Multiplayer Browser-Based Game Utilizing JavaScript and WebGL Frameworks

By
C.Y. Tan
Benjamin Naftali
Matthew Tong
Vincent Chan

Computer Science Department
College of Engineering
California Polytechnic State University
2015

Date Submitted: June 13, 2015
Advisor: Dr. Shinjiro Sueda

View metadata, citation and similar papers at core.ac.uk
brought to you by CORE
provided by DigitalCommons@CalPoly

Introduction

Online browser-based games have shown variable success over the years, often falling into one of two common types: free-to-play games made by independent developers relying on peer-to-peer connections or large-scale social games meant to target a casual audience developed by companies such as Zynga Incorporated®. However, there are a few browser-based games that fall out of this norm: massively multiplayer online role-playing games (MMORPGs). While several have risen and fallen over the course of the past decade, we believe that opportunity for a browser-based game in this genre to be larger than ever. With the shift away from Adobe® Flash® and Java® along with the rapidly evolving versatility of Javascript®, Web Graphics Library (WebGL®), and Hypertext Markup Language 5 (HTML5®), the potential for a Javascript-based MMORPG is huge but largely unexplored. Our goal is to push the limits of what a browser can do and to match the graphics, gameplay, and usability that many people had originally thought to be only possible with local client-based games.

We decided to add another twist on our development goal by modifying the genre, furthering how far we decided to venture into barely explored territory. Instead of focusing on a traditional MMORPG, we opted to develop a massively multiplayer online tactical role-player gaming (MMOTRPG) instead. Stemming from the tactical role-playing game (TRPG), a genre best defined as players taking turns to control units on a board, an MMOTRPG seeks to bring this experience online. To further elaborate, a TRPG borrows significantly from the role-playing game (RPG) genre, allowing a player to develop and upgrade units under his or her control by overcoming challenges in the game. However, a TRPG also has many components similar to classic strategy games such as Chess and Shogi, where players have control over multiple playable units and must implement tactical planning in order to be successful. The dimensionality of this strategic thinking in unison with the satisfaction from progression and customization through gameplay is by and large the reason why many players have become fans of this genre.

Historically, in the small number of MMOTRPGs that have been developed, only two games have been able to become financially successful, both of which were created by French company, Ankama Games®. However, much like the traditional offline TRPGs in the past, the MMOTRPGs made by Ankama Games are still player-versus-enemy (PVE) centric, resulting in players working together to defeat computer-controlled enemies. The intention for our browser-based game is to instead have the primary focus be player versus player (PVP) with PVE as a less major component in the game world. This would require that the game be designed for balance, where each potential playable unit and strategy has an equivalent match or counter, while still retaining the appeal of the genre. Due to being PVP oriented, our vision would be that the game would have the potential to be played in a competitive scene. This would mean that some players would find the PVP gameplay challenging and engaging enough to be interested in playing it at hosted events with spectators. Due to the rapidly growing electronic sport industry, we believe that a competitive TRPG has great potential due to the sizable fanbase of the genre and the depth and excitement inherent in the genre.

Development Goals

Our goals for the project, at a high level, could be broken into two large components. The first component is the development of a traditional back-end game server and database that operates the game for each player and the other being the development of a client-based game engine to be run in the player's browser to provide the visuals and interface necessary to playing the game. For all components, development was to be "from scratch," using only third party libraries for necessary utilities.

For the client, the goals were to create a 3D environment where players can interact with whatever peripherals are available to them on their device, whether it be mouse and keyboard, a touchscreen, or a video game controller. To assist in this interaction, the client would also require a 2D interface that allows the player to quickly perform tasks and access information integral to gameplay. Both the 3D and 2D components must be aesthetically pleasing and technically impressive in order to demonstrate the capabilities now inherent in modern web browsers.

For the server, the design goal was to operate independently from the player and keep track of players as they connect to and disconnect from the server. With connected players, the server and client would then communicate on a frequent basis, making gameplay possible. The frequent communications would be the result of all game logic "living" in. the server. This means the server must handle all actions the affect gameplay, and the client only functions to display data present on the server. This approach would greatly increase the consistency and security of the game and naturally prevent players from modifying client side code in order to perform unfair actions.

For the database server, the goal was to intelligently store player data so it remains persistent across instances. This means that each player's data would be stored and updated automatically, allowing the player to disconnect at will and return to playing the game while having his or her gameplay data still available. Additionally, should the structure of database need to be changed in the event of a game update, the database should be designed to be sufficiently versatile for it to accommodate minor to moderate structural changes.

Due to the expectations and aspirations we had for our project, it was likely to be too ambitious. As a result, at the end step of this project, we have yet to fully meet any of our goals. Instead, we had developed a playable prototype and in the future seek to expand upon it. Over the course of the development, we have learned that although the project is definitely feasible, it required significantly more time and a larger team size than we had available.

The Potential of Web Browsers as Gaming Platforms

Prior the advent of HTML5 and WebGL and its widespread implementation, the only practical options for creating a graphical online game were to use Java or Adobe Flash. However, both these approaches required the player to install and regularly update local, proprietary software in order to play them. While many machines generally had these libraries pre-installed, many also did not. Additionally, these games required a number of fairly large packages to accomplish basic tasks that the player would have to load each time he or she wanted to play the game. These games, as a result, were not truly

“portable” and had few advantages over downloadable the games. However, these types of browser-based games were able to hook players more easily due to the idea of minimal commitment to play.

As Adobe Flash was eventually phased out of common web use due to a variety of factors, and Java web applications became unpopular due to inefficiencies, performance issues, and platform limitations, Javascript grew in popularity. With the reintroduction of asynchronous Javascript and Extensible Markup Language (XML®), otherwise known as AJAX, Javascript received a rapidly growing amount of support and development effort, resulting in a swell of the number of libraries. Additionally, due to its standardization and usability across platforms, Javascript became the programming language of choice for modern web applications. Being a scripting language, Javascript could be run quickly and modified on the fly. With AJAX, components could be loaded on demand, significantly saving the amount of time users had to spend waiting in order to view web page content. A number of optimization strategies such as automated minification, post-loading, and dependency bundling have also further improved the performance of projects written in the language, allowing large-scale complex applications to be loaded extremely quickly.

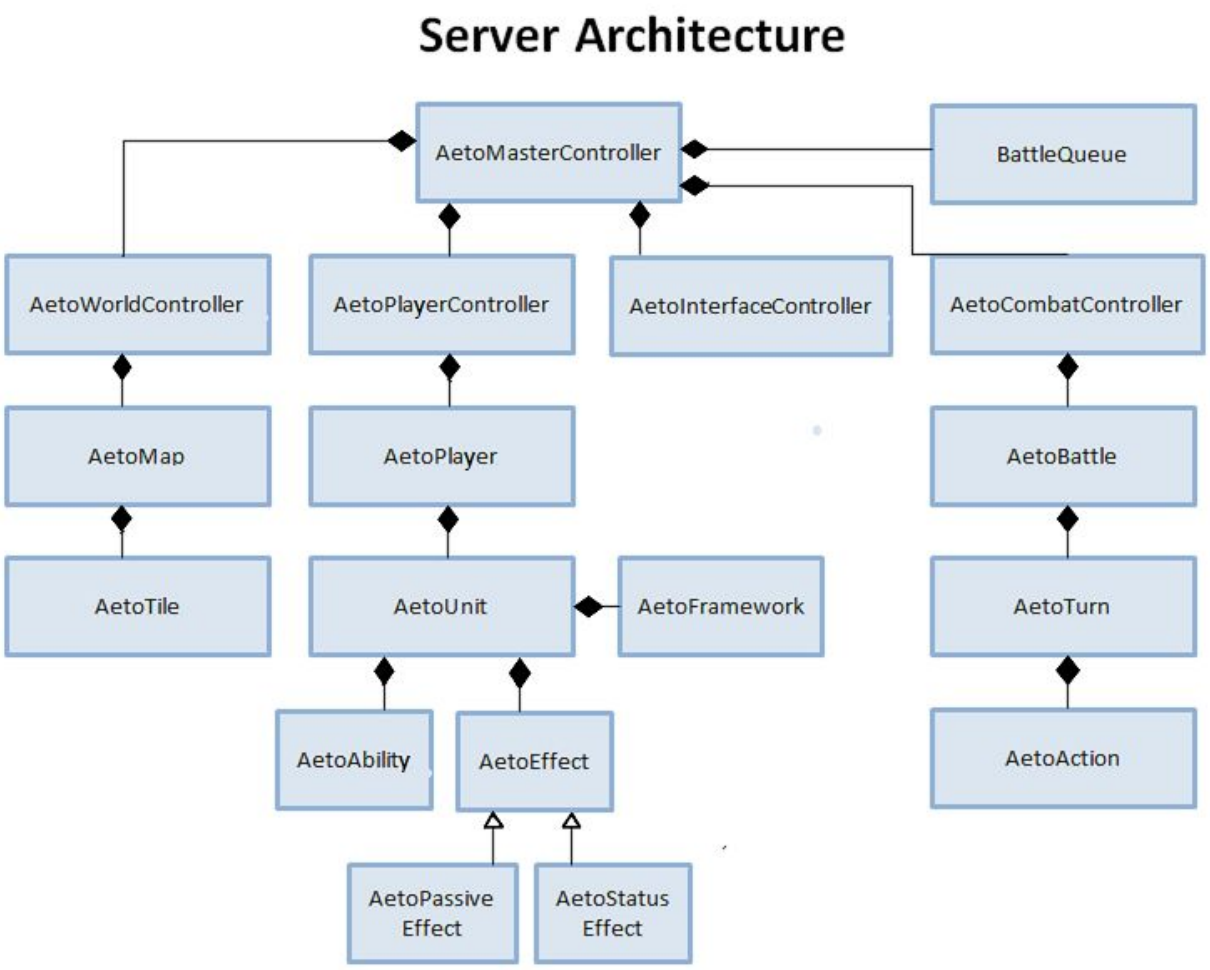
In 2014, almost two decades after HTML4, HTML5 was completed and introduced as a new iteration of the standardized HTML used across browsers. The revision introduced a massive number of new features that had shown significant potential, were very robust, and had several practical applications. In terms of relevancy to gaming, components such as WebSockets, Document Object Model (DOM) Scripting, new, platform-aware media support, the 2D Canvas, a File API, WebStorage, and Cross-document messaging are by no exaggeration, revolutionary. Although limitedly explored due to recency of its introduction, the robust, cross-platform feature set available in HTML5 has made the process of developing games, quick and highly efficient. The new markup language allows a game to be run on most any modern web browser, operating system, and hardware with far less effort than was necessary before. This was made possible due to HTML5 largely improving what Javascript was able to accomplish for a web page. It had essentially made the browser a full-featured application environment. As a result, new libraries that leverage HTML5’s new features have surfaced, a few of which are game development centric.

Just a couple years earlier, in 2012, Cascading Style Sheets 3 (CSS3) was completed and standardized, The improved stylesheet language introduced a number of new things that could be done to a browser’s DOM elements. Additionally, its change to module based stylesheets allowed for rapid feature development resulting in a number of CSS4 languages as well a consistent and constantly growing set of CSS3 features. The adaptability and effectiveness of the new format, which included useful items such as selectors and inheritance, had made modifying complex DOM element structures much easier. Variations such as Syntactically Awesome Stylesheets (SASS) which introduced scripting components as well as cross-platform standardizations of CSS4 languages allowed for web developers to have their pages and applications appear uniform across machines while leveraging many CSS3/CSS4 features more easily. With components such as transitions, animations, filter effects, masking, 3D transforms, and clipping, it became possible to create almost any kind of graphical interface or web page. These effects were then optimized by many modern browsers engines, such as Gecko® and Webkit®, improving load and render speeds by more fully utilizing a machine’s hardware. This was especially pertinent in animations and transitions.

One year before the release of CSS3, WebGL was introduced and became widely supported in most modern browsers. WebGL introduced 3D graphics rendering to the Javascript application program

interface (API), allowing browsers to draw 3D scenes with graphical process unit (GPU) acceleration without the need of any external software or plugin. Shortly after, a number of WebGL Javascript libraries were developed that had further simplified the process, making its usage much more popular and functional. WebGL operates very similarly to OpenGL, being its subset, and as a result, it has shader code that also prioritizes execution on a machine's GPU. This means it has many of the optimizations in OpenGL and operates similarly in performance. Although it has limitations due to operating within a browser, most modern browsers will translate WebGL calls into equivalent OpenGL calls, minimizing the performance difference. This means the 3D graphics and even optimized 2D graphics that are pivotal to most modern games are natively available in a browser and is directly accessible component of the Javascript language. This was the key step that allowed modern browsers to be taken seriously as a gaming platform. Additionally, as of 2013, WebGL 2 has begun development and actively seeks to bring high-end 3D graphics performance to the web browser.

With HTML5, CSS3, and WebGL working in unison through the Javascript API, the development of scalable, mobile, cross-platform, full-featured games are no longer a pipedream but a certainty. The modern browser is rapidly becoming a type of "universal operating system," and similarly, with operating systems in the past, games will always find success. As the web browser continues to grow in functionality and capability, its function as a gaming platform will grow alongside it.



Server

The server is a remote game-engine that manages player data through network interfaces and drives gameplay. It accepts incoming connections from players, and sets up a battle for each pair of queued up players. It keeps track of all active battles, using a lookup table to connect player requests to the battle they are involved in. Upon receiving a requested action, the server validates it and then runs the command, sending back responses with updated information to both players in the battle. It continues this cycle until the battle completes. The overarching master controller controls each of the four sub controllers, which oversee their own responsibilities of interface, world, player, and combat in order to run the server in a safe and effective way.

