item is dropped in the right place, it is supposed to come to life and jump out at the user. To accomplish the task, we can utilize the animate, fadeout, and hide JQuery functions in tandem. The mech spider is intended to be dragged, so to have it steady and move it deterministically, we remove that original spider and replace it with a new one, changing the name from mech spider to critter, just to indicate that two are different. We then animate this new spider by stating that it is to move two hundred and fifty pixels to the left:

drop: function (event, ui)
{
    $('#mechSpider').hide();
    $('.critter').html('<img src="images/artwork/fivelegged.png" style="position:absolute;left:0px;top:0px">');
    $('.critter').animate({left:'250px'}, function()
Upon finishing the movement, the animate function then calls another function.

So far, the user has seen the spider run around the screen. To get it to jump out at the user, we remove the moving spider and replace it with a bigger one, to indicate to the user that the spider is now closer. The spider is supposed to attach itself to the character’s head and allow the player to communicate with the i through a screen on the mechanical spider’s underbelly, so we want the large spider image to fade away slowly into what will ultimately be a textual dialogue exchange like that of game.js:

```javascript
$(".critter").hide();
$(".biggerCritter").html('<img src="images/artwork/televisionWithLegs.png" style="position:absolute;left:0px;top:0px">');
$(".biggerCritter").fadeOut("slow", function()
```

The next game level, game3.js, uses the same tricks as the first two game levels, but the fourth game level introduces a new game element. Up to now, declaring the scope of a droppable has been merely a formality since every game level has only had one draggable and one droppable. However, game4.js has the user deciding between three draggables, of which only one will be accepted by the droppable. The two false droppables do not receive the scope “important,” so they will not be accepted by the droppable:

```javascript
$("#trinket").draggable({
  scope: "important"
})
```
The game task in the final game level is different from that of the first four levels. There is no dragging or dropping, just entering a code, and JQuery makes the task easier to code. In game level four, the player learns about a commemorative date for one of the NPCs, and that date can be used as the password to be entered on a number pad on this level. Each number is a JQuery object assigned a click event. When a user clicks on a number, that number is appended to a password variable. Upon typing in the right code, the user will see the next dialogue scene. An example of a number:

```javascript
$.one().click(function()
{
    levelFive.password += '1';
    if(levelFive.password=='0214') levelFive.loadText();
});
```

Local Storage

Entry

Index. localStorage uses name-value pairs, as opposed to the data objects that are used in database languages such as mySQL and SQL [7];
objects in those languages are stored in tables and retrieved through sophisticated queries. In order to mimic the concept of the unique keys used in database languages, we create an index [8]. This index starts at one and is automatically incremented after the creation of each entry. Therefore, each entry receives a unique identifier, so no two are ever alike. When an entry is deleted, that identifier disappears forever so that no new entry will ever receive a previously created id. On creating a new entry, the user has the capacity to input the name of their account and every other field, but they cannot access the index, so they will never duplicate it.

Values. The very purpose of using local storage is to allow the user to save their game and then restore their save for play later, so it is essential that users are able to assign their game a name, and to be able to distinguish which game belongs to what users on the same computer, so we allow the user to set the name “name.” As each user may have more than one game in play at the same time, we also allow the user to set the “gameName” field. Thus, I may have “name” Aldo and account 1 and 2.

The rest of the fields are all set by the system as the user progresses through the game. We do not allow the user access to the other fields so they cannot cheat. The field level shows how far the user has progressed in the game; we start at level one and go to level eighteen. Each level number corresponds to a different web page, so it allows us to know what html page the user will be directed to. Most of these web pages have preset branches to other pages, but
there are a few branches that will only open if the user has completed certain tasks. Each level encompasses either a transition hallway or a game level where the player has to solve a puzzle.

Certain doors in the hallways will only open once a game goal has been fulfilled. This logic is encompassed in the value `storyProgressMarker`. In this field, we store the most current goal the user has fulfilled, and certain hallways will check the value to see whether the user is cleared to progress further. Earlier hallways check for both their own game markers and for later game markers in case the user has returned to earlier web pages. With the combined `storyProgressMarker` and `level` fields, we assure that users are not able to simply type in the url of a html page further down the story line.

Another important member of the entity object is the `currentGame` field. This member contains a boolean value and tells us whether the current entry is the one the player is using. Only one entry can be used at a time, so only one of the entries created will hold the value “true,” and all other entries will hold the value “false.” The entry that has the value set to true would be the one loaded if the user were to bypass the accountManager page and try to skip to a game level.

The final member of the entry object is inventory. In this location, we store the name of any item stored in the inventory. Only one item can be stored in the inventory at a time, and we will load the item into the inventory whenever the user opens the inventory during play. If the user drops a new item onto the
inventory, the user will be prompted about whether they wish to replace the old
item with the new, and if the user replies “yes,” the name of the new item will be
written in the inventory field. In case the name of the new item is the same as the
name of the old item, the new item will be given an extended name when loaded
onto the game page in order to assure that only one item of the a given element
id will be loaded onto the page. Otherwise, all css and JavaScript definitions
would only apply to one of the items. For example, if we have an item named
“chair” in the inventory, and the user drops another item also named “chair,” one
of those items will be named “chaircopy.” However, when the name is stored in
the inventory, the “copy” is removed. So, when “chaircopy” is actually stored in
the inventory member, it will only be named “chair.”

Storage

Stringify. The name value pairs in local storage are stored as strings.
Therefore, if our key was “name” and it was assigned the value “Joe,” the pairing
would be stored in storage as “name=Joe.” Because of this system, our entry
object will not actually be stored as an object, but instead, will be stored as a long
string encompassing the various member values and their keys.

Fortunately, we do not have to worry about converting the object into a
string, as a JavaScript library called JSON contains a command called stringify,
which will perform the task for us automatically [9]. Thus, the following sample
object can be transformed:

id=1
name=john
gameName=game1
level=2
currentGame=true
inventory=chair
storyProgressMarker=iMet

It will become one big string,
"id=1;name=john;gameName=game1;level=2;currentGame=true;inventory=chair;
storyProgressMarker=iMet"

Parse. Our entry objects are stored as strings, so in order to use individual components again, they must be parsed. The JSON library aids with the command “parse.”

Form

The accountManager page is divided into two components: the form and the table. The main task of the form is to handle any submitted data, which will either be in the form of a new entry added to local storage or an edit to existing stored data. It is easiest to handle the submitted data using JQuery, so we turn the form into a JQuery object:

$form: document.getElementById("Player-form"),

The action allows us to add any JQUERY functionality to the form. Specifically, we reset the form to clear it:

Player.$form.reset();
Then, because the form is particularly concerned with submittals, we add an event listener that will run when data is submitted. What we expect to be submitted through the form will be an entry. Here we officially declare all the member variables that will be needed and fetch them from their corresponding html elements:

```javascript
Player.$form.addEventListener("submit", function(event)
{
    var entry = {
        id: parseInt(this.id_entry.value),
        name: this.name.value,
        gameName: this.gameName.value,
        level: this.level.value,
        currentGame: this.currentGame.value,
        inventory: this.inventory.value,
        storyProgressMarker: this.storyProgressMarker.value
    };
}
```

The user is allowed to set name and gameName, so their values will be assigned to the new entry. If it is a new entry though, the user has never played with that account, so all other values are set to zero, the empty string, or false. The level is set to zero since we have not started a game, so we are not even at level one. CurrentGame is false because the entry account has not yet been selected for playing. Inventory and progressMarker are set to the empty string since nothing has been stored and no story events have transpired:
<input type="hidden" name="id_entry" value="0" />
<input type="hidden" name="level" value="0" />
<input type="hidden" name="currentGame" value="false" />
<input type="hidden" name="inventory" value="" />
<input type="hidden" name="storyProgressMarker" value="" />

After the entry is created, we want to know whether the entry is new, and therefore to be added to the table and storage:

```javascript
if (entry.id == 0) {
    Player.storeAdd(entry);
    Player.tableAdd(entry);
}
```

Adding an entry to the table will be explained in the table section, but adding to localStorage is fairly simple. We retrieve the current index from the player item, increment the index so it is ready for the next item, and then add our entry using the `stringify` function:

```javascript
storeAdd: function(entry) {
    entry.id = Player.index;
    window.localStorage.setItem("Player:index", ++Player.index);
    window.localStorage.setItem("Player:"+ entry.id, JSON.stringify(entry));
},
```

Alternatively, if an item is not new, we want to know if it is old and to be modified:

```javascript
else
```
Again, the table function will be explained later. For local storage, we only need to write over the old entry in storage with the new data:

```javascript
storeEdit: function (entry) {
window.localStorage.setItem("Player:"+ entry.id, JSON.stringify(entry));
},
```

Once we are done performing any additions or edits, entry.id is set to zero, so it is ready for a new action:

```javascript
this.reset();
this.id_entry.value = 0;
```

Finally, we prevent the submittal default action, which is typically a page redirect to another page. We do not yet want to move to another page when adding or editing. That will be done later when the user selects an account to play with:

```javascript
event.preventDefault();
```

**Table**

The table is what displays any current account in existence. To do this, we check whether there is anything in local storage. We iterate through each item in local storage, and if we find that it is a Player item, we place it in the Player_list array. Because each item is a string, we use the regular expression "/Player:" to
find if that expression occurs within the string, and since it is a string, we convert it into a JavaScript object before placing it in the array:

```javascript
if (window.localStorage.length - 1)
{
    var Player_list = [], i, key;
    for (i = 0; i < window.localStorage.length; i++)
    {
        key = window.localStorage.key(i);
        if (/Player:\d+/.test(key))
        {
            Player_list.push(JSON.parse(window.localStorage.getItem(key)));
        }
    }
}
```

After every player object has been placed in the list, we organize the list and add it to the table:

```javascript
if (Player_list.length)
{
    Player_list.sort(function(a, b)
    {
        return a.id < b.id ? -1 : (a.id > b.id ? 1 : 0);
    })
    .forEach(Player.tableAdd)
}
```

The way it is added to the DOM is by using JQuery. We create a parent tr for the table, and the children are populated through iterating through the keys:
```javascript
function(tableAdd) {
    var $tr = document.createElement("tr"), $td, key;
    for (key in entry) {
        if (entry.hasOwnProperty(key)) {
            $td = document.createElement("td");
            $td.appendChild(document.createTextNode(entry[key]));
            $tr.appendChild($td);
        }
    }
}
```

However, we are not just informing the user about the entries in storage, but we are allowing them to interact with each entry, and either edit it, remove it, or play it:

```html
$td.innerHTML = '<a data-op="edit" data-id="' + entry.id + '">Edit</a> | ' + entry.id + '">Remove</a> | <a data-
```

Right now, we simply provide the html text to be displayed, along with its ids, though we will later make it interactive when we add event handlers.

When we return from the function call to tableAdd, we are ready to add the event handlers. There are three possible actions a user can perform on table entry: edit an entry, remove an entry, and play with the entry. The first option,
editing, is essentially like adding an entry. The only difference is that before we
add the entry, we populate the table with the already existing entry, passing
values from the entry object onto the JQuery form:

```javascript
if (op == "edit")
{
    Player.$form.name.value = entry.name;
    Player.$form.gameName.value = entry.gameName;
    Player.$form.level.value = entry.level;
    Player.$form.currentGame.value = entry.currentGame;
    Player.$form.id_entry.value = entry.id;
}
```

The values passed in will populate the form, and the user will be able to maintain
whatever values they want and replace the ones they do not. On submitting the
values, the values will be added to the table.

To remove an entry, it must be deleted from both the storage and the
table, so we call their respective functions to perform the task:

```javascript
else if (op == "remove")
{
    if (confirm('Are you sure you want to remove "' + entry.name + ' ' +
entry.gameName + '" from your local storage?'))
    {
        Player.storeRemove(entry);
        Player.tableRemove(entry);
    }
}
```
Removing from storage involves a single command that finds the entry by its id and removes it:

```javascript
storeRemove: function(entry)
{
    window.localStorage.removeItem("Player:" + entry.id);
},
```

Removing from the table is just as easy, as we only need to remove a row, which is a child of the table:

```javascript
tableRemove: function(entry)
{
    Player.$table.removeChild(document.getElementById("entry-" + entry.id));
},
```

The final action to be performed on the entries is to actually choose one to play with. We only want one account active at a time, as we only want one person to receive credit for story progression, so we set the selected entry as the current game; we need only the id and not the entire entry. We then proceed to investigate what progress the user has made in the game, if any:

```javascript
else if (op == "play")
{
    Player.setCurrentGame(entry.id);
    Player.loadLevel(entry);
}
```

We have the id of the entry to be used, so when we iterate through local storage, parsing the stored strings into JavaScript objects, we look at every entry id and...
see if it matches our passed in id. If it does, that’s the account we want to write, so we set it to the current game, and set the current game field of all other accounts to false. We finally save all the changes:

```javascript
setCurrentGame: function(id)
{
  for (var key in localStorage)
  {
    if (/Player:\d+/.test(key))
    {
      var x = window.localStorage.getItem(key);
      var y = JSON.parse(x);
      if (y.id == id)
      {
        y.currentGame = true;
      }
    else
    {
      y.currentGame = false;
    }
    window.localStorage.setItem("Player:"+ y.id, JSON.stringify(y));
  }
}
```

After setting the current game, and setting the values of all other stored entries to false, we want to load the level of the current game. So that we do not have to iterate through all local storage again, we cheat by sending in the same entry we
had sent into setCurrentGame. Of course, since this is a copy that has not yet had its value set to true, we do that manually. Before we finally redirect the user to the right game level url, we check if it is a new entry. If it is new, the level field will be zero, so we must increment it so it is directed to the first level:

```javascript
loadLevel: function(entry) {

    if (entry.level == 0) {

        entry.level++; 
    }

    entry.currentGame = true;
}
```

We save any changes before redirect:

```javascript
window.localStorage.setItem("Player:"+ entry.id, JSON.stringify(entry));
```

Finally, we run through a switch statement to determine what level to send the user to. A few examples of possible levels:

```javascript
switch (entry.level) {

    case 1:
        window.location.assign("storyScreen.html");
        break;

    case 2:
        window.location.assign("hallway.html");
        break;
```
case 3:
    window.location.assign("hallway2.html");
    break;
case 4:
    window.location.assign("game.html");
    break;
}

**Game Loop**

Local storage is essential to saving and restoring the game state, as well as keeping track of current game progress. Because html is stateless, the moment we move from the accountManager.html page to the first game page, we lose all memory of previous changes. To maintain a sense of persistency, we must reload any important saved information from storage.

**Current Game.** The first entry we want to load is the one labeled as the current entry, which is accomplished by iterating through the stored entries, parsing them, and finding the one that has currentPlayer set to true. Then, for convenience, we store the entry in memory, as a member of the gameLoop object for easy access later:

```javascript
setCurrentGame: function()
{
    for (var key in localStorage)
    {
        if (/Player:\d+/.test(key))
        {
            var parsedPlayer = JSON.parse(window.localStorage.getItem(key));
        }
```
if (parsedPlayer.currentGame)
{
    gameLoop.currentPlayer = parsedPlayer;
}
}
}

Once we have the entry in memory, we can check what level the player is in, what story events have been finished, and what inventory the player has.

**Level.** Most concerns regarding level are handled by JavaScript once we have the level in memory. This works well until the user wants to either move to a new page or a significant even occurs. In those cases, we make use of local storage. For example, when we move to a new page, we call upon the function `setLevel`, which takes the current level number as an argument and checks it against the stored level. Because this function is only called upon during legitimate transitions, the system knows that the user must have been sent to a higher level, so if the current level is higher, we set the stored level to the higher level. We save this new higher level in storage and then forward the user to this new page:

```
setLevel: function(levelNumber)
{
    if(levelNumber>gameLoop.currentPlayer.level)
    {
        gameLoop.currentPlayer.level = levelNumber;
    }
```
Besides storing the player’s game level, another important task performed by local storage is storing the player’s inventory. This can occur in two situations, either when the inventory is empty because it has never been used, or when there is already an item in storage and the user opts to replace the old item. In either case, we call the function `storeInventory` with the id of the item to be stored. We assign this id first to the JavaScript and then to the local storage:

```javascript
storeInventory: function(itemID)
{
    gameLoop.currentPlayer.inventory = itemID;
    window.localStorage.setItem("Player:" + gameLoop.currentPlayer.id, JSON.stringify(gameLoop.currentPlayer));
    gameLoop.idItemLoadedToInventory = itemID;
},
```

A final essential duty of local storage is to check whether players have fulfilled certain game tasks. For example, players will not be allowed to move past the second hallway until they have had the first meeting with the i. This is checked in the gameloop and uses the entry object stored in the gameLoop object:

```javascript
checkFirstEncounter: function()
{
`
if(gameLoop.currentPlayer.storyProgressMarker == "iMet")
{
    $('frontDoor').html('<a href="javascript:gameLoop.setLevel(5)" style="text-decoration:none"><img src="images/background/frontOpenDoorFinal.jpg" border="0"></a>');
}
}

The door will appear closed until the player meets the i, and then the player will see an open door when the value iMet is true. The value iMet is set when players finish their first meeting with the i in game.html:

    case 5:
        window.gameLoop.currentPlayer.storyProgressMarker="iMet";
        window.localStorage.setItem("Player:"+window.gameLoop.currentPlayer.id,
        JSON.stringify(window.gameLoop.currentPlayer));
        window.location.replace("hallway2.html");
        break;

This fifth case represents the final dialogue option with the i, after which the user is sent to the hallway with the previously closed door, now to find the door open.

WebGL

WebGL utilizes the same standards used in traditional openGL, including access to the programmable graphics pipeline via shader programs. The big difference is that WebGL provides graphics operations to JavaScript running in the browser whereas traditional openGL provides graphics operations to
programs that link directly to the operating system’s OpenGL library. JavaScript
uses WebGL operations to render scenes through HTML canvas elements. In
order to render the scene, there are a few tasks that must be fulfilled, such as
initializing the canvas, shaders, buffers, event handlers, and textures. After that,
the scene is drawn in a loop, rendering each frame one at a time. We will need to
follow each step in order to create the motion-tracking eyeball mentioned in
previous chapters. The task is accomplished by creating a three-dimensional
sphere, texturing it, creating lighting, and responding to user input. Each step is
accomplished by functions in the webGLstart() function:

```javascript
function webGLStart()
{
    var canvas = document.getElementById("iWorld");
    initGL(canvas);
    initShaders();
    initBuffers();
    initTexture();
    gl clearColor(0.0, 0.0, 0.0, 1.0);
    gl.enable(gl.DEPTH_TEST);
    canvas.onmousedown = handleMouseDown;
    canvas.onmousemove = handleMouseMove;
    tick();
}
```
Canvas

WebGL graphics are built on top of HTML canvas elements, so the first step in JavaScript is to get a reference to a canvas element. The canvas element, along with its dimensions, is defined within the html body:

```html
<canvas id="iWorld" style="border: none;" width="500" height="500">
</canvas>
```

Since the element is part of the DOM, a reference to it can be retrieved by JavaScript and stored in a variable:

```javascript
var canvas = document.getElementById("iWorld");
```

Once we have a reference to the canvas element, we can obtain a reference to its associated OpenGL context.

```javascript
gl = canvas.getContext("experimental-webgl");
```

As of this writing, WebGL’s implementation of canvas is supported on the latest browsers, but not on most phones or tablets. Before proceeding, we need to verify that the OpenGL context exists.

```javascript
if (!gl)
{
    alert("Could not initialise WebGL, sorry :-(");
}
```

Assuming the OpenGL context exists, our next step is to set the dimensions of the viewport it will render into, which we get from the width and height attributes of the HTML canvas element:

```javascript
gl.viewportWidth = canvas.width;

gl.viewportHeight = canvas.height;
```
Shaders

Once the canvas is initialized, the next step is to set up a number of functions essential to drawing the objects in the 3D world. Most importantly, setting up the vertex shaders needed to determine the lighting of 3D objects in the world, as dictated by the location of each item’s vertices. We also need to initialize the fragment shaders (also known as pixel shaders) required to manipulate individual pixels when they are translated into the 2D viewpoint. To do this, we call the initShaders function, and retrieve the type of shader desired.

Fragment. In JavaScript, we obtain a reference to the source code for a shader program by calling the getShader function with a reference to the context and the id of the script element that references the shader source code:

```javascript
var fragmentShader = getShader(gl, "shader-fs");
```

This reference is passed in to the getShader function, which in turn stores it in the variable shaderscript:

```javascript
function getShader(gl, id)
{
    var shaderScript = document.getElementById(id);
    if (!shaderScript)
    {
        return null;
    }
}
```

The shader file is attached to the HTML document through the script element.
Though the file looks like a simple JavaScript script, it is actually written in GLSL (OpenGL Shading language) and intended to run on the client’s GPUs. In our example, the file shader-fs contains the source code for a vertex shader. The purpose of this shader is to set up the color of each pixel when given a number of factors including the light-weight on the objects in that pixel and the particular textures applied to an object.

```glsl
vec4 textureColor = texture2D(uSampler, vec2(vTextureCoord.s, vTextureCoord.t));
gl_FragColor = vec4(textureColor.rgb * vLightWeighting, textureColor.a);
```

**Vertex.** The vertex shader is more complex in the sense that it requires computation of 3D world objects, as opposed to the 2D pixels of the fragment shader. Even though the fragment shader is loaded first in our code, in execution, the vertex shader actually runs first. In the graphics pipeline, we define buffers, attributes, and uniform variables using JavaScript, and those values are then passed on to the vertex shader. The vertex shader, in turn, has its varying variables modified through rasterization and then passed to the fragment shader [10].

Similar to the vertex shader, we use the `getShader` function to obtain a reference to the source code for the fragment shader:

```javascript
var vertexShader = getShader(gl, "shader-vs");
```

The source code for the fragment shader is also loaded through a script element.
The main purpose of the following vertex shader is to emulate lighting by using the Phong Light Model, which uses ambient, diffuse, and specular light sources to mimic lighting reflection. Without this reflection, all polygons in a model would be lighted equally, as we set in case lighting is not turned on:

```glsl
if (!uUseLighting)
{
    vLightWeighting = vec3(1.0, 1.0, 1.0);
}
```

To display the lighting correctly, we must store the position of every vertex in order to determine the vector normal of each polygon. This vector represents the direction a polygon is facing and conveys the angle at which any given light will hit, or whether any light will hit at all. Next, it is necessary to determine the direction and color of any light source:

```glsl
vec3 transformedNormal = uNMatrix * aVertexNormal;
float directionalLightWeighting = max(dot(transformedNormal, uLightingDirection), 0.0);
vLightWeighting = uAmbientColor + uDirectionalColor * directionalLightWeighting;
```

Once the shader has been retrieved, we tell openGL what type of shader it is, and hand it off to be compiled, and then returned:

```javascript
var shader;
if (shaderScript.type == "x-shader/x-fragment")
{
    shader = gl.createShader(gl.FRAGMENT_SHADER);
}
```
else if (shaderScript.type == "x-shader/x-vertex")
{
    shader = gl.createShader(gl.VERTEX_SHADER);
}
else
{
    return null;
}
gl.shaderSource(shader, str);
gl.compileShader(shader);
...
return shader;

Those returned shaders are attached to a WebGL storage object called a program. This program runs on the graphics card and can hold one vertex shader and one fragment shader:

shaderProgram = gl.createProgram();
gl.attachShader(shaderProgram, vertexShader);
gl.attachShader(shaderProgram, fragmentShader);
gl.linkProgram(shaderProgram);

Once the WebGL object and the GPU-running shader program are linked, we can then establish a reference that the vertex shader needs for every vertex position, texture coordinate, and vertex normal [10]. An example for retrieving an attribute:
shaderProgram.vertexPositionAttribute =
gl.getAttribLocation(shaderProgram, "aVertexPosition");
gl.enableVertexAttribArray(shaderProgram.vertexPositionAttribute);

We can also retrieve any uniforms stored in the shader program. Unlike attributes, uniforms do not change per vertex. They contain the same values for each vertex. An example of retrieving a uniform:

shaderProgram.pMatrixUniform = gl.getUniformLocation(shaderProgram, "uPMatrix");

Buffers

All the data sent to the shaders has to be held somewhere before rendering, so the buffers must be initialized. Of particular importance are the vertex positions for the sphere to be rendered. In WebGL, there is no preset command to draw a sphere, so the geometric shape must be created out of individual polygons.

Sphere. The task is accomplished by first declaring the number of latitudinal and longitudinal bands of the sphere, along with the radius:

```javascript
var latitudeBands = 30;
var longitudeBands = 30;
var radius = 2;
```

Mathematical functions find the angle from the origin to each vertex by looping first through the latitude bands and then through the longitude bands.

```javascript
for (var latNumber=0; latNumber <= latitudeBands; latNumber++)
{
    var theta = latNumber * Math.PI / latitudeBands;
```
var sinTheta = Math.sin(theta);
var cosTheta = Math.cos(theta);

for (var longNumber=0; longNumber <= longitudeBands; longNumber++)
{
    var phi = longNumber * 2 * Math.PI / longitudeBands;
    var sinPhi = Math.sin(phi);
    var cosPhi = Math.cos(phi);
}

It may be noted that the range of the latitude bands is from zero to pi, whereas the longitudes range from zero to two-pi. The reason for this is that the sphere is built like a globe, with the latitude encompassing north through south and the longitude encompassing east to west. Thus, the latitude bands start in the “north pole” and head south, while the longitude bands are created in a clockwise manner. Therefore, there is only one sweep of latitudinal bands, from the top to the bottom, and then the longitudinal bands will cover the rest, encircling the globe three-hundred and sixty degrees, or two-pi.

The trigonometric angles found are then used to locate the three-dimensional coordinates of each vertex, along with their normals, which are needed for texturing purposes:

var x = cosPhi * sinTheta;
var y = cosTheta;
var z = sinPhi * sinTheta;
var u = 1 - (longNumber / longitudeBands);
var v = 1 - (latNumber / latitudeBands);
normalData.push(x);
normalData.push(y);
normalData.push(z);
textureCoordData.push(u);
textureCoordData.push(v);
vertexPositionData.push(radius * x);
vertexPositionData.push(radius * y);
vertexPositionData.push(radius * z);

The x, y, and z values are stored in a queue called normalData in order to determine the vector normal for each vertex. The three-dimensional values computed are multiplied by the radius of the sphere to find the exact position.

The work performed so far has focused on finding the vertices of the sphere. To draw the globe, though, it is essential to specify what vertices will connect to one another to draw the individual triangles comprising the sphere. The sphere will be made of rectangles defined by the vertices, and each rectangle in turn will be made of two triangles. The first triangle is created from the top left corner of the rectangle, the top right corner, and the bottom left corner—respectively first, first + 1, and second. The second triangle is made from the bottom left corner, bottom right corner, and top right corner of the rectangle—respectively second, second + 1, and first + 1 [10]:

```javascript
for (var latNumber=0; latNumber < latitudeBands; latNumber++)
{
    for (var longNumber=0; longNumber < longitudeBands; longNumber++)
    {
```
var first = (latNumber * (longitudeBands + 1)) + longNumber;
var second = first + longitudeBands + 1;
indexData.push(first);
indexData.push(second);
indexData.push(first + 1);

indexData.push(second);
indexData.push(second + 1);
indexData.push(first + 1);

The last task is to bind the vertex data to the buffers, and WebGL’s bindBuffer performs the task. The first argument specifies what type of buffer it is, and the second argument specifies the name of the buffer. For example, the buffer for vertex normals:

```
function useEyeViewData(normalData)
{
    eyeVertexNormalBuffer = gl.createBuffer();
gl.bindBuffer(gl.ARRAY_BUFFER, eyeVertexNormalBuffer);
gl.bufferData(gl.ARRAY_BUFFER, new Float32Array(normalData),
gl.STATIC_DRAW);

    eyeVertexNormalBuffer.itemSize = 3;
    eyeVertexNormalBuffer.numItems = normalData.length / 3;

    Because each vertex has three components, the x, y, and z coordinates, the item size is set to three.

    In another example, we bind the index buffer data:

    eyeVertexIndexBuffer = gl.createBuffer();
```

gl.bindBuffer(gl.ELEMENT_ARRAY_BUFFER, eyeVertexIndexBuffer);

gl.bufferData(gl.ELEMENT_ARRAY_BUFFER, new Uint16Array(indexData),
gl.STATIC_DRAW);

eyeVertexIndexBuffer.itemSize = 1;

eyeVertexIndexBuffer.numItems = indexData.length;

The buffer is declared as an element array buffer, as opposed to a simple array buffer, because it will be used to draw the actual elements—the triangles. Each item is only of size one since each only consists of an index number. When the buffers are read, every set of three indexes comprises a triangle.

Texture. At this point, texturing an object is not much more work since we have initialized the vertex shaders and loaded buffers with all the relevant data pertaining to positioning the texture. All that is needed is to call WebGL’s createTexture function, which automatically loads the texture if provided the right path to an acceptable texture; WebGL only accepts textures with dimensions in the power of two.

eyeTexture = gl.createTexture();

eyeTexture.image = new Image();

eyeTexture.image.onload = function ()
{
    handleLoadedTexture(eyeTexture)
}

eyeTexture.image.src = "images/eyeTexture/wonkyeye.gif";

When the onload function calls handleLoadedTexture, it performs a number of boilerplate, but important tasks. First, it flips the orientation in the y-
axis. Most 2D textures exist in formats where the y-value increases upon heading down the image, but in WebGL, the y-value increases as it heads upwards. The flip flag allows the two systems to be in accord. The next task is to upload the current texture into the graphics card and specify the format:

```javascript
gl.pixelStorei(gl.UNPACK_FLIP_Y_WEBGL, true);
gl.bindTexture(gl.TEXTURE_2D, texture);
```

The next two lines indicate what to do with the textures when they are scaled up or down, i.e. if the user zooms in or out. Textures are essentially a way for the fragment shader to tell what color a pixel should be, and there are three methods that determine the pixel color on scaled images: nearest, interpolation, and mipmapping. The easiest function, the “nearest” function, instructs the system to find the current pixel color by looking at the nearest adjacent pixel. However, the techniques of linear interpolation and mipmapping are more effective. With linear interpolation, if a pixel lies between two different colors, the method finds the mean of the two colors and assigns that value to the pixel. So, if one pixel is black and the other white, the pixel is grey. This technique is particular useful for close-ups of an image. With the “nearest” method, items would appear blocky because pixels near one another would take on the same color as their neighbors, but with linear interpolation, there is a gradual transition:

```javascript
gl.texParameteri(gl.TEXTURE_2D, gl.TEXTURE_MAG_FILTER, gl.LINEAR);
```

Explaining the last line requires delving into the final technique for filtering: mipmapping. This method requires the creation of various smaller copies of the
original texture, each at half the width and length of the last. For example, if our
texture is 512 x 512, the next copy would be 256 x 256, then 126 x 126, and so on. The reason for using this technique is because textures tend to look jagged
the further out they are since less and less detail is paid to applying a large
texture. The solution is to apply a smaller, less detailed texture the further out an
object is, and that is exactly what mipmaps do. The textures are pre-generated
and stored in memory, and then loaded according to the distance. What the
following command does though is not simply mipmapping, but a mixture of all
three techniques discussed so far. With gl.LINEAR_MIPMAP_NEAREST, we are
telling WebGL to pick the nearest mipmap after linear interpolation (Wyman):

```javascript
    gl.texParameteri(gl.TEXTURE_2D, gl.TEXTURE_MIN_FILTER,
        gl.LINEAR_MIPMAP_NEAREST);
```

The final step is to create the mipmap and bind the texture:

```javascript
    gl.generateMipmap(gl.TEXTURE_2D);
    gl.bindTexture(gl.TEXTURE_2D, null);
```

**Event Handler**

Now that the sphere is created and textured, the event listeners that
handle user input are built. In this particular example, it is imperative to know
where the user will click first to establish the base of rotation, and then where the
user moves the mouse in order to determine how much to rotate about the x and
y axis:

```javascript
    function handleMouseDown(event)
    {
```
mouseDown = true;
lastMouseX = event.clientX;
lastMouseY = event.clientY;
}

function handleMouseMove(event)
{
    if (!mouseDown)
    {
        return;
    }
    var newX = event.clientX;
    var newY = event.clientY;
    var deltaX = newX - lastMouseX;
    var deltaY = newY - lastMouseY;
    var newRotationMatrix = mat4.create();
    mat4.identity(newRotationMatrix);
    mat4.rotate(newRotationMatrix, degToRad(deltaX / 10), [0, 1, 0]);
    mat4.rotate(newRotationMatrix, degToRad(deltaY / 10), [1, 0, 0]);
    mat4.multiply(newRotationMatrix, eyeRotationMatrix, eyeRotationMatrix);
    lastMouseX = newX
    lastMouseY = newY;
}

The main concerns are deltaX and deltaY, which store the value of the x and y difference between the last mouse event and the current mouse event. This allows the setting of the rotation matrix the vertex positions will be multiplied by.
After the updated rotation, lastMouseX and lastMouseY receive both the current mouse x and y coordinates, and the new events to dictate further rotation.

Because there is only one object in the sample world, there is one model view matrix and no concerns about pushing or popping the model view stack. There is only one matrix called the newRotationMatrix. This matrix is originally set to all zeroes using mat4.create(), and immediately after is set to the identity matrix. Any coordinate multiplied by the identity matrix equals itself, so this reflects the state the model should be in before the model is rotated. To incorporate any rotations, the new rotation matrix is multiplied first by a 1 x 3 matrix that represents rotations about the x-axis, and then another 1 x 3 matrix representing y-axis rotations. WebGL does not concern itself with how much the vertices have rotated since the last rotation, i.e. it does not care that a model is rotated thirty degrees and then rotated a further forty-five degrees. It simply keeps track of a composite matrix that aggregates all transformations performed and then multiplies the original identity matrix by that aggregate matrix. At every frame, WebGL is redrawing the world anew.

**Draw Scene**

Because the world is constantly changing, there is a loop that perpetually checks for the most recent updates and displays the current frame. Once the current frame is drawn, the scene can finally be drawn:

```javascript
function tick()
{
    requestAnimFrame(tick);
}
```
drawScene();
}

When drawing the scene, one must first define from what position users will be viewing the scene:

gl.viewport(0, 0, gl.viewportWidth, gl.viewportHeight);
gl.clear(gl.COLOR_BUFFER_BIT | gl.DEPTH_BUFFER_BIT);
mat4.perspective(45, gl.viewportWidth / gl.viewportHeight, 0.1, 100.0, pMatrix);

The next step is to set whatever lighting is cast on the 3D world. The info is retrieved from the html and can be set by the user:

<h2>Directional light:</h2>
<td><b>Direction:</b></td>
<td>X: <input type="text" id="lightDirectionX" value="-1.0" /></td>
<td>Y: <input type="text" id="lightDirectionY" value="-1.0" /></td>
<td>Z: <input type="text" id="lightDirectionZ" value="-1.0" /></td>
<td><b>Color:</b></td>
<td>R: <input type="text" id="directionalR" value="0.8" /></td>
<td>G: <input type="text" id="directionalG" value="0.7" /></td>
<td>B: <input type="text" id="directionalB" value="0.7" /></td>

<h2>Ambient light:</h2>
<td><b>Color:</b></td>
<td>R: <input type="text" id="ambientR" value="0.2" /></td>
<td>G: <input type="text" id="ambientG" value="0.2" /></td>
<td>B: <input type="text" id="ambientB" value="0.2" /></td>
The lighting is slightly red to give the white iris a slightly reddish hue, but not so red that it obscures the eye’s textured capillaries. JavaScript passes on these values to WebGL:

```javascript
var lightingDirection = [
    parseFloat(document.getElementById("lightDirectionX").value),
    parseFloat(document.getElementById("lightDirectionY").value),
    parseFloat(document.getElementById("lightDirectionZ").value)
];
var adjustedLD = vec3.create();
vec3.normalize(lightingDirection, adjustedLD);
vec3.scale(adjustedLD, -1);
```

