The Struggle of the Rubiniumite Wars

Timothy Mendez

in collaboration with
Evan Peterson, Darren Huang, Kyle Lozier

advised by
Zoë Wood

Winter ~ Spring 2015

June 12, 2015

http://52.24.1.150:8081

Abstract

The Struggle of the Rubiniumite Wars is a browser-based, one- to two-player, simultaneous turn-based strategy game set amongst the stars. It uses WebGL and Three.js for 3D graphics in the browser, Node.js for game engine and artificial intelligence design on the backend, and Socket.io for networking using websockets. The development group's inspiration, motivation, and reflections are discussed. Additionally, details on the development of the game engine, database integration with Parse, user registration with Nodemailer, graphics with Three.js and HTML/CSS, and audio with HTML5.

Introduction

Games blend creativity and engineering disciplines – they provide experience working with teams as well as constant opportunities to learn and familiarize oneself with new concepts and technologies. Developers with interests in widely-varied areas can collaborate to create a rich, fully-featured game, contributing their individual expertise to improve the overall game. When originally brainstorming possible senior project topics, our four-person team's interests included 3D graphics, Artificial Intelligence, Storytelling, improving our general skill in software development, and fulfilling graduation requirements. Creating a game seemed the natural choice to bridge these topics.

The Struggle of the Rubiniumite Wars (or simply, *The Struggle*), is a competitive strategy game set amongst the stars. Two players battle against each other, each controlling a fleet of starships on a shared grid game-board. The key feature of the gameplay is a simultaneous turn-based system: players plan and submit their units' moves to the server during the planning phase, and see how their moves fared against the opponent's during the playback phase. There are a few existing examples of games with similar style of gameplay, but this type of strategy game is largely unexplored in the game industry. The anticipation of carefully submitting a set of moves, bracing oneself for either disaster or triumph, and watching one's fleet either destroy or be destroyed hooks players on *The Struggle*.

The Struggle offers 3D graphics from within the browser using WebGL. Writing any kind of cross-platform code poses challenges, and this is especially apparent when developing OpenGL applications. Due to the nature of being "close to the metal," careful consideration must be given to the nuances of Windows, OS X, or Linux. By using a browser to deliver a 3D game, we're able to focus on the game and leave operating systems to the browser implementation. Besides the desire to work with GL graphics in a new setting, we chose to use WebGL to simplify cross-platform development. Any potential player using a modern browser that adheres to the WebGL standard will be able to easily load and play the game.

Related Works

Most of the gameplay inspiration for *The Struggle* came from the extremely popular *Fire Emblem* franchise. This was chosen as the basis for *The Struggle* due to its success, with the latest title in the franchise selling 1.57 million units globally (VGChartz, 2015). However, *The Struggle* was never meant to be a *Fire Emblem* clone, but the best of all worlds. Unfortunately, due to time constraints, one of the main aspects of *Fire Emblem*, the storyline, was not implemented.

Fig. 1: Fire Emblem

As Figure 1 shows, enemies have red circles around their locations to help spot them easier (Intelligent Systems, 2005). Early playtesters of *The Struggle* found difficulty in distinguishing user-controlled units from enemy-controlled units after a game had progressed. Thus, not only were red circles added around enemy units, but green circles were added around user-controlled units as seen in Figure 2.

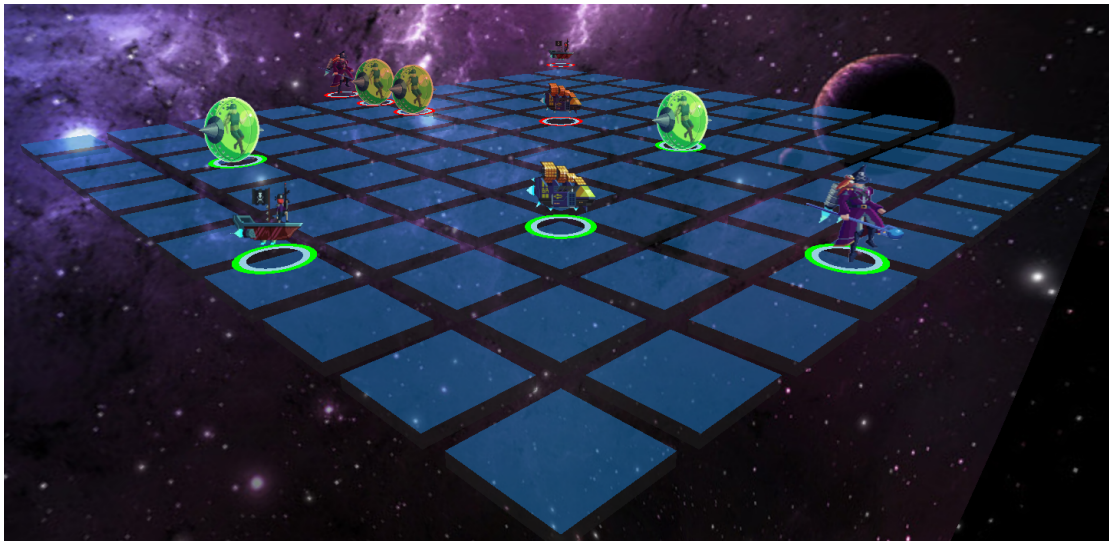

Fig. 2: The Struggle Grid

The Struggle is also built upon *Fire Emblem's* isometric-viewed grid system (Intelligent Systems, 2005). In Figure 1, one can see the rotated bird's eye view of a grid (Intelligent Systems, 2005). *The Struggle* incorporates this system for various reasons. Firstly, the rotation of the camera along the grid is a simple solution to abstract the game from checkers or chess. However, camera controls are also implemented, allowing the user to zoom in and out, and rotate the camera about the center of the grid. Camera controls ensure that the user is comfortable in viewing the gameplay. Second, the bird's eye view grants the player vision of the entire battlefield. Early on, a fog-of-war component (hidden enemies until within a certain tile count) was entertained, but ultimately scrapped due to not meshing well with the rotated bird's eye view of a grid.

Fire Emblem is lacking a player-versus-player component (Intelligent Systems, 2005). *The Struggle* aims to provide turn-based strategy online player-versus-player play.

A game that influenced the multiplayer aspect of the game is *Puzzle Pirates.* In *Puzzle Pirates*, players may play each other in the browser (Three Rings Design, 2003).

Fig. 3: Puzzle Pirates

As seen in Figure 3, the aesthetic appeal of *Puzzle Pirates* is quite limited. Yet, the game is still popular, having over 4 million users (James, 2008). This must be attributed to its multiplayer and gameplay aspects, as there is no storyline. Upon realizing this, *The Struggle* also took inspiration from *Puzzle Pirates* gameplay. The

main gimmick is that players move their ship, choose spaces to attack, and then submit their move. Then, all moves play out at once (Three Rings Design, 2003).

The Struggle has a similar idea. The player moves all his or her units, sets attack tiles, and presses the end turn button. Then, both the player's moves and the opponent's moves are played out at the same time. Though *Puzzle Pirates* allows for battles larger than two players, *The Struggle* does not (Three Rings Design, 2003). Originally dabbled with in the planning stages, the feature of having more than two players was scrapped due to the sheer number of units that would be required to be on the board at a given time; the game would be too hectic and packed.

Figure 1 and Figure 3 show that both *Fire Emblem* and *Puzzle Pirates* include obstacles. This, again, was another concept considered at the beginning of development. Illustrated in Figure 2, obstacles did not make it into the final iteration of the product. While obstacles could potentially add an extra layer to strategy, it was decided that they would take up too much space on the grid.

Under the Hood

The Struggle is split up into two key components: the server, which hosts the game engine and interacts with a database, and the client, which provides a visual interface to play the game via interacting with the server. Though it consists of two standalone parts, *The Struggle* is written entirely in JavaScript.

The Struggle is hosted on a free-tier Amazon EC2 instance hosted in Oregon (Amazon EC2). In order to ensure uptime, a script is running that attempts to launch the server every thirty seconds. Additionally, continuous deployment is used in the manifestation of dploy, linking *The Struggle*'s GitHub repository to the EC2 instance (dploy.io). Due to this, the server hardly ever needs to be touched.

To learn about other components, refer to the reports of the corresponding group members, referring to the following list:

Technologies and Developers

- WebGL graphics (Evan Peterson)
- Web server (Evan Peterson & Tim Mendez)
- Log in and registration (Evan Peterson & Tim Mendez)
- Two-player mode (Darren Huang)
- Logic and interface for skills and attributes (Darren Huang)
- Database interaction (Darren Huang & Tim Mendez)

- Unit selection interface (Tim Mendez)
- Game engine (Evan Peterson & Tim Mendez)
- Artificial intelligence (Kyle Lozier)
- UX (Evan Peterson & Tim Mendez)

Game Engine

At the heart of any game lies the engine. Built entirely in JavaScript, *The Struggle's* engine is a beautifully abstracted beast, ready for scalability and new features at any time.

The engine is located on the server, and interacts with players by sockets. When choosing to play another player, the client sends the *skirmish manager* a request for a *skirmish* against another player. The player then waits for the *skirmish manager* to receive another request for a *skirmish* against a player. Once the *skirmish manager* receives two requests, the players who sent the requests are paired against each other in a *skirmish*. If the player chooses to play AI instead, an AI player is created and matched with that player by the *skirmish manager*.

Each player's units are lined up on opposite ends of the grid. Players may then start entering moves. Players do so by selecting a unit. Tiles within that unit's movement range are then highlighted orange to show where the player may move as seen in Figure 4.