One of the first and foremost tasks when developing an online game is undoubtedly the interactions between client and server. Going into the project, we understood that all of the requests and responses sent between the client and server would be tailored to the game. Simple events such as moving, attacking, abilities, and other actions would require sending information in a structure matching our game's design. Since we were developing in a Javascript environment and this data would have to be

customized for our implementation, we decided to use the Node.JS library for server communication. Node.JS is a light-weight event driven model and its feature set fit well for the game's design. With more research, we found that the Socket.IO and standard util modules that were commonly used to support Node.JS for creating bidirectional communication interfaces. Socket.IO greatly simplifies the communications between client and server. It abstracts away a lot of the difficulties involved in network communication and allows us to send information attached to an event in a single line.

Another goal we had when implementing our server was persistent data. To do this, we used a lightweight scalable form of NoSQL called MongoDB. Node.JS supports MongoDB through the use of the library mongodb. There is also a higher level library called mongoose that allows for even easier interactions with a MongoDB database. However, this fell outside the scope of this project. If this were to be continued in the future, Mongodb and mongoose are plugins that would easily help the integration of persistent data.

Our final concern is security. Originally, the project was supposed to include a login system to interact with the database. Like with many online games, there is a possibility of users stealing or accessing other people's accounts. To avoid this, we would encrypt the database with Bcrypt, NaCl or another hashing algorithm that would help secure user information. However, this fell outside the scope of our project. If this project is continued at any point, the developers must undoubtedly use some form of encryption to secure the server.

Sockets

Once we knew what frameworks to use, we set up a simple client connection interface that would listen to any connections on a specific port. Socket.IO provides a very simple setup function that allows you to run a server and listen to incoming connections. This was useful because we could set the port to read from the local host for testing or from a dedicated server when it became ready for use. After the server was assigned to a port and listening for events, we added an event that would listen for client connections. This initial event sets up the event handlers for every other event the client can send to the server. This includes creating a player, adding units to a map, selecting units, queueing for and starting battles, battle actions, disconnecting, and more. The last action required for setting up the server was creating a simple data structure to store the client ID's in. We created a simple array for this, and upon each new client connection, we created a parallel player to go along with it. Each player would encompass their own socket ID, allowing us to relate each incoming socket event with a player and vice versa.

Players

Players are represented by the AetoPlayer class, one of the simpler classes in the AETO structure. Each player has a client ID, username, list of units, and color. We created a loosely structured pseudo-database for the players and a few units with some values for playtesting. This was eventually supposed to be transferred over to a MongoDB database that coincided with player logins, but fell outside

the scope of the project. The most important part of each player was the list of units he or she was able to command, their correlation to client ID's along with their representation in a battle.

Units

An AetoUnit consists of all the information a unit is comprised of, including base stats and the player it belongs to.

There are two types of stats. The first type, base stats, comes from the units' class and level. These stats are private variables, inaccessible by the player, and should only be changed from events such as unit framework changes or leveling up. The second type is made up of numbers that are constantly changing throughout the battle, and are only accessible through an accessor function, "getAttribute ". These stats are initialized at the beginning of a battle, and come from the unit's private base stats after modification from equipment. By having these two sets of attributes, we are able to prevent the unit from losing its original stats. For example, if a unit were to be charmed, allowing it to be controlled by the opponent instead, we would not want the unit to forget whom it actually belongs to after the fight.

A unit can also have status effects applied to it throughout the game, including status effects that modify a unit's stats. For this to be executed properly, the unit has to keep a list of status effects applied to it, as well as keep track of what the effect does to the unit. But again, since we do not want the unit to forget its original stat values prior to the battle, each call to the "getAttribute" function will look through all the effects modifiers and apply them before returning the value of the attribute. This allows adding and removing effects to be easily implemented since they don't actually mutate the battle stat fields themselves.

Effects

An AetoEffect is broken up into two types of effects. The first type is a PassiveEffect, a passive ability that is innate to the unit's class. The second type is a StatusEffect, an effect that is applied to a unit due to an ability. The main difference between these two is that a passive effect is permanent and not removable whereas a status effect can be added and removed at any point in the battle.

To simplify the design, both of these fall under AetoEffect and are generalized to operate the same way. By doing so, it forces a uniform implementation of all the different effects, making effects more easily incorporated into the rest of the design. In Java terminology, the idea is that all effects must "implement" an AetoEffect and all its required functionalities.

Specifically, an AetoEffect requires a function that returns a list of modifiers that could affect an attribute, even if the effect does not have any. It also requires functions for the following callback events: 'start turn,' 'end turn,' 'attack,' 'attacked,' 'heal,' and 'healed.' This allows for a multitude of abilities to come into existence. For example, lifesteal, where dealing damage to another unit causes the attacking unit to regain health, can be triggered during an 'attack' event. Conversely, a unit that has an effect to return damage dealt to it back to the attacking unit could be triggered during an 'attacked' event.

Map

The map represents the game field that the players will place their units on and where the battle will take place. We obtain the map data from a JavaScript Object Notation (JSON) file that is either made by our map editor or by other means that can produce the data with the same formatting. We figured that it would be best to create an editor to construct our maps rather than just hard-coding them so we could create maps more quickly while being able to see them as we build them. This map editor has been a handy feature that we have used quite a bit to test different maps and how abilities, movements, and attacking had worked on them.

After we receive the JSON file, we parse that into a 3D hashmap of x, y, and z. This hashmap contains the x, y, and z positions and a data object that contains the tile type and what is on top of it. We figured this would be the best approach due to the constant retrieval of data from the map and we wanted the method to run in O(1) time. It is on average O(1) time to add an object to the map, and is only run at O(n) time when the battle is initiated. We used a 3D hashmap as opposed to a 3D array for the performance boost since we typically do not have to worry about the y value since it is different depending on character height. Because of these two things, we can generally ignore the y index, only keeping a reference of the height. Hashmaps in Javascript are very handy and contain a lot of features that you can do with them if needed.

Our hashmap is keyed by x and z location and contains AetoTile objects. An AetoTile object contains the position, type, top, and other important data for the battle. One important aspect is that it allows us to nicely apply spell effects to certain tiles.

Inside the AetoMap file we also have methods that allow location verification for movement, attacking, and abilities. These are mostly generalized utility functions used by an action to obtain a list of valid tiles when moving, attacking, or using an ability. The different kinds of verification algorithms we have include obtaining the tiles for a circle, a square, and straight lines. These are the most basic shapes that our game uses and everything more complex stems from them.

On the server side, these algorithms are used primarily for verification. On the client side, there are near identical algorithms. The reason we have it in two places is because the client's CPU can perform this function faster and the player can request the information from the algorithm indefinitely. If the results from the algorithm were always requested through the server, the server could suffer performance and timing issues since some of the algorithms can be very time consuming. The verification process on the server only happens when a player confirms that he wants to do something on a tile, and then the server checks if the tile is a valid place for that action to happen.

The AetoWorldController is where the AetoMap is constructed and initiated. The controller's functions are called from AetoCombatController mostly to setup the battle for that specific battle between two players. The AetoMap is passed along as a shallow copy to almost every object that deals with the battle. Since it is a shallow copy, it is being updated on all scopes which is very useful because our game does not need synchronous calls.