```javascript
gl.uniform3fv(shaderProgram.lightingDirectionUniform, adjustedLD);
gl.uniform3f(shaderProgram.directionalColorUniform,
    parseFloat(document.getElementById("directionalR").value),
    parseFloat(document.getElementById("directionalG").value),
    parseFloat(document.getElementById("directionalB").value)
);
```

The next step is to set the model view matrix to the identity matrix. After which, the camera moves six units back on the z-axis to start drawing, and is then multiplied by the eyeRotationMatrix stored in the buffer:
mat4.identity(mvMatrix);
mat4.translate(mvMatrix, [0, 0, -6]);
mat4.multiply(mvMatrix, eyeRotationMatrix);

The last bit of code involves finally drawing the triangles that comprise our 3D sphere. For example:

gl.bindBuffer(gl.ELEMENT_ARRAY_BUFFER, eyeVertexIndexBuffer);
setMatrixUniforms();
gl.drawElements(gl.TRIANGLES, eyeVertexIndexBuffer.numItems, gl.UNSIGNED_SHORT, 0);
CHAPTER FIVE

ART ASSET CREATION (GIMP)

Because hiring an artist is cost prohibitive, the graphics for the game were created from photos that were then modified to look hand-drawn. Copyright laws on images are strict, to the point where even if an image is altered to the point of being unrecognizable from its original form, the image is still protected by law. Thus, all images must be created by the programmer or procured under a royalty-free license.

Edge Detect

Once a royalty-free image has been obtained, we can apply an artistic filter that mimics a hand drawing. A particularly helpful filter is dubbed "edge-detect." What this filter does is detect any edges in a drawing and emphasize those edges, while downplaying non-sharp features. The purpose of this is to have the outline of items stand out, as one would do when drawing something by hand. In real life, edges are representative of the outlines of things and are usually in black, but the algorithm makes things black to de-emphasize details, so to emphasize the edges, the algorithm flips their color, making any black edges white. Of course, in a picture composed of pixels, the fact that a pixel is black does not always indicate that it is an edge, since a swath of black pixels could comprise a black object, so to know whether something is an edge, the algorithm looks at any color intensity variation. If there is a small gradient
between neighboring pixels, there is a high chance that we are looking at the same material in a picture. However, when we see a drastic color change between pixels, say bright red to black, there is probably a border present [11].

This variation can be represented mathematically. An image’s component pixels can be grouped together into rows and columns. If we think of each row as something that may be represented by a linear equation, with the x-axis representing the pixel location and the y-axis representing the color intensity in that location, we can graph the equation. When there are drastic changes in the graph, by finding the first derivative; we may note when there is a drastic change in color variation because the slope will change quickly. These intervals are where there is an edge, and they will be marked by a thick white line [11].

For more subtle gradients, another algorithm is required. Take for example a photo of a sky with clouds in it. When the drawing transitions from the light sky to the light cloud edges to the dark gray cloud inner core, we do not want thick edges delineating the transitions. We want thinner edges that denote a more subtle transition, not the drastic change denoted by the first derivative. The demarcation of the subtle change is accomplished by using the second derivative [11].
Another way that GIMP helps modify images is through layers. Sometimes we may want one image to be the background to another image, or cut and paste only part of an image to enhance another. It is helpful if we can layer these changes one on top of another, deleting or altering any layers without having to alter the original image. The Gimp native xcf format allows us to do this. If we have a background of a mountain, we may copy and paste the picture of a couple so that it looks like they are on the mountain, and we can easily erase that layer of the couple and replace it with a picture of a Saint Bernard. Then we can
scale that dog to look like a giant, all without altering the original mountain background.

Save Format

GIMP is not only helpful for altering images, but also for creating them. Of particular use is the ability to export a picture to a given format. Web technologies are intended for static art, usually of uniform rectangular size. However, for draggable items in a game, it is sometimes required for items to be of different shapes and sizes. Items that are not just blocks, like drawings of humans, will have many different edges and contours, and even though there edges will still have to be defined by rectangular or square boundaries. We can make those extra boundaries invisible by setting the transparency of an object.

Not all formats support transparency though. So it is helpful that GIMP allows many formats, including png, which will save an image with a transparent background. That way, when a user drags a picture of a human in the game, they feel like they are dragging the human along, not the human and some filled in rectangular background.

Matrix Transformations

A final use of GIMP is matrix transformations that allow perspective shift. When an image is moved or scaled, the transformations are straightforward because an image will still be on the same plane and will usually have the same relative
dimensions. Difficulty arises when we try to transform a flat image to a plane not perpendicular to the user’s view. The relative dimensions would have to change because the parts of the picture that are farther from the viewer would be smaller and the parts closer to the viewer would be bigger. An image is simply slanted, and not transformed through a matrix, will have one side smaller than the other, but it is not enough to stretch the image, as that will simply make the image look warped. The dimensions of the image must be mapped to the new plane, and GIMP’s Matrix Transform allows us to complete the task.

Figure 11: Original Door  Figure 12: Perspective-Shifted Door
CHAPTER SIX
SAMPLE TEST RUN

Welcome Page

The first introduction the user will have to the game is the welcome.html page, which uses JQuery to prompt the user to continue to the game, and gives the player a hint of the i’s behavior as the eye opens while the page transitions to a new page:

![Welcome Page](image)

Figure 13: Welcome Page
Account Manager Page

After the introductory page, the user is redirected to the accountManager page, where the user may create, edit, or delete any accounts on the browser. Upon first arriving at the page, the page will be empty, but it will subsequently be populated by each new account the user creates:

![Account Manager Page](image)

<table>
<thead>
<tr>
<th>ID</th>
<th>Name</th>
<th>Game Name</th>
<th>Level</th>
<th>Current Game</th>
<th>Inventory</th>
<th>StoryMarker</th>
<th>Actions</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>john</td>
<td>game1</td>
<td>1</td>
<td>false</td>
<td></td>
<td></td>
<td>Edit</td>
</tr>
<tr>
<td>2</td>
<td>john</td>
<td>game2</td>
<td>4</td>
<td>false</td>
<td>chair</td>
<td></td>
<td>Edit</td>
</tr>
<tr>
<td>3</td>
<td>amy</td>
<td>game1</td>
<td>14</td>
<td>false</td>
<td></td>
<td>RileyMet</td>
<td>Edit</td>
</tr>
<tr>
<td>4</td>
<td>amy</td>
<td>game2</td>
<td>6</td>
<td>true</td>
<td>meshSpider</td>
<td>iMet</td>
<td>Edit</td>
</tr>
</tbody>
</table>

Name: 
Game Name: 

Discard  Save

Figure 14: Account Manager Page

Story Screen

When the user first starts the game, they will be introduced to the story by the story screens. The user may proceed to the next story segment by
pressing the “next” button, or may return to earlier story text by pressing the “back” button. All these story screens occur on the same url, without any page transition:

Figure 15: First Story Screen. Hand Drawing by Kimberly Cooprider
Hallway Screen

Hallway screens serve as transitions between levels. The player will be able to enter any open door, denoted by a black opening. There will be doors the user can enter by either going up, left, or right. A return to previous pages is also possible using the arrow shaped back button. A locked door will mark any doors that are not yet open, and the player will be able to decipher what doors can be unlocked through looking at the map. If they notice that there are no rooms or hallways in a given direction, then that door will never open, but if there is a path through a given door, then the user will only need to fulfill certain game criteria to have the door open:
Figure 17: Hallway with Closed Door

Figure 18: Hallway with Open Door
Game Screen

Most of the challenges the player will face are in the game levels. In these screens, the user will be charged with completing certain tasks, such as finding the right item to drop on a receptacle or inputting the right password on an input screen. In the sample screen here, the user must move the chair under the counter:

![Game Screen Upon Page Load](image)

Figure 19: Game Screen Upon Page Load

After solving the given puzzle, the user is rewarded with dialogue that develops the story line:
Inventory

Most draggable items are useful only in the level they are found, but some might be needed for later levels. These items can be stored in the inventory by clicking the inventory button, “inv.”, and dragging the item to the inventory:
Figure 21: Inventory Appears on Clicking Inventory Button

Once an item has been dropped into the inventory, it can be accessed in any other page:
Though not a necessary part of the game, the moving i screen is a fun demonstration of what WebGL is capable of. When a user clicks on the iris, the eye pupil follows the mouse. The cursor does not appear in a print screen, but in the initial position it would be in the middle of the iris:
Click on the iris to have the eye follow the mouse while within the canvas.

Figure 23: Initial Eye Position

The cursor is moved to the top left:
Figure 24: Eye After Mouse Movement

Test Cases

The following test cases illustrate the precautions taken to ensure the game continues to function in case the user attempts to use it improperly.

Proceeding to Incorrect Level

If a user attempts to proceed to a level beyond what they have reached so far, the system should revert them back to the right level. Therefore, if hypothetical player John, who has only progressed to level 4, tries to advance to
the level 7, the url http://www.aldolewis.com/TheiProject/hallway4.html, he should be diverted back to the correct url, http://www.aldolewis.com/TheiProject/game.html:

![Image](http://www.aldolewis.com/TheiProject/hallway4.html)

Figure 25: User Attempting to Skip to Higher Level

This is indeed what happens once the user presses enter after typing in the incorrect level url:
Another behavior players may exhibit that has to be handled is using the wrong item to fulfill a game task. For example, in the fourth game challenge, the user is tasked with finding the right item to drop on the pedestal. Of three possible items, only one has the scope required to activate the event listener and complete the challenge. If a user drops the “falseTrinket1” item on the pedestal, nothing should happen:

Figure 26: User Redirected to Right Level
Figure 27: Incorrect Trinket Dropped on Droppable

However, dropping the correct item, “trinket,” causes the event handler to fire and the game progresses:
We must commend you on your work! That leaky pipe had effectively separated the complex in two! All that water--
Conclusion

The intent of this project was to create a simple point-and-click game that could be run on any system. It uses web technologies in order to reach a wide audience, whether the player uses a regular laptop, tablet, or phone. The game can be accessed without any plug-ins, which is accomplished by using web technologies such as JQuery, local storage, Touch Punch, and WebGL. JQuery facilitates server independence by building upon the web standard javascript, which allows dynamic user interaction on the client. The advantage of using JQuery, though, is the fact that it facilitates the use of javascript, making it easier to use through truncated standardized commands that work uniformly on all browsers. This works well on easier tasks such as making non-hyperlink items clickable and simplifies difficult tasks such as scheduling drag and drop interaction. Of course, JQuery’s more sophisticated functionality does not work on mobile devices. The company does have a library called JQuery mobile that enables click event equivalents on phones and tablets, but it does nothing for dragging and dropping, and that is where Touch Punch comes to the rescue. Because the library supplements all mouse events with touch events, anything that can be performed on a personal computer can be accomplished on mobile devices.
The other two pieces of the project, Local Storage and WebGL, work well on all personal computers, but one is not fully supported, and that is the reason it was only used as a sample. Local Storage is a standard of html 5, and allows one to store the game state on any digital device without the need to resort to a server. Instead of having to log in, a user simply uses a browser of choice to find that any progress made in the game has been saved automatically. This feature is platform and browser independent, so it works everywhere. WebGL also relies on an html 5 web standard, canvas, but that particular standard is not universally supported, and as a consequence, neither is WebGL. As of this writing, the newest version of Internet explorer has barely begun to offer support to the graphics library on personal computers, so one might image how difficult it is to find a mobile device that supports it. So far, I have only found it supported on the latest iPad version. However, it is only time before WebGL gains widespread support. After all, OpenGL has been supported on mobile devices for a while, so it should not be long before it is clear that WebGL is supportable. Whatever graphics pipeline is used for OpenGL can be used to transmit GPU commands through the Web using javascript, and at that time WebGL can be fully implemented into the game.

Lessons Learned

In developing this game, I have learned the strengths of using libraries, instead of trying to reinvent the wheel. After some months of using raw
JavaScript to perform simple tasks such as selecting an item and dictating the behavior of the item after the item is clicked and the mouse moved, I realized the difficulty of scheduling the order that event handlers should fire. Though I made progress in the endeavor, it was ultimately too time consuming. The libraries also helped in dealing with different browsers. For example, in Internet Explorer, the x-coordinate of the cursor position is retrieved by event.clientX, but Mozilla Firefox uses event.pageX [12]. The differences are smoothed over by JQuery, which has programmers calling one command for both browsers.

Thankfully, tips to solving these problems can be found on the Web. For instance, an example illustrating drag and drop functionality using javascript can be found on Emanuelle Feronato’s website [13]. The code is excellent, but it also makes clear the difficulty in scheduling events, as even understanding the concerns between different browsers was difficult to understand, and that is where a programmer’s best friend, Stack Overflow, comes in handy [14]. The site hosts comment boards where apprentice developers can asks questions and receive help from experienced programmers. Most senior members are helpful enough to even look at posted code and provide possible solutions. Though I never posted any questions, the variety of chronicled concerns encompasses any questions one might have. Reading those posts, I knew I was not the first to struggle with those problems. Another site I found helpful was Code Academy, which aided in understanding JQuery and its conventions. Any concerns not covered by the Code Academy tutorials were covered by the JQuery website,
which provides both explanation and sample uses of their product. In order to debug, the debugging tools in different browsers were helpful, as were the sites that provided information on how to use them [15]. From there, it was an iterative process of learning the capabilities of a language, experimenting, hitting a wall, and looking for help—which entailed learning new capabilities of the language.

Eventually, when the gameplay elements were settled, it was time to assure the game could save the game state. The initial solution was to use cookies to track if a user had begun to play the game, and if so, how far the gamer had progressed. Such a solution was inelegant, outdated, and only allowed one person to play at a time. When the new solution of using local storage was required, the website Dive Into Javascript was extremely helpful. The owner had developed a program using local storage that allowed users to create an address book. The beauty of it is the way it was programmed, with a unique index that ensures no entry is duplicated. The code is explained fairly well, but a number of the commands he uses are sophisticated, so research was needed to decipher them. Again, stack overflow was helpful, but so were the W3 school tutorials that explain the different aspects of javascript and local storage in depth. Once I understood every command, I was able to modify the sample to my purposes.

The WebGL sample, like any graphics programming application, was more difficult to understand. I am thankfully more familiar with graphics programming than javascript, but I still had to learn how to use WebGL. An
extraordinary resource that aided the endeavor was Tony Parisi’s WebGL site. The tutorials progress from drawing simple geometric figures to texturing, to using lighting, to handling user input, and finally to drawing a moon. I read and experimented with each tutorial, though there were still questions. Thankfully, WebGL is very similar to OpenGL, and many of the original NeHe tutorials helped explain what was happening in discrete sections of the samples [16]. To emphasize the helpfulness of OpenGL’s similarly to WebGL, it should be noted that most of the questions I had of the WebGL tutorials were answered in message boards regarding OpenGL. For example, when trying to understand the function `gl.enableVertexAttribArray`, I was unable to find any WebGL explanation, but I found plenty of comments about the OpenGL equivalent.

Two final resources essential for the game are those for Touch Punch and royalty free images. Touch Punch is written by Dan Furfero, and his site provides plenty of explanation for the use of his library as well as a discussion of inspirations during development [17]. For royalty free images, one may visit Morgue File [18]. The site has a comprehensive search engine that provides artists with images ranging from doors and temples to peacocks and models.

**Future Direction**

The game has the architecture for scaling the size easily. There are a number of items that could be implemented without much more work in order to enhance the game experience.
Manifest File

Even though the game runs with limited server interaction, it is possible to build the game with even less server interaction while still using web technologies. In order to do this, the developer would need to create a manifest file that would store all changes to the game state, and any new web pages would be loaded from the client, not the server.

API use

The game is designed to run through the Internet-connected browser, but if it had a manifest file, it could be downloaded completely like an app and then uploaded to phone-specific game hubs like Google Play. Currently, the game is organized independently of any API, but to contribute to Google Play, the easiest way is to organize the game using Eclipse.

Level Development

The game presented here serves more as a first-fifteen minutes proof-of-concept rather than a full-fledged game. Even with its simplicity, completing a full game would require a dedicated team of artists and a writer. Right now, the game takes place in only one sector of the entire maze the player is exploring; the later mazes are only foreshadowed in the story dialogue.

WebGL Game Incorporation

When WebGL becomes standardized, we might see other libraries, like JQuery, interact with it. There would need to be some restructuring of the game code, but both libraries could be used organically in the game. For example, the
sample eyeball could be a part of the map, and whenever the user moves the mouse to drag items, the eye would follow the cursor movement.

**Server Interaction**

Because of the client browser emphasis, the game state is not stored on the server. In future iteration, the user might be offered the option to save to a server in order to access the saved game on any computer, as opposed to only one computer. Alternatively, progress could also be saved on the cloud.
APPENDIX A

HIGH DESIGN DOCUMENT
Reach into a world of long-lost technologies to discover the only means of saving all you know.

Aldo Lewis
Based on a Collaboration with Mark Chapman for CSCI 440
Dr. Concepcion
Name
The i

Platform
Web Browsers

Genre
Puzzle/Adventure

Target Audience
Casual game players ages 14+

Mission Statement
I want to create a game where players can explore vast technological ruins in an interactive way. By selecting and manipulating objects using a drag-and-drop interface, users will interact with technology-driven puzzles.

Player’s Role
Players take on the role of Atticus, a man trying to find some way of saving his family and best friend from the effects of a plague. Atticus finds his way into a technological ruin of decades past named Escher, where he searches for a cure, while simultaneously discovering why the old technology was lost to humanity and interacting with a previously dormant AI personality, “i”. He will have to manipulate these old technologies to find his way through the sprawling labyrinth near his small village.

Story

Background
In the future, mankind begins to see the internet as all all-encompassing repository of knowledge. Every man, woman, and child has a chip implanted in his or her brain that allows the upload of every waking memory, thought, and experience. The hope of this project is to both democratize knowledge and to allow humans to be capable of creating genuinely new ideas.

Unfortunately, such omniscience and super-connectivity is too much for humans to handle. Private residences become technological opium dens as people become addicted to the lives and experiences of others. After all, why live one’s life, when one can live the life of a rockstar? Even if people managed to evade the pointlessness of addiction, their bodies still suffered from years of misuse –obesity and heart disease plague the human blobs of the future. Those who are able to contribute to society find themselves inundated with the increasing amount of knowledge, and many become insane. All the while every thought and experience is documented.

Eventually people begin to accept the consequence of linking to the network. The few survivors capable of disconnecting from the Web are incapable of prospering in a world without technology and are cast back into a stone age.

A thousand years later, we find that mankind has begun to harness old technologies, but they are still in a dark age. People know little of their ancestors’ reliance on computers, and the only memories that remain are in superstitious admonitions. Religious leaders preach that an evil spirit lives within the old technology, and that the technological empires of the past should be avoided.
Past

As a child, the main character, Atticus, lives near the technological ruins of a prominent city. The ruins are labeled as a restricted zone, but over the years security becomes lax.

Through a dare, Atticus finds himself in the city, which is like a maze because it was catered for interconnectivity instead of human transversal. As he gropes through the dark and deserted alleys, he finds himself falling and breaking his leg. Hopeless, and unable to find an exit, the old technologies reboot and show him a holographic memory. The memory details a nano-technological gun that heals his broken leg.

Traumatized, he suppresses the incident.

Present

Years later, a plague sweeps his village. He watches helplessly as families are decimated, but it is not until the disease reaches his doorstep that his repressed memories resurface. When his wife and best friend become ill, all his fears and superstitions are swept away, and he considers a journey into the forbidden ruins. Surely, some of the technology must help him fight the plague.

Characters

<table>
<thead>
<tr>
<th>Atticus:</th>
<th>A young doctor and family man who</th>
</tr>
</thead>
</table>
suppresses his memories of a childhood incident that occurred in the ruins. The fact that he has never been sick, along with his inborn empathy for others, has led him to become a doctor. He has a sharp intellect that allows him to learn and think quickly. His outspoken nature has made him a natural enemy of the church, which often sends spies to keep track of Atticus’ method, including his forays into new technologies.

<table>
<thead>
<tr>
<th>i: A collective consciousness that remains from a time mostly forgotten. In it lie the wisdom, knowledge, fears and neuroses of a whole civilization. The fact that the ruins are so well preserved implies that there might be a corporal body that maintains the ruins.</th>
</tr>
</thead>
<tbody>
<tr>
<td>Hand Drawings by Robert Redd</td>
</tr>
</tbody>
</table>
Amy & Jake:

Atticus' wife and son. Being the daughter of a bishop, Amy is perpetually aware of the dangers Atticus faces by opposing the church, but her faith in his judgment ultimately wins out.

Jake plans to follow in Atticus’ footsteps, and he is already a competent physician assistant. However, there is the fear that he might have been the one who contracted the plague and brought it into the household.

**Primary Game Mode**

The primary game mode will focus on exploring and traversing a 2D world filled with ancient technology. Players will interact with all the elements in the world by using the drag-and-drop functionality.
Players can select objects and manipulate them with a click, and then drag objects to different places as needed. If the object is tethered to the rest of the environment, it will not move further than its tether.

Puzzles have the capacity to be solved in different ways, with different results. A bad solution to a puzzle could have Atticus opening a door that leads to a longer path to his next objective, a path that might have more puzzles. Another bad solution could have the same effect on the environment and progress, but hurt Atticus in the process.

Another example is that of a broken monitor next to a computer. There is a computer hooked up to a projector, but the computer display has a crack in it, so a projector must be linked to the computer and turned on. However, the projector is too high to reach, so the player will need to pick up a chair and move it under the projector to reach it. The player will find that the projector has a blown bulb, so the player must rummage around the room to find a bulb, drag the bulb to the projector, and get the screen to display the information needed. In gameplay, this would require the player to identify that the computer is hooked up to a broken display, identify a projector, move a chair using the interactive interface, move the projector’s cords to the computer, explore the environment to find the missing bulb, and finally move the bulb to the projector.
Game Objectives

Find the cure for Atticus dying family.
Find out what “i” actually is.

Secondary goals include:
Solving puzzles in order to open up and access different areas of the game world.
Learn more about how and why the technology-driven society collapsed.

Pick up and Play / Game Saving

In order to facilitate a pick-up-and-play nature to this game, progress should not be stymied by arbitrary, time-intensive goals, but should instead focus on the intuitive nature of its puzzles.

Also, saving should be able to occur everywhere and the player should be able to pick up exactly where they left off.

Additional Sample Puzzles

In one puzzle, the player will know that the terminal to a computer can be traced by a wire that runs through separate apartments. As such, the player will have to
search through several different rooms. Each room will require different requirements to access. One room will require the user to find and drag a crowbar to a door to open it. Another will require users to remove a lot of debris. While yet another room will require moving boxes in the room to make space in the room for accessing the terminal.

In another puzzle, the user will have to access a database containing various incidents of plagues.

**Game Progression**

The critical game path takes the player throughout the majority of the technological labyrinth. Progress is controlled through the linear gathering of information about the game world and solving puzzles. Upon concluding each event in the critical path, players will be urged to go to a new location and work through the events that await them there. Once every puzzle on this path is completed, the game will be finished.

Outside of the critical path, players can solve additional puzzles to open up different areas in the world and explore content they have already passed through. In doing this, the player is rewarded with additional information about the history of the labyrinth, its time of creation, and the world in general.
years ago. As information is found, the critical path will be lengthened with additional gameplay text.

After a certain amount of information is found outside the critical path, a second main goal of finding out what the AI “i” actually is emerges. This would lengthen the game past its first ending and present the most difficult puzzles and sequences of the game, as well as give more of the story.

**Game World**

The people of the civilization that constructed i had little need for leaving their houses. The only roads remaining are the ancient highways they never bothered tearing down and the railroad-like ducts that carried raw materials and supplies. Because of the hermitic nature of these people, as well as the constant need for living space in a growing population, cities were not meant to be easily navigated. After a thousand years, many of the city structures have been ravaged by time and the expanding quality of nature.

However, if a city were left up to the ravages of nature, it would have been destroyed within a few hundred years. There must be some non-human presence that keeps the ruins active.
APPENDIX B

MAINTENANCE MANUAL
Directories

The project is organized around the different components required, with the program entrance html in the main folder and separate directories for the css, images, and javascript. The html corresponds to each page denoted on the map level array and includes both all the hallways and game pages (e.g. hallway1.html, hallway2.html, game1.html, etc.) It also includes the html for the account manager page and the sample moving eye.

The functionality and resources required for these html pages is in the other directories. The css directory holds the .css code needed for positioning the elements for the html page. Therefore, if a programmer needs to define the position of the open door image in hallway1.html, they need only look at the class type:

```html
<p class="frontDoor">
and find its corresponding definition in hallway.css:

```css
.frontDoor
{
    position: absolute;
    left:348px;
    top:150px;
}
```
The css for all the hallway levels is stored in one file as most of the hallway elements are positioned in the same positions. However, the items in game levels vary drastically, so they each have their own css files.

The images that need to be loaded are stored in yet another directory, the image folder, which is further subdivided into background, buttons, draggables, artwork, and the eye texture. An example of how each folder is utilized can be seen in the game1.html. The first level has a static image that serves as a setting, and the image is stored in the background folder. In front of this image is a chair that the user needs to move around the screen to fulfill the first game task; this image is loaded from the draggables folder. After the chair is dragged to the right position, a number of different elements will be created for interaction: character drawings that represent the game NPCs being spoken with and buttons that allow users to call the next chunk of text. The character drawings are retrieved from the artwork folder, and the buttons are summoned from their namesake folder.

Of course, not all these css and image elements are originally in the html, but they are instead called through javascript files in the script folder, and for game levels, specifically in the level_loader folder. Each game level has individual javascript files that load the content and interactivity of the level. For example, upon loading the game.html file on the browser, one may notice that there are elements on the page that are not declared in the html, like the chair. The reason for this is that the chair element is actually loaded in the game.js:
```javascript
var level = '<div id="chair" class="ui-widget-content"><img src="images/draggable/standalonechair.png" />
```

Furthermore, looking at game.html, one may notice that most of the ids defined have no content (e.g. there is a class “the i,” but it has no content):

```html
<div class="thei">
</div>
```

The content of the id is actually loaded in the javascript file game1.js, as a reaction to other events:

```javascript
$('thei').html('<img src="images/artwork/theiTotemFinal.png" style="border-style:none;">');
```

Therefore, a programmer should be aware of the use of the level_loader folder in the script directory, but there are other essential folders that script directory. Looking at the files in level_loader makes it clear that not all interactivity is defined here, so that requires the other two folders in the script directory, external_libraries and game_controller.

Any functionality not defined by the original programmer is defined in external_library. For instance, in game1.js, the original programmer defines the chair as draggable; this is because it is defined in an external library. This library is called first in the html code before it is used in the game javascript code. The library jquery-1.7.2.min.js makes non-hyperlink items clickable and jquery-ui-1.8.23.custom.min.js makes them draggable. The folder contains the other external libraries required for other files too, such as the glMatrix-0.9.5.min.js and webgl-utils.js needed for the moving eye WebGL sample.
The final directory in the script folder is the gameloop folder, which singularly contains the gameloop, the most important file in the project, as it keeps track of the game state. The javascript file gameloop.js is built as a self-contained object that performs all the functions necessary to keep track of the game state, so it does not call any other files.

Running the Program

The project is written in client browser languages, so most elements may be tested using a text editor, but local storage does not have the ability to work offline since the storage is not built to work on folders, so testing the storage requires a server, which can either be local or online. The testing procedure used by the original programmer was to upload the entire project to a web-hosting site. Any modifications could be made using the site’s editor, and those changes are reflected in the http address of the project. The web developer tools on different browsers help debug any problems in the code.

A more efficient way to run the program and debug might be to use a local server, which avoids Internet connectivity problems and web hosting issues. An API, such as Eclipse with Apache, could be used to run the code on a programmer’s development machine. This will allow the programmer to test any game state functionality such as creating an account, loading the inventory, updating the game state, etc. Other APIs could serve similar functions. For
instance, Microsoft Visual Studio has the capacity to build web pages, and it has a built-in web server.

Performing Alterations to the Code

To make changes to the project, one must determine the depth of modification intended. Changing a picture requires adding the new picture to the correct image folder and altering the path in the code, and modifying the position of the image involves editing the css file. However, more complicated tasks will require more work. For example, in order to add a new level, one must add the html for the level in the main directory, and depending on whether the level is a hallway level or a game level, it may need further additions. Hallways use the same css file, so one need only amend the file for a new level, but game levels have unique positions, so a new file must be created. Also, game levels have unique javascript interaction, and as a consequence, a new .js file must be written for them, while a hallway has no dedicated .js file; thus, no further change is needed.

The modifications needed for the javascript files are the trickiest, as might be seen when attempting to change the game dynamic. An example might be trying to add a gravity effect to draggable items, wherein items that are “dropped” in the air when the user releases the mouse fall to the floor, instead of staying suspended. When the user releases the mouse, the draggable object performs the appropriate JQuery action, to stay where it is left. To have the item move to
an arbitrary floor, say four hundred pixels down on the page, a programmer might retrieve the y-location of where the item was released and animate the draggable so that it moves downward the remaining distance. This is only one potential solution, and it could be performed by writing a single javascript file for a new game level.

Grander changes, such as enhancing elements that track the game state, require altering the game loop. The functionality is in one file, and most of it is self-contained in an object. In the case of a change such as outputting a message if a user has returned to a page that has already been visited, a programmer might want to keep a counter array that contains the number of times a user visits a page. The javascript gameloop object’s fields contain both member variables and methods, so the counter array can be stored as a member variable, while performing the actual counting each time would be done by a method. As a reminder, the checkGameProgress() function of the gameloop runs on each load of a page, and checks what the current page is, so we could use the logic in that function to pass in the name of the loaded page to our counter function. This type of alteration is more difficult as it deals with the algorithmic execution of the game state.

Implementation Strengths

The strength of the current implementation is that it is a self-contained project with all necessary files included and no need to link to other resources.
The project makes use of external libraries, but those libraries are included. A programmer who wishes to make changes only needs to upload the project to a web server, and all the elements work.

Implementation Weaknesses

The weakness of the current implementation is that it is not built using a specific API, so a user who needs to test the project on local server will either need to build their own server using Apache or will need to coordinate all assets to work on a given API such as Eclipse or Microsoft Visual Studio.
REFERENCES