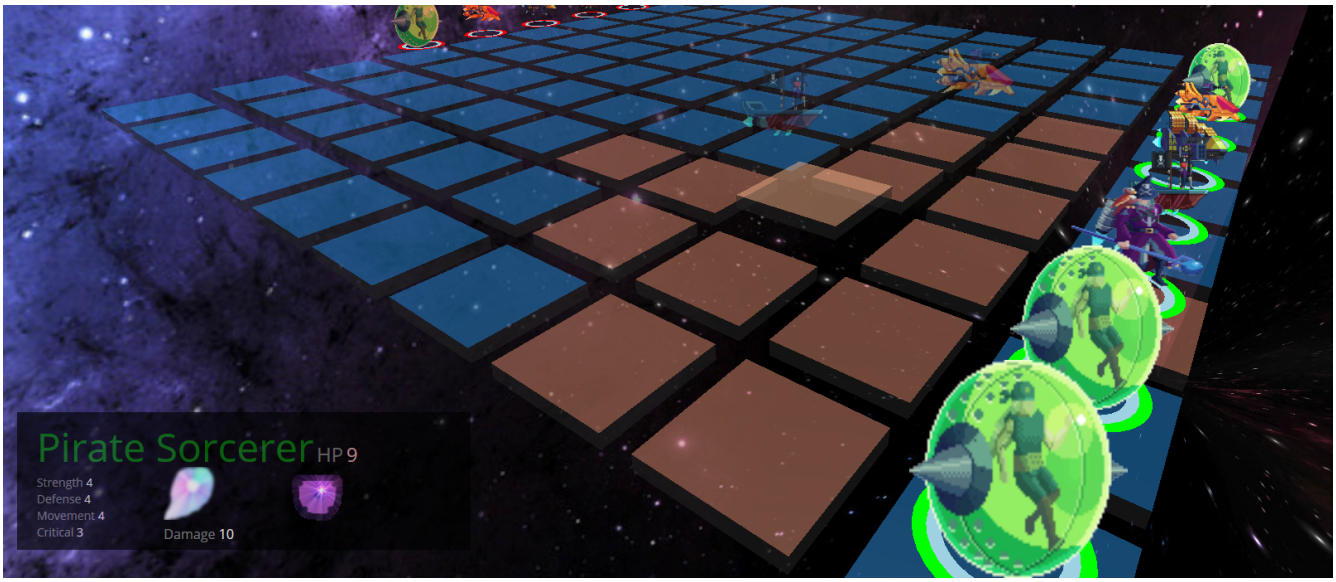

Fig. 4: Moving a Unit

Tiles are also raised wherever the player hovers his or her mouse to show the tile he or she is about to select. When the player decides on a tile to move a unit to, selecting that tile will cause a ghost image of that unit to appear on the tile, and attack tiles to be

highlighted (Figure 5). The player may rotate the mouse around the unit to attack in different directions. As Figure 5 shows, when hovering over an attack selection, all of the tiles are raised to show the player all of the tiles that will be attacked upon selecting that direction of attack.

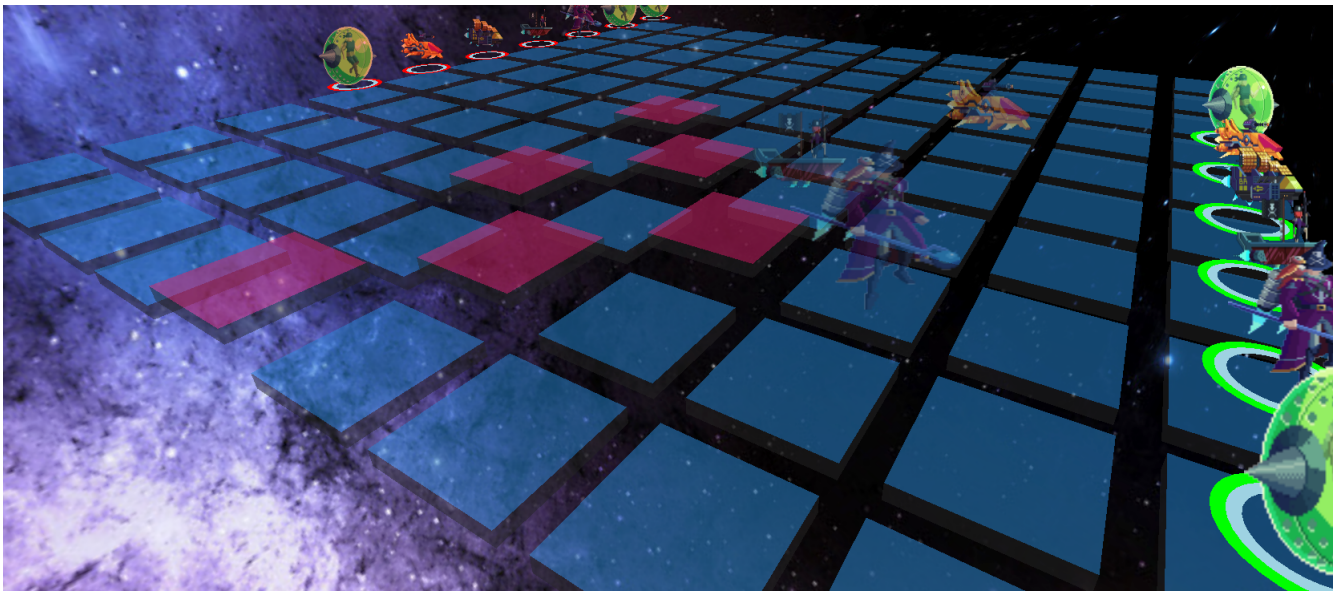

Fig. 5: Choosing Attack Direction

Selecting the attack direction saves the direction for that player's unit. Once players are finished moving and selecting attack directions for their units, they can submit their moves to the server.

The server waits for both players to submit their moves, then starts calculating the changes to the *skirmish*. First, the server will attempt to move each unit to their new location on the grid. If both players decide to move a unit to the same spot, the units both bounce back one space, and their attacks translate accordingly. Once all units have their new positions on the grid, the server calculates damage.

Attacks are sent to the server in a list of grid positions hit with a specific unit, so the server goes through each grid position hit, and updates the health any units located on attacked tiles accordingly. Critical hit rating, strength and weapon type of the attacking unit, and defense and armor type of the defending unit are all taking into consideration when calculating damage.

Damage is calculated as follows:

$$Damage = AttackerStrength - TargetDefense + (WeaponDamage \times Y) + Z$$

$$Where\ Y\ is\ \frac{3}{4}\ if\ the\ attacker's\ weapon\ is\ weak\ against\ the\ target's\ armor$$

$$and \quad Y\ is\ \frac{5}{4}\ if\ the\ attacker's\ weapon\ is\ strong\ against\ the\ target's\ armor$$

$$and \quad Y\ is\ 1\ if\ the\ attacker's\ weapon\ is\ neutral\ to\ the\ target's\ armor$$

$$Where\ Z\ is\ \frac{AttackerStrength}{2}\ if$$

$$Math.Random() \times 30 - Min(30, AttackerCritical) \leq 1$$

$$and \quad Z\ is\ 0\ if$$

$$Math.Random() \times 30 - Min(30, AttackerCritical) > 1$$

Fig. 6: Damage Calculation

The server then sends back the client all the information it needs to play out the turn to the player. All units move along paths to their final position, and bounce off of each other if there is a final position collision. Then, the camera will rotate to a side view of the units attacking, and go through each unit's attack. Each attacking unit jumps to show that it is the one attacking. The sound effect of the attacking unit's weapon is played when it attacks. Upon a hit, the hit unit will rotate. In the event of a critical hit, a critical hit sound will play, and the screen will flash white. All attacks are played out and deal damage, regardless if a unit is killed before it's attack animation is shown.

Players then repeat this process until there are no units left on the grid. The player with the last surviving unit wins the *skirmish*.

Database

Parse, a database service, is used to store data for *The Struggle.* Parse itself is built upon mongoDB, which is a key-value database management system (Sridha, 2014). Parse is an optimal choice for *The Struggle,* because a player's username can be mapped to store all of the information in a JSON object.

The server contains an API key to interact with Parse through object creation and a series of function calls. Adding data to the database requires a Player object to be created, setting the key to the player name, and the value to JSON data containing all the information necessary to create the player upon login. Querying data from Parse is just as simple: a Query object is created using the Player object stored in Parse, and then the “equalTo” function is called on the Query object. The “equalTo” function takes in a key parameter, and callsback on success or failure.

All of the database interaction is done from the server to Parse, which sanitizes inputs on behalf of *The Struggle.* Thus, due to abstraction and scrubbing, no user can access

any part of the database not allowed, and no malicious code can be injected.

Registration

Users may register for accounts to gain access to *The Struggle's* unparalleled content. Users first start by clicking the “Register” button on the home page. This allows the user to enter his or her email, desired username, and password. When the user presses “Register” after entering in the required information, the input is sent to the server for checks. During this time, the “Register” button is disabled so the user cannot spam the server with registration requests. However, even if multiple people attempt to register with the same email or password, the server checks against all pending registrations to prevent the race condition of a username/password combination being overwritten with a new password in the database. If the email address or desired username is already in use, or part of a current pending registration, the server will callback to the client with an error, displaying an error message for the user.

If the username and email are unique, then a ten character alphanumeric case-sensitive registration key is generated and stored on the server. Nodemailer is then used to send an email to the user's supplied email address containing the generated registration key (Reinman). Simultaneously, the client is called back with success, telling the user to check his or her email address to complete registration by entering in the registration key contained in the email. Once the user presses “Complete Registration”, the server is sent the registration key. The server attempts to validate the key against the stored registration keys. Upon not finding a match, the server will callback with failure and prompt the user to reenter the registration key. Upon finding a match, the server creates the account.

The server creates the Player object, hashes the user's password, adds a salt, and stores the information in Parse using the username as a key. Then, the registration key and account information mapped to it are removed from the server. Success is then called back to the client.

At this point, the user may enter his or her password and login to *The Struggle*.

Graphics

The graphics are mainly done in Three.js, which uses WebGL (Cabello, 2015).

The skybox rotates, the units bounce up and down and translate, the camera updates, Starduck talks, and tiles highlight and move. The client attempts to draw each of these animations at sixty frames per second. This ultimately gives *The Struggle* the smooth, high-quality animations that are seen in the final product.

Naturally, all of the graphics are rendered client-side. This means that the player's

system is responsible for calculating all of the mathematics required to animate *The Struggle*. In fairness to the player, quick and simple computations are used to calculate each frame. To cut down on memory usage, all actors are removed from the scene once they are no longer needed, such as killed units.

Creating the skybox involved a lengthy process of camera adjustment. The skybox texture is merely a .png file. Rendering a .png file was simple enough in Three.js, but encapsulating the camera in all directions proved to be a more challenging feat as seen in Figure 4.

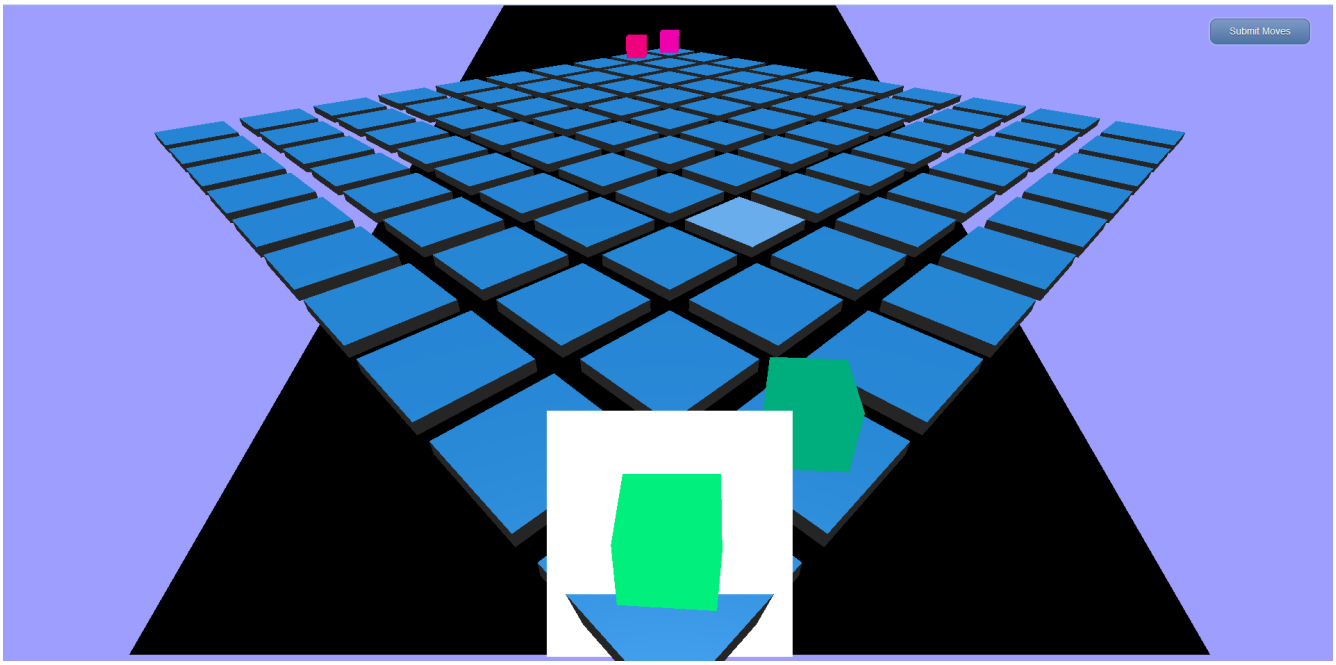

Fig. 7: Early Attempt at Skybox

Eventually, the camera and skybox were synchronized and Figure 7's background turned into Figure 2's background. Thanks to some extremely detailed tutorials and example code, *The Struggle*'s skybox turned into a skysphere (WebGL with Three.js).

The graphics that aren't done in Three.js are done using HTML and CSS. All of the buttons, the custom cursor, the unit selection, Starduck, and the heads up display (HUD) are done in HTML and CSS.

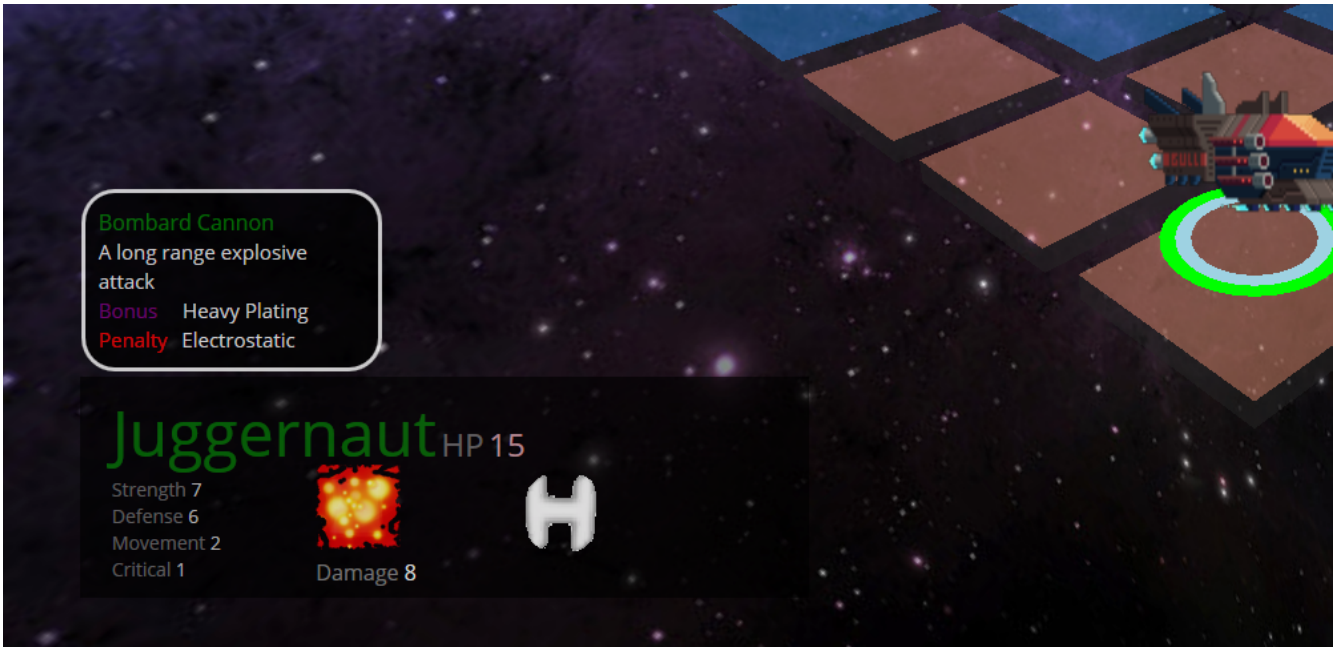

Fig. 8: Heads Up Display (HUD)

The HUD (Figure 8) was extremely difficult to implement. It relied on a lot of data that only the server had access to at first (unit attributes, updated health, etc.). The solution resulted in several new client-server communication functions and storing more data on the client's Unit object. This allowed the client to have access to variables such as the unit's health after being updated. The majority of the HUD is statically drawn, but the “HP”, unit name, and attributes are dynamically inserted into the HUD each time a unit is clicked.

The units all have wonderful looking sprites (see Figure 9).

Fig. 9: Sprites

The sprites are all pixel art done by a single commissioned pixel artist. They are all displayed using Three.js's Sprite object.

Audio

Sound is controlled through HTML5 audio. All audio is loaded at once on the client when the player loads the webpage. The audio.js file contains the statically created audio, making it very easy to add, remove, and play new sounds or music wherever necessary (Mozilla Developer Network).

The Struggle limits itself to only using .mp3 and .ogg files for audio. This allows for the audio to be high quality, while still taking into concern browser compatibility, disk space, and load time (Xiph, 2003).

All of the audio is either public domain or royalty-free. Even so, *The Struggle* does have custom mixed audio, created for its very purpose. For example, the critical hit sound effect is a public domain audio clip of knives scraping together, but overlayed thrice, offset, and sped up. Additionally, all of Starduck's audio was contracted by a freelancer on Fiverr.com. Unfortunately, the voice actor fell off the grid shortly after delivering the Starduck audio; *The Struggle* is missing some important Starduck-driven player feedback such as letting the player know that he or she may use WSAD to control the camera.

Knowing that music can sometimes be too repetitive, *The Struggle* allows players to toggle the music by clicking on the sound button in the top left of Figure 10.

Fig. 10: Audio Toggle Button

Results

The final result of *The Struggle* is a well designed, well written, and fun turn-based strategy game.

Players can choose to take the game seriously and spend hours upon hours perfecting strategy and unit combinations for enjoyment, or they can just mess around to have fun (Figure 11).

Fig. 11: Hamster Ball Guy War

The Struggle was officially playtested by peers, family, and the roommates of the developers; they were asked to provide feedback on the project. Playtesters across the board seemed to mainly agree on three things:

1. The game is fundamentally fun.
2. It looks very nice.
3. There needs to be more explicit instructions.

Not knowing where enemy units will be seems to be the driving force behind the fun. The anticipation a player feels when the *skirmish* is down to one unit left for each player is exhilarating. It is moments like the one in Figure 12 that keep players hooked on *The Struggle* and coming back for more.

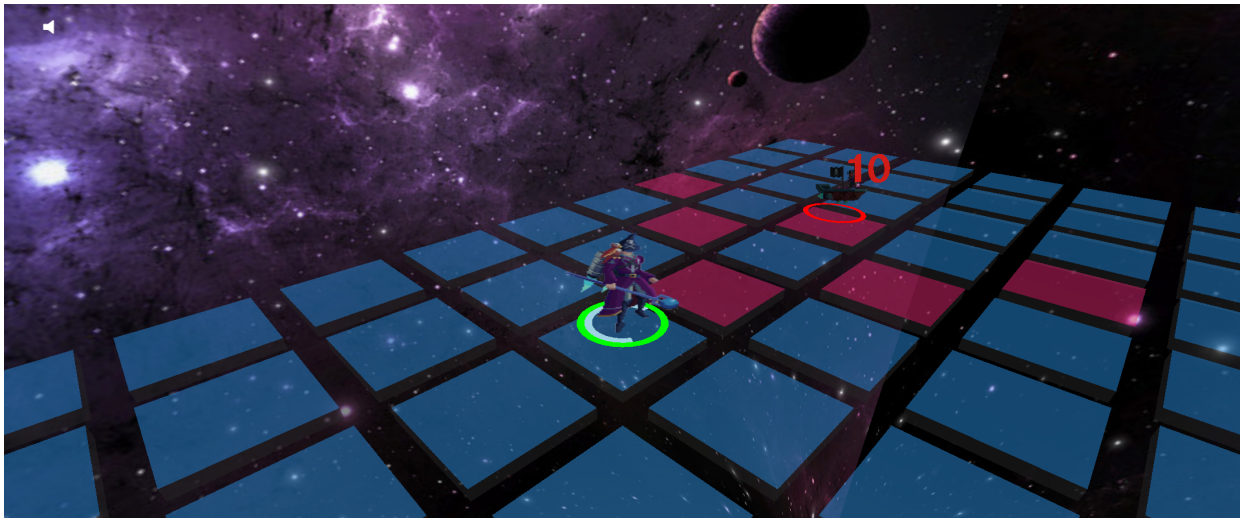

Fig. 12: One Unit Left

The Struggle prides itself on production quality. It was made sure that the game would not look like a run-of-the-mill computer science senior project game. Effort and finances went into *The Struggle* to help push it closer towards being a game desired to be played. From the art to the music, sound effects, and voice acting, the goal of an immersive environment has been reached. One playtester, Mark Mendez, even said that *The Struggle* "feels like an actual game" due to the production quality. Another playtester, Carl Lind, said that he "really liked the colors and animations of the moving characters" and that "the art work was pretty nice."

However, even with all the praise, criticism must be addressed. Playtesters report being confused on exactly how the game works and how to use certain features. Playtester Sam Romano said that *The Struggle* needed "a tutorial screen, or some way to learn how to play." With this feedback, it is determined that the game was written in a way that pandered to veteran players of the genre. A playtester with much Fire Emblem experience, Mark Mendez, had no trouble figuring out how to play. However, even players who do not have much strategy experience ended up liking the game after a few rounds. Carl Lind said "I really liked this game and I don't even like these kinds of games."

The camera controls of WSAD are not intuitive to most players in a strategy game, and not explicitly stated for use in *The Struggle.* Friendly fire is enabled, but players have to attack their own units in order to find this out. Attacks are not visually saved for the player when setting moves, so it is very easy for players to not see where they are attacking, and end up attacking their own units by accident. Thus, *The Struggle* fails to provide optimal feedback and instructions to the player.

Ultimately, the project is still considered a success by its creators. It is a fun and challenging game, built entirely in JavaScript, HTML, and CSS, thoroughly enjoyed by playtesters. One playtester's comments speak to this: Dannie Alfaro summed his experience up nicely by saying that *The Struggle* "is overall fun."

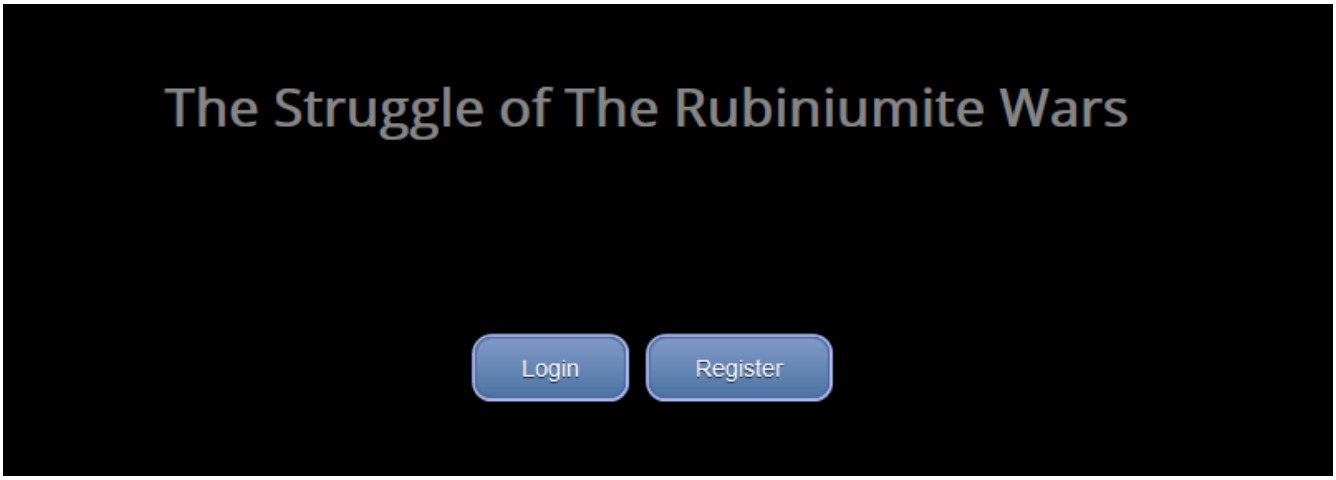

Fig. 13: Join The Struggle

Future Work

Initially reaching for the stars, *The Struggle* is built upon a foundation allowing for major expanse. As a result, the codebase is extremely readable with major parts abstracted from each other. Every pull request went under extreme scrutinization to ensure quality code, readability, and the potential for future changes. The following are features that may be added in the future:

Upgrades/Currency

Upgrades can be purchased from an upgrade shop using the in-game currency: rubiniumite. Rubiniumite is awarded after *skirmishes* to the victor. Players may choose to upgrade specific weapon types, armor types, and unit attributes. For example, a player could purchase the Level 2 Darkon armor upgrade, which would give all of the player's units with Darkon armor two extra defense points. A player could also purchase the Level 1 Critical upgrade for Marauders, which would add a point to all of the player's Marauders' critical attribute.

Particle Effects

Fig. 14: Particle Effects in Dota 2

The Struggle would very much liked to have incorporated particle effects. However, due to time constraints, particle effects did not make the final cut. The aim of particle effects was to have all weapons have their own projectile – hopefully looking akin to the particle effects shown in Figure 14.

They would optimally be written in Three.js in order to keep dependencies at a minimum.

Player-versus-Player Ladder System

A full ladder system has been designed for *The Struggle of the Rubiniumite Wars*, but has not been implemented.

There are six different brackets separated by unit supply count

- Cabin Boy: 3-5 supply — 3 badges for win, -1 for loss, can't go below 2
- Knave: 6-10 — 6 badges for win, -2 for loss
- Scallywag: 11-16 — 12 badges for win, -5 for loss
- Quartermaster: 17-24 — 24 badges for win, -11 for loss
- Jack Ketch: 25-34 — 48 badges for win, -23 for loss
- Captain: 35-47 — 96 badges for win, -47 for loss

Every fresh account starts off at 3 supply. Players create an *Armada* (personal unit pool) out of all available units. For example, a player with five supply points may create an *Armada* consisting of two Hamster Ball Guys (2 supply points each) and the level 1 upgrade to the Ammunition weapon type. This player may be matched against another player who only has an interceptor (3 supply points) and the level 1 upgrade to Aluminum plating. The winner of the battle receives 1 permanent supply point (for longevity of the ladder climb while steering clear of ridiculously huge battles). Players may choose to enter a lower ladder bracket if they choose not to use all available unit supply.

Badges are used to rank players globally. The player with the highest badge count is ranked as number one. Each unit point of difference in supply count usage in a battle affects the badge loss by one. It cannot go below zero. For example, Player1 is using 25 supply battling Player2 using 28 supply. If either player wins, he will receive 48 badges (and one supply point). However, if Player1 wins, Player2 will lose 23 badges, but if Player2 wins, Player1 will lose 20 badges.

Going Mobile

The Struggle would be very difficult to transfer to a mobile application in its current state; much work would be required to build the application natively on a separate platform.

However, once transferred to a mobile platform, the game would be able to generate copious amounts of revenue. Additional features that can be added to make money include:

- Microtransactions

    - Players would be allowed to purchase rubiniumite in an in-game store in order to purchase more upgrades.
- Time Cooldowns
    - Players may only play X amount of games per day, but may reduce the timer by spending rubiniumite.
    - Upgrades take on the order of days to complete once purchased, but may be sped up or completed instantly by spending rubiniumite.
- Pro vs Free Version
    - The pro version would be ad-free and cost a fee to initially download, whereas the free version would be a free download, but have in-game advertisements.

References

Figure 1: Fire Emblem - http://199.101.98.242/media/images/66764-Fire_Emblem_-_Path_of_Radiance_%28Europe%29_%28En,Fr,De,Es,It%29-3.jpg

Figure 3: Puzzle Pirates - http://www.lgdb.org/sites/default/files/node_images/1260/2330.jpg

About Node.js. (n.d.). Retrieved June 11, 2015, from https://nodejs.org/about/

Amazon EC2 Product Details. (n.d.). Retrieved June 11, 2015, from http://aws.amazon.com/ec2/details/

Cabello, R. (2015, March 16). Three.js. Retrieved June 12, 2015, from https://github.com/mrdoob/three.js

Continuous deployment for everyone. (n.d.). Retrieved June 11, 2015, from http://dploy.io/

Fire Emblem: Awakening. (2015, April 4). Retrieved June 11, 2015, from http://www.vgchartz.com/game/70704/fire-emblem-awakening/

Intelligent Systems. (2005). Fire Emblem: Path of Radiance [Nintendo GameCube video game]. Japan: Thoru Narihiro, Hitoshi Yamagam.

James, D. (2008, December 8). A Letter from the Captain. Retrieved June 11, 2015, from http://www.puzzlepirates.com/newsletter/spyglass/2008/dec.xhtml

Mozilla Developer Network. (n.d.). Using HTML5 audio and video. Retrieved June 11, 2015, from https://developer.mozilla.org/en-US/docs/Web/Guide/HTML/Using_HTML5_audio_and_video

Reinman, A. (2015, April 1). Nodemailer. Retrieved June 12, 2015, from https://github.com/andris9/Nodemailer

Sridha, P. (n.d.). Parse. (2014). Retrieved June 11, 2015, from https://www.parse.com/questions/which-type-of-database-does-parse-use-sql-or-nosql

Three Rings Design. (2003). Puzzle Pirates [PC video game].

Vorbis.com: FAQ. (2003). Retrieved June 11, 2015, from http://www.vorbis.com/faq/

WebGL With Three.js – Lesson 5. (2014, June 12). Retrieved June 11, 2015, from http://www.script-tutorials.com/webgl-with-three-js-lesson-5/