Tiles

An AetoTile is the basic building block of a map. It knows its x, y, and z locations in the scene, and whether or not it has a unit on top of it. It also holds onto a list of "status causing effects", for lack of a better name. The reason for this feature is that some abilities are able to leave effects on the field that can heal or damage units that end their turn on an affected tile. Units that end their turn on these tiles could also have a status effect applied to them for a certain number of turns. For this reason, a tile needs to know what status effect to apply, the number of turns it lasts, and the amount of damage or healing to apply to a unit, if any.

Battles

In the scope of this game, a battle is an environment where two players can interact with each other by commanding their units and taking actions. Because this game was designed as a MMO (Massively Multiplayer Online) game, there needed to be a scope that would manage multiple battles and tie players to them. To do this, we created a Combat Controller. This controller queues up players when they are ready to battle, sets up a basic representation of a battle, assigns two players to a battle, receives actions from players, and processes meaningful data. The combat controller has a running list of active battles and ties socket driven events with those battles. When a battle is initialized, the combat controller creates event handlers to receive information from the battle, process data changes, and send the corresponding changes along. These changes will eventually be sent back to the client in order to change the visual representation of the battle.

Each battle is a large data representation of everything tied to the actual battle. It has players, units assigned to those players, turns, rules, and a map. Once the battle is initialized, it creates an event driven environment where actions create events that are responded to or followed by other actions. Upon battle start, the battle signals an event for players to place their units. Once the placing phase is finished, the battle initializes the first 'turn' that begins the cycle of events. As soon as turns are initialized, the battle constructs callback functions for turn events. This allows the battle to process actions from turns, manipulate data upon events, and send information to the combat controller. The structure of this game is that battles are made up of multiple turns and turns are made up of multiple actions. Both actions and turns are driven by user events and run continuously until the battle is 'over', meaning one player has beaten the other. This happens, as you would suspect, when all the units belonging to one player on the map have been killed.

Turns

As mentioned above, there are multiple turns that comprise a battle. Each turn has a unit that gets to act, a list of units that are part of the battle, a copy of the map, an allotted amount of time that the unit is able to act during, and a list of actions. As soon as a turn starts, it signals the battle that it has started and the battle passes the relevant information to the client side for visual updates. When the user decides to act during their turn, the combat controller passes the desired action to the battle which sends the information over to the current turn. The turn then constructs the specific action and adds it to turn's

action list. As soon as the turn constructs an action, it sets up the event handlers for specific actions that can occur such as move, attack, or abilities. If the turn's action list indicates that the turn is over or the allotted time runs out, the turn sends an 'end turn' event to the battle. The battle then completes any necessary actions such as updating the battle data and begins a new turn. A turn is over if time runs out, if an unit performs a combination of actions that ends their turn, or as soon as the player decides to wait.

Actions

An AetoAction is an object that represents one command a user may take. It is either an active action, a movement action, or an end-turn action. Values are then assigned to each different type of action and those values determine how much of a turn the action takes up. Actions are made up of an execution function, verification function, unit, map, target location, and action name. By using the action name, we are able to create a database of specific actions that can be taken. Each action has its own verification and execution functions. We use the unit, map, and target location to determine specific effects and values that the action will change, the validity of the move, and the affected units. The verification function of the action determines whether the desired action will be added to the turn, or ignored as an impossible action or potential security hack. Actions use a copy of the map specific to the turn they are part of in order to create local changes to the map. If the turn ends correctly, the changes made to the map and units will be applied to the original copies in the battle. If the turn is unsuccessful or a player wishes to undo the commands they have taken before ending the turn, the changes made can be undone because of the turn's local information.

Once enough actions have been added to a turn and the turn ends, the execution function of all actions that are part of the turn will be run. This creates an object holding information of what the action will change and uses a callback to signal the turn that the action has been completed. The turn verifies the callback and uses its own callback to signal the battle of the changes. The battle then creates the desired changes, and creates an event to be passed back to both clients that are part of the battle. This event signals the client of the changes that occurred and allows it to update correspondingly. Once the turn is over, another event is fired to end the turn and begin the next unit's turn. Again, this information is sent to the client in order for the turn animations to complete and the new turn animation to begin.

Ability

An ability is a specific type of action, allowing it to easily fit into the rest of the system. Currently, all abilities are referenced from a static library, an implementation that needs to be changed. One reason is that abilities need to be able to be instantiated, since abilities might have cooldowns tied to a unit, so they shouldn't be statically referenced. Abilities also should implement an AetoAbility interface, allowing for a uniform implementation of all abilities.

Specifically, an ability is required to know which class it is associated with and its EP (Energy Point) cost, EP is one type of resource that a unit has, and is consumed upon using an ability. Then, along with the information provided to actions, an ability can calculate the rest of the information it needs. This includes being able to determine the casting range of the ability, to determine the area of effect of the ability, and to obtain the list of units affected by the ability. After calculating the information related to an ability, the ability can then run its execution function just like any other action.

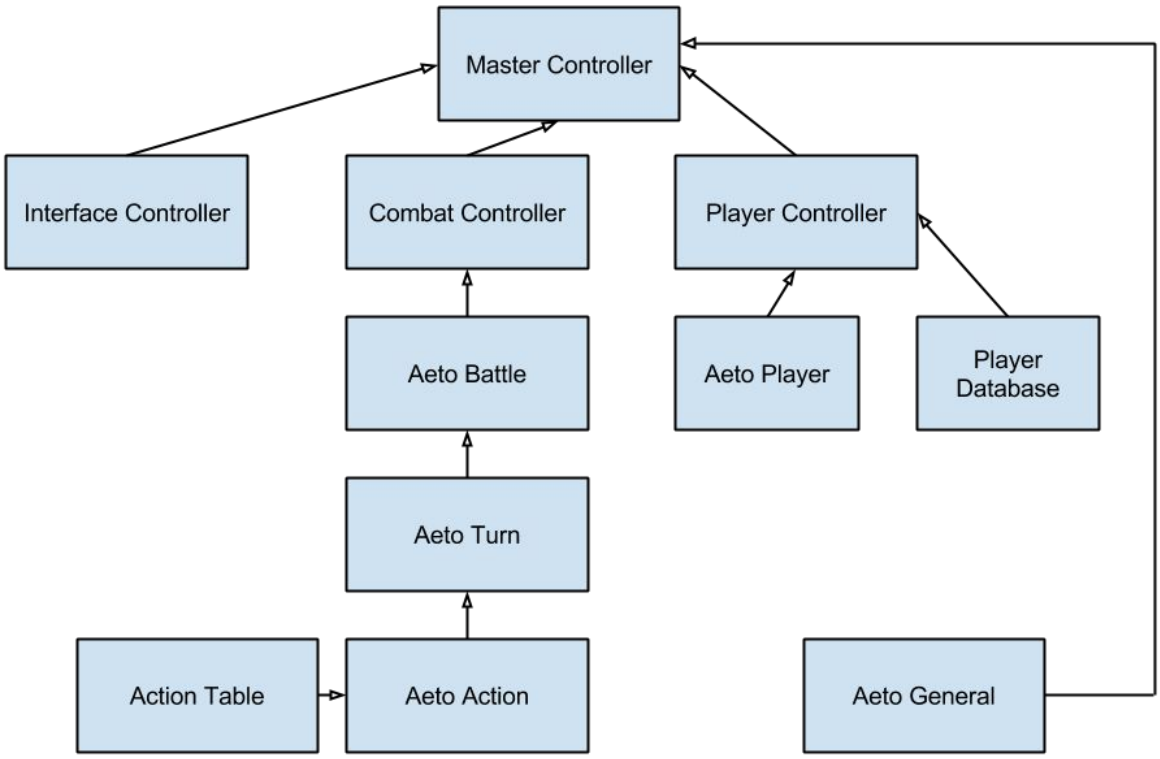
An ability has two private variables to help determine its casting range. The first one is a number, specifying the range of the ability. The second variable specifies the type of limiting algorithm. For example, some abilities can be casted anywhere in a circle surrounding the caster, as long as it is within range. Other abilities can only be casted if it is in a direct line away from the caster, meaning that the target tile and user both share a common x-axis or z-axis.

Similarly, an ability has two other private variables to determine its area of effect. The first one is also a number to determine the radius of the effect. The second variable helps determine the algorithm for calculating the area of effect, since the area of effect could be a circle, square, or something entirely different.

The main reason behind abstracting away the algorithms from the abilities is to allow the abilities to contain only the bare essentials it needs to function as an ability, without over extending its scope. By asking the map and specifying an algorithm, the map is able to provide the ability with the information that it needs.

Interactions

In an above section, we explained how the server accepts requests from clients. Those requests are boiled down to specific actions as part of a battle and create events. But how do these events flow through the hierarchy and pass back to the correct clients? As briefly mentioned earlier, we heavily rely on a callback system to create these interactions between client and server. Although specific actions differ in the scopes they pass through, the general idea is that actions occur on the smallest scope. When these actions occur, they traverse from their own scope to the battle, and from the battle to the specific controller in charge of whatever action had been requested. These controllers, similar to the combat controller, oversee multiple instances of the game. The controller then uses the master controller to connect with other controllers. The master controller is a sort of parent scope that exists throughout the server and connects all different controllers together. Using this parent scope, the controllers can interact with an interface controller that socket emits the changed values over to the affected clients.



In summary, a child action creates a chain of callbacks to send data to a controller. The controller uses the parent scope in order to send information to the interface which finally signals the clients that changes have been made. Besides a couple of helper objects and functions that exist throughout the program, each action or object is only aware of its own scope. These scopes pass data into higher scopes through callbacks and interact with each other through a system of controllers. By doing this, we created a very safe way for data to be passed throughout the server for multiple battles without a chance of corruption. This model took many attempts before looking the way it does, but it serves as a base for a much larger project. Although the functionality of our project could have been finished without this controller based model, any reasonable MMO will have to implement a similar system to keep the massive amount of data and objects safe from each other.

Client

The client is the local game engine that directly interacts with the player. It has two primary purposes: the first being the rendering of a corresponding graphical interface and scene for the data passed from the server and the second being the processing of user input and passing it to the server. The client is represented as a web application on a web page and uses a number of Javascript libraries alongside its core engine in order to achieve this behavior.

For 3D graphic rendering, we had decided to use Three.js, a simple, light-weight WebGL library. We decided to use Three.js as opposed to other WebGL libraries such as the newer, more structure Babylon.js since Three.js had been around longer, is better documented, and has a few more resources. Although Babylon.js might have made development easier in a few areas due to being designed for gaming, Three.js had proved to be sufficient.

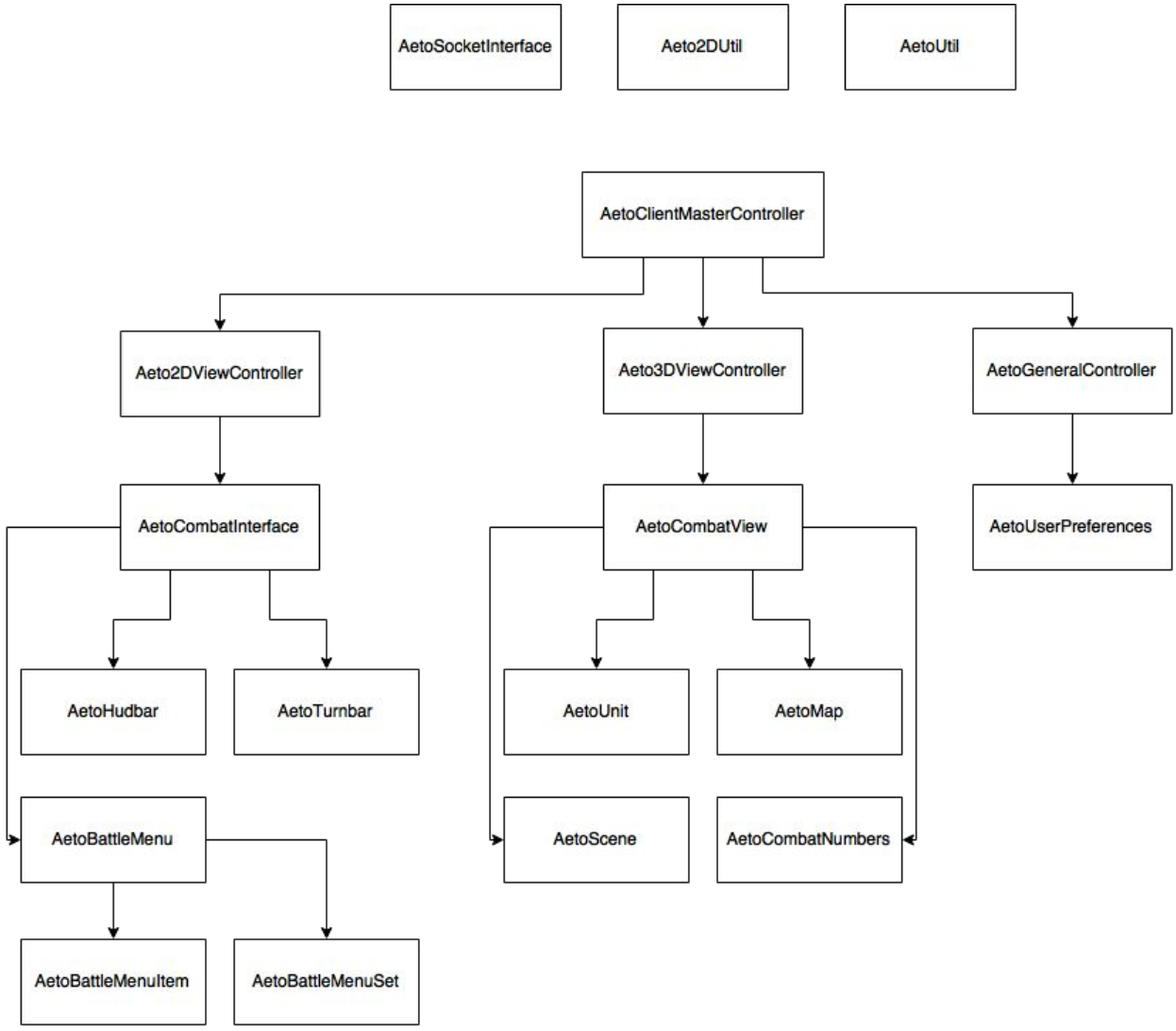
For 2D graphics, where were primarily used for the user interface, we had to make a key decision between two choices, to either leverage the HTML 5 2D Canvas to render the graphics in a traditional graphics-programming approach or to use HTML5 DOM elements and gain DOM functionality at the tradeoff of versatility. We ultimately decided to work with DOM elements since it was much more friendly for a structurally complex interface and was more consistent across platforms and resolutions, Additionally, all browsers focus on DOM elements in their development tools, and as a result, working with DOMs has a much faster development cycle than working with the canvas. While the performance gap between the 2D canvas and DOM elements might have been wide in the past, new optimizations in DOM animation libraries and improvements in modern browser engines has made it much thinner. The framework we used for DOM manipulation was Alloy User Interface (AUI®) due to its robustness and module based structure, allowing it to become lightweight through selective loading.

For 2D animations, we had used CSS animations originally, but after discovering weaknesses, we had switched to the Greensock® Animation Platform (GSAP®) for both 2D and 3D animations. Out of all other animation libraries we had looked into, GSAP was top in performance, robustness, and support; it had even outperformed CSS animations in several cases. As a result, we used it to interpolate all the attributes we use for animation. Beforehand, we had also used Velocity.js, an animation library that functions similarly, but midway through the project it was also dropped for GSAP. Although adapting old animations to the GSAP structure took a significant while, it significantly improved performance.

For managing WebSockets, we had used Socket.IO, which had greatly simplified the usage of WebSockets and made it very convenient through a simple event callback system. Connecting to the server was also very simple due to this library. Although we had considered supplementary libraries such as Express.js and Pomelo.js for additional features, at the time of development, the project was simple enough that this was not necessary.

Structure

For non-code resources such as CSS, HTML, scalable vector graphics (SVGs), fonts, and textures, they are organized into folders and loaded upon need through AJAX calls available in the AUI and Three.js libraries. The structure of our client code is a layered model-view-controller (MVC) architecture with additional global utilities and libraries. It's broken into several components, diagrammed below.



Most classes in this structure are then loaded sequentially and made available to one another on demand through the Require.js framework. Since each class represents a discrete module, Require.js acts as a module loader that speeds up load times.

Each controller class also has inherent callback functionality allowing it to retain the callback model relationship. This functionality was generalized and abstracted into the AetoUtil class and attached to each controller class's constructor.

Sockets

To manage WebSockets, we create a general utility class called AetoSocketInterface that acts as a wrapper function for the Socket.IO library. This allows us to retain all WebSocket calls in one closure, allowing for easier maintenance and network tracking. Unlike other classes, AetoSocketInterface is a global utility object, and as a result, it is not loaded as a module by Require.js but is made directly available on page load.

Upon load, an instance of the AetoSocketInterface class is instantiated and made available to local classes. Upon initialization, it will attempt to open a connection to server, allowing other controllers to use its accessor functions to request data from and send data to the server. These accessor functions utilize the Socket.IO 'on' and 'emit' calls to communicate with the server. Other controllers can then leverage Socket.IO's callback system in similar practice to the server.

Upon connecting with the server, other controllers may directly access the objects 'onResponse' and 'send' functions in order to receive and emit selected WebSocket data. The structure of these requests and emits are exactly similar to that of the server's. The function 'send' takes an event name and a data object to pass to the server, and the function 'onResponse' take an event name and the function to run that takes in the received data as arguments.

Map

In order to render a game map, we created a client version of the AetoMap class. It comprises primarily of an associative array or hashmap keyed by the tile locations and holding the actual Three.js mesh objects representing the map blocks. This implementation is very similar to that of the server's AetoMap class and for the same reason, the hashmap was the preferred data structure due to performance.

An AetoMap object is generated by taking in a set of coordinates in 3D space, passed in from the server as a data object. Upon initialization, it iterates through this data and creates all corresponding meshes for each coordinate, texturing it based off additional information passed in the data object. The texturing process uses the Three.js MeshPhongMaterial which allow the application of specular and bump maps in addition providing phong shading. This allows the cubes textures to appear to have depth and ridges as well as reflect light in the environment,

Lines for drawing the grid overlay are also generated alongside this process using the Three.js EdgesHelper function that identifies edges within a mesh's vertices. On top of each map cube, a flat surface mesh is also generated. This mesh is primarily not visible but is used for raycasting, allowing the player to click on blocks select them. These surface meshes are also made available through an accessor function, allowing them to be colored to illustrate tile ranges and areas on the map, resulting in a tile highlighting effect.

Our original intent was to not have the cubes textured at all, but to only provide the grid overlay, tile raycasting, and tile highlighting. These effects would be laid on top of a more appealing 3D

environment. Unfortunately, since we have had insufficient time, we were unable to implement this functionality.

Units

For units, we wanted them to be represented as 2D sprites with multiple sprite angles, similar to many early RPGs. This allows them to visually adjust to the 3D environment by switching direction based on camera angle.. Below is an example of this sprite technique, using a set of sprites to illustrate a character at multiple angles.

Sprites courtesy of Ragnarok Online®

To do this, we created the AetoUnit class to represent a unit. It is an extension of the Three.js Object3D and has multiple children meshes that comprise the object. They had originally started out as SphereGeometries during early development and were then replaced by Three.js Sprites to achieve the effect. However, due to a number of issues such as 3D rotations, rendering priority, and completely ineffectual raycasting, we had switched to specialized Three.js plane geometries with sprites textured on top. To mimic the effect of the Three.js sprite that would always face the camera, AetoSprite would mirror the camera's current quaternion on update. To minimize performance loss, this would only occur whenever the camera had moved.

To determine which directioned sprite to display based off the camera's angle, we calculated the azimuth angle to the center of the scene from the camera,, to achieve an orthogonally accurate angle. This angle is then thresholded across a number of rotational parameters to best determine which way this unit is facing. Similar the facing-determination function,, for performance's sake, it will not update unless the camera has moved. This angling also keeps track of sprite mirroring, since some directional sprites are reused. For example, a sprite facing right would simply be a reversed copy of a sprite facing right.

Lastly, we wanted our sprites to slightly deform upon viewing it from upward angles. This is so the sprites appearances would have a depth-like effect, making them appear shorter upon higher camera angles. This was done by thresholding the camera's elevation angle to the center of the scene and then adjusting the angle by subtracting a modified version of the camera's minimal positive elevation angle to the center of the map. The sprites are then rotated on the x-axis based off this angle.

For sprite animations, sprites retain a state such as standing or moving. States are also directionally based and are updated in the sprite's 'update' function as well. Animations occur by cycling through the textures of each state at a set interval. This interval are kept track of by a repeating set of GSAP-provided delayed called. Textures are changed by replacing the entire plane geometry since this is the only way to prevent vertex deformation from sprite aspect changes.

To ensure that sprites always render on top of terrain despite being position below it, sprites are kept in a separate Three.js scene, acting as separate rendering layer. Originally, turning off render sorting for the sprite was sufficient, but when effects were added to the sprites, it was no longer possible to layer the effects properly. Sprites always need to be explicitly rendered on top of terrain when above it because the base of the sprites are not at the bottom of the texture. Since the center point of a sprite is generally above a more forward placed foot, sprites need to be positioned lower than the terrain to appear on it.

Our sprites also have effects such as shadows and team indicators. These were accomplished by adding them to the AetoUnits children, alongside the unit mesh. There are added in a set order to achieve expected rendering priority. Since the sprite resources are loaded asynchronously, none of the children will be generated until the required textures have been loaded via the Three.js texture loading AJAX functions. This allows the unit to appear incrementally instead of missing a component or two upon load.

Scene

Over the course of development, we had learned that the original Three.js Scene class was insufficient for meeting our needs. As a result, we had extended the class, creating the AetoScene subclass. It includes features such as keeping track and hashing all added objects and their appropriate raycasting handlers and multiple subscenes to simulate rendering layers. Objects can then be raycasted easily and generalized in the scene's utility functions. Removal of raycasting handlers was also significantly simplified through this process. Raycasting will also keep track of each Object3D's children, allowing them to fall under their parent's raycasting handler.

Orbit Controls

For camera controls, we had originally used Three.js OrbitControls class which had provided zoom, rotate, and panning functionality for multiple devices for an orthogonal camera. However, these controls were jagged and the controls were limiting. As a result, it had been significantly altered. Panning now has velocity values for smooth entry and exit. Zooming had to be more rigid and accurate than panning and, as a result, instead interpolates the zoom delta supplied by the orbit controls and updates it over time instead of per scroll tick. Rotation required smooth exit, allowing a drag-throw effect, but is expected to halt on grab, The was accomplished by summing the phi and theta changes over time and tracking their rate of growth and decrementing these summed delta and phi values over time. These modifications in orbit controls required significant changes in the algorithm.

Combat View

To represent the aggregation of the 3D elements that comprise a battle, we create the AetoCombatView class. It instantiates AetoUnits, AetoCamera, AetoScenes, and variety of other objects and operates each one of them and sets up their required relationships. An AetoCombatView is generated upon combat start, as indicated by the server response. It is instantiated by its parent controller, the Aeto3DController. It sets up a variety of event handlers for various combat events such as attacking, unit placement, and other unit actions. It then reacts to these actions appropriately and passes relevant information to its children. As of now, many components in the AetoCombatView could be abstracted into separate modules. However, due to time limitations, it is a temporarily a view controller that wears a lot of hats and manages several child objects.

HUD Bar

In order to represent unit data such as health points, energy points, and turn points along with other stats and identification information, we wanted to create a convenient multi-bar interface item that anchors to a corner of the screen. As result we created the AetoHudbar class, simplified to “hudbar”, which represents a UI element that meets this need. Upon initialization, it makes an AJAX request via the AUI IO interface to get the necessary DOM structure that represents the element. With this data, breaks up the DOM structure into several components, attaching handlers and modifying data where necessary. This entire structure is then placed in the predesignated UI DOM in the webpage.

The hudbar has a wide variety of futures including complex and simple entry. Complex entry is comprised of 190 discrete animations and takes longer. Simple entry is only comprised of 4 animations which happen very quickly. The complex entry was initially intended to be used upon game initialization. However, due to time constraints, it was not formally implemented. The hudbar is programmed to be unnecessarily robust and can handle any number of cycling components for each point bar and can be patterned into a variety of different DOM segments as necessary. For example, although the HP bar is comprised of a repeating pair of blue DOM segments, if the markup was instantiated with a triplet or five-set of varyingly styled DOM segments, no issues would arise since no magic numbers are used. All it would take to modify the hudbar’s components would to be directly modify the HTML.

The hudbar also requires zero images and instead uses a variety of masking, clipping, and filtering techniques to achieve its appearance. The use of SVGs for the masking process allows it to be infinitely scalable at any resolution, reshapable, and load much faster. This also allows each individual component on the hudbar to be separately modified and animated since there are no images that hard set a component's appearance. This is utilized to recolor the hudbar based on team color and animate the portrait of selected units with active turns.

The hudbar is animated using GSAP’s CSS plugin and is designed to primarily use acceleratable transforms to ensure good performance. Combined with AUI, GSAP was also used to animate all text values such as a unit’s HP over time or ATK over time. Although sparsely used, this allows stat modification effects to be recognized visually. The hudbar also supports status effects, representing them as fillable trapezoids in the bottom row. These trapezoids will automatically move and flip as effects are added and removed to retain a uniform appearance. The hudbar will also resize based off the screen size

and the length of the user's name. The number of status effects applied to the unit will also cause the bottom row of the hudbar to extend.

The point bars in the hudbar provided a rather difficult challenge. When using CSS animations, it required a complex queue system, but has been simplified by the GSAP's tween killing functions which allowed queued animations to be cancelled or removed on the fly. Since point segments could enter and exit at the same time and rapidly change if a unit is healed and damaged at the same time or several times in a short period of time, all segment positional properties had to be cached, and each segment had to be re-instantiated upon need. The complexity of this problem was furthered when these segments would constantly cycle from left to right as part of the hudbar's idle animation. Additionally, since SVG masks themselves could not be readily resized due to their complexity, a number of SVG masks had to be laid on top of each other to achieve the point bar slicing effects when a unit's HP, EP, or TP is altered. The positions of these layered masks then had to be consistently kept relevant to each other and updated in uniformity.

The hudbar also has a number of other features. It will naturally scale based on the window size, within certain thresholds, while retaining its anchored position. This was done through simple transforms and transform-origin modification on resize handlers. The hudbar is also has stat value modification functions, allowing the introduction of new animated stat values with automatic position updating. The rounded positions of these text values were accomplished using the SVGs recently implemented text path component. Although there were Javascript alternatives for completing the same activity, the SVG textpath was more robust and efficient despite its increased complexity.

Turnbar

In order for the player to keep track of the unit turn order visually, we had to create a UI element that would allow the user to view a turn order timeline while taking up minimal screen space. The result was the AetoTurnbar class. Similarly to the hudbar, it requests HTML upon initialization and splits up the DOM structure, stores repeatable DOMs, and populates a complete DOM representation before inserting it into the predesignated UI parent DOM. The turnbar is comprised of three major components: the unit list, the scroll handle, and the turn timer.

The unit list is a sprawling object list that represents the turn order. It is comprised of a large set of cached DOMs in sequence, allowing the player to rapidly cycle through them with minimal to no hang time. These DOMs are generated upon creation but will also autogenerate upon increased logging demand. Each item is comprised of several child DOMs, and similarly to the hudbar, uses no images for rapid loading and scalability. Its components include a unit name, the unit's class, represented by an icon, a turn order value, and the unit's portrait. Its color will adapt to the team color of the unit. These unit detail DOMs also have their own animation set and will play upon turnbar entry. This was completed in GSAP, and through queueable and interruptible animation timelines, the turnbar can be used before entry animations for these elements even complete. Cycling through these DOMs also have their own animation, which will adjust for performance depending on cycling speed. The number of displayed DOM elements defaults to 10 for large screen resolutions. However, the size of each item as well as the number of items displayed in the list will vary for smaller resolutions. At minimal size, only 4 DOM elements will be displayed. This is accomplished via a two-step process, the first being the attaching of a

module that thresholds the window width in accordance with margins and padding to a resize handler. The thresholding determines the number of items to display. The second step involves the readapting of the size of all unit detail DOM elements to fill the new size based off the number of displayed items, this includes cached elements as well.

The scroll handle is a complex DOM object that houses a variety of handlers to simulated expected scroll behavior. It does this by first measuring screen size and adjusting it based on the loaded size of the adjacent and child DOM elements. This size is then used to map locational values to the number of DOM elements in the unit list. This functionality automatically repeats upon window resize, allowing it to scale and function on a variety of browsers without issue. These handlers were made possible by utilizing the AUI event module. Although AUI had provided its own scrollbar system, its flexibility and aesthetic potential was insufficient. The scroll handle can be moved in three ways. The first is by using the scroll wheel when the turnbar is hovered. Hovering will also extend the turnbar, making this behavior more apparent to the player. Each scroll tick will then result in moving forward or backward one unit detail item in the unit list. The second way is by dragging. When the scroll handle is dragged, it will animate, transforming to illustrate is a draggable item. When dragged, the handle will rotate child elements to create a neat visual effect. The dragging process will still have the handle “click” to certain points that directly represent a position in the unit turn list, but since the movement is animated, the appearance is still smooth. The dragging process also continues until the user lets go of the mouse or until the mouse has moved out of the screen. At the same time, the mouse icon will change to indicate this behavior. The third way to move the scroll handle is by directly clicking on a location in the scroll bar area. This will instantly jump to a location in the unit list. Since this is a rapid shift in unit list items, the animation of the unit list to that specific point has been sped up and simplified, resulting in no performance loss. The availability of this functionality is indicated by an animated flag that shows up at the mouse position on scroll area hover.

The third component of the AetoTurnbar is the timer system. Although it is not directly relevant to displaying order, it does indicate how long a player has available to take a turn. It is comprised of both a clock, which cycles through a 12 hour period to represent the entirety of a turn and an animated digit-based timer. This timer is kept synchronous via the GSAP delayed call function, allowing it match system time fairly accurately and avoid issues present in the native Javascript ‘timeout’ function. This is especially pertinent since the alternative of using GSAP timelines would cause the timer to desynch upon switching tabs in order to minimize resource costs of background processes. This timer also has its own complex entry and exit animation that are set upon a timeline.

BattleMenu

Since the gameplay is that of a TRPG, players would need to be able to choose between multiple actions rapidly in order to complete their turn. This need was met by the AetoBattleMenu system which presents a menu that is accessible through, mouse, and keyboard that is instantly available with no input delay or input hangs. This menu is presented each turn and supports submenus, screen phases, and menu item updating on the fly, allowing it to always only show available actions at the time. It also supports resizing and implements a rather innovative and original menu design. Due to being massively more

complicated than any other UI element, I will not explain how it specifically functions or its other more complex features.

Combat Interface

The combat interface manages all the 2D elements that are necessary for a player to complete a battle. Upon initialization, it will setup all DOM element structures and UI elements that require early rendering for performance and attach relevant handlers so they may properly receive and send data to the server. Since these handler are managed by the combat interface, it also translates these user inputs into readable events for the server. This is applies for translating server information into parsable data for the UI elements.

The combat interface also manages a significant number of settings and values necessary to displaying appropriate UI elements. It also manages the caching and cycling of UI elements for improved performance. This is especially applicable to the hudbar where data rapidly changes upon unit selection and a player may cycle through units very quickly. Since each hudbar has an entry and exit animation, multiple hudbars are needed since the information cannot just be replaced to represent a new unit. As a result, a set number of hudbars are cached and cycled through, with their data being updated upon need.

Future Work

We had a lot of features that we wanted to implement that were initially on our first proposal, but after the first quarter, we had realized we would be unable to accomplish all of them. We had predicted this and fully expected it to happen. Meeting the requirements of our first proposal would have required us to work eight hours a day on the project like a full time job. As a result, our second proposal was much more simplified and realistic and we were able to meet every milestone on it. Our feature list that we wanted to implement is as follows:

- Blender®/Maya3D® for map
- Different map variations that would be playable
- All the framework's abilities/spells - only implemented three frameworks
- Items, Inventories, Equipment
- Phase/Confirmation system for doing an action
- Unit art/animations/pathfinding
- Ability animations and projectiles
- Database usage
- Social system

Some other features that we wanted to add in the project's future are:

- An Overworld
    - Quests
    - RPG elements
- Singleplayer mode

The first feature list was on our initial proposal, and the second list are ideas and thoughts that we wanted to add to the game in time. Some of these features can be implemented more easily since most of the structure is already set up for them. However, due to time constraints, we had put them on hold to delve into different areas. For some of these features, the code is already available, but they had not made the complete implementation due the team having to prioritize other features to get a complete battle working. Our hope is to continue improving our battle system and expand the game world to be more expansive and include an overworld.

Conclusion

Even though this was a senior project, we want to continue development a little more to see where it can go. This type of game has not been published before, and similar types of multiplayer strategy games have become more popular in recent years. The game concept was to have an outside world with RPG elements like trading, quests, crafting, and professions where players could walk around. But so far, we have only designed the bare essentials for the tactics or battle portion of the game. Therefore, future steps will include perfecting the battle system and creating the outside world. The ultimate goal is to create a complete demo of the game, allowing us to get a Kickstarter going to fund development and server costs, allowing it to become a full-fledged game.

Sources

http://threejs.org/
http://fullscream.com/what-is-webgl-and-how-will-it-change-web-design/
http://www.w3.org/TR/html5-diff/
http://www.w3.org/TR/2011/WD-html5-diff-20110405/
http://www.w3.org/2009/06/xhtml-faq.html
http://trac.webkit.org/wiki
https://developer.mozilla.org/en-US/docs/Mozilla/Gecko
http://www.htmlgoodies.com/beyond/article.php/3893911/Web-based-Mobile-Apps-of-the-Future-Using-HTML-5-CSS-and-JavaScript.htm
http://www.w3.org/TR/CSS2/propidx.html
http://media.wiley.com/product_data/excerpt/88/07645790/0764579088.pdf
https://developer.mozilla.org/en-US/docs/Web/JavaScript/New_in_JavaScript/1.8.5
http://www.ecmascript.org/es4/spec/overview.pdf
http://www.brenkoweb.com/articles/world-wide-web/web-design-software/history-of-javascript.php
https://www.youtube.com/watch?v=pU9Q6oiQNd0
http://rawkes.com/articles/creating-a-real-time-multiplayer-game-with-websockets-and-node.html
http://buildnewgames.com/real-time-multiplayer/