Samuel Crispin

Project Documentation

Heart Of Demaxius

Contents

[Introduction 2](#_Toc523755355)

[Proposal 3](#_Toc523755356)

[Specifications 5](#_Toc523755357)

[Design Methodology 6](#_Toc523755358)

[Design 7](#_Toc523755359)

[Implementation 16](#_Toc523755360)

[Testing 23](#_Toc523755363)

[Refactoring 26](#_Toc523755364)

[Screen Shots 27](#_Toc523755365)

[Conclusion 29](#_Toc523755366)

# Introduction

Heart of Demaxius (abbreviated to HoD) is a game based off of another game called *FTL: Faster Than Light* developed by Subset Games. *FTL* is outlined best as a sci-fi space travel roguelike game with an advanced point and click, real time combat engine. HoD is similarly described as a fantasy roguelike game with a rudimentary point and click combat system.

There were a few factors that influenced how this decision came about. I have always wanted to make a full blown functioning game, many attempts have been made and a lot of them were semi successful and good learning experiences, but none of them were ever finished. As for the basis of the game, the roguelike genre has always been a personal favourite, with *FTL* being a great example. The roguelike genre is best defined as a subgenre of the RPG (role-playing game) identified by dungeon crawling through procedurally generated levels, turn-based combat/gameplay and the most iconic characteristic, permanent death for the player. When a player dies in a roguelike game, they lose all progress and have to start again making the difficulty curve generally quite high and the replayability of a good roguelike incredible.

As for the fantasy aspect, fantasy is another favourite of mine, whether it be in books, films, TV, games etc. SciFi was a close second as a genre but seeing as *FTL* is already based around space travel and combat, over plagiarising was something to stay away from. If done again, the biggest improvement to make would be to the combat engine. The current one is primitive and unimpressive but it was the best that I could plan for given time constraints and capabilities at the start of the project.

Overall, the project went fairly well. There was quite a bit of deviation from the original proposal for multiple reasons which will be talked about later.

# Proposal

Overview

The general concept of the project will be a text based (with perhaps some slightly graphical elements) RPG. The player plays as a person fighting a series of randomized enemies, each getting progressively harder until an eventual “boss” fight. If the player wins this fight then they will have the option to end the game after defeating the boss or carry on until they are eventually defeated (or clear the map and then the player will get the same option again).

Story

The protagonist is the grandson of a disgraced warlord by the name of Evandar Hrothgar, Evandar’s grandfather fled the scene of a great battle between the Kingdom of Demaxius and the Dark Elves of the south beyond the Dead Plains (a large Death Valley like dessert). The war lasted years but the Elves were eventually repelled hence why the Demaxius still stands.

Evandar’s family has since had no place as descendants of a deserter within the new rebuilt Kingdom and he strives to find a way to recover honour and pride back to his family. Evandar has heard tales of an ancient, precious and powerful relic that was lost during the Great War, supposedly stolen by the Elves over its rumoured power.

Evandar plans to retrieve the relic and restore some honour to his name. The story starts as he has just made the treacherous and unlikely journey across the Dead Plains and has arrived at the Elven town of Elaris. Evandar has very little on his person, just a few gold coins left over from his time spent in Demaxius, now worth little here due to the abundance of gold mines south of the Plains, The Elves favour a bronze based currency due to it being more expensive. He has his trusty oak staff and his mastercrafted broadsword passed down from his grandfather.

As much as there is a ‘peace’ between humans and Dark Elves, there is still tension between the two races.

The player controls Evandar around the town and eventually into multiple caves/dungeons/ruins in search for the fabled relic. The player eventually comes across an encounter that involves a cave troll who has hoarded lots of gold and other treasure, including the relic that Evandar is looking for. If the player hasn’t cleared all of the other points on the map yet, they will not be allowed to fight the boss yet.

Gameplay

The gameplay itself is inspired by a few different sources, the main one being a game by the name of *FTL: Faster Than Light.* A space travel game where interactions are completely randomized (dependant on the players location within the game, previous interactions and equipment and crew etc.). Interactions between Evandar and the inhabitants of the Land of the Elves will be randomized like interactions in *FTL* as well as having different options depending on your progress, characters stats, inventory etc. Outcomes of interactions will be mostly random with some having fixed outcomes. The game will also try to follow the difficulty of *FTL* in which death means restarting the game.

When defeating an enemy, it will drop loot. The looting system will be inspired by games like the *Diablo* or *Borderlands* series in which every enemy will drop some form of loot. This loot can be as little as some currency all the way up to legendary weapons or accessories.

The game map will be randomly generated, a 20x30 grid will be used to represent where points on the map can be with a random number of points (between 15 and 20) generated and placed on the map. When clicked on, the player will be taken to a location with an encounter. The encounter will be chosen randomly from a pool of set interactions, this encounter will then be taken out the pool so it’s not repeated in another location. One location will always be a town which will include different shops, such as a merchant (low level gear as well as food and other supplies), smithy (higher level gear), an inn (stay the night to restore health) and an arcane merchant (potions and occasionally magical gear with special effects). The town will have 3 different layouts, one of which will be chosen randomly so it won’t be as repetitive (obviously it won’t change throughout a single game).

Enemies stats will be randomly generated within certain bounds (depending on how far in the player is through the story and potentially the difficulty chosen, these stats will be used to generate a score for the player defeating the enemy which will be totalled into a leaderboard.

The boss fight location will be chosen randomly but the player won’t be able to access the fight until the player has cleared every other location. After the player has defeated the boss, they will get the message ending the story and then asked the question if they would like to continue to try and beat a high score. The map will be regenerated and the player carries on with their current inventory.

# Specifications

As mentioned previously *FTL* was the main inspiration for this game and its mechanics. Because of this, all the navigation and all the combat is done by point and click (*FTL* also uses keyboard short cuts for navigation, for example, being able the press the 1 key will have the same effect as clicking the first option in a menu however that isn’t the case in this project) and the game is very menu based, as shown in the screenshots just later in the document.

Basic Feature Overview

* Menu based navigation via point and click
* Map populated with clickable points that include interactions
* One of the points being a town which includes 4 visitable shops
* Shops randomly generate items which the player can buy and equip
* Shops regenerate items after a point has been cleared
* One of the shops is an inn where you can spend the night and restore health
* Have an inventory menu where you can access different item types and equip them
* Separate screens for each item type
* Make tooltips appear with the items stats when you hover over them
* Combat system
  + Turn based, player will have a way of knowing it’s their turn
  + Will have 2 options: attack and eat
  + Attacking does damage to enemy
  + Eating something from your inventory heals you by a set amount
  + Damage will be randomly generated between 2 amounts depending on players attack and enemy’s defence
  + If players health is reduced to 0, player loses the game and has to restart
  + If enemy’s health is reduced to 0, the player wins the battle and then they have the option to collect loot
* Loot system
  + A random number of items (between 1 and 3, with 1 being more common than 2 and 2 more common than 3) is generated and the player can take as many of them as they want
  + The items stats will depend on how far through the game the player is, again randomly generated between 2 bounds
  + 9 types of item (listed in my implementation section under pseudocode so I won’t repeat it here)
  + Will generally be called after a victorious battle but can be called from an interaction function as well
  + Consumables won’t be generated as loot, just equipment

# Design Methodology

As for design methodologies, this project is a mix and match of a few different ones. At a basic level, it is bottom up, separate parts of the game were created and then later merged together to form a cohesive whole. At the bottom level, it is a combination of object oriented, functional and procedural programming. The object oriented side of things came into the project in the much later stages due to the fact that I didn’t really have a grasp of objects and how to use them at the start of this project, also aided by the fact that JavaScript isn’t really an objectified language like Java of C++. Although it came in later into the project, a lot of refactoring did go into the earlier parts of the project to accommodate objects and their functionality. Things like turning the players separate statistics into one player object along with methods related to the player, alongside creating other functions to help reduce repeating code, for example, a changeHealth function instead of repeatedly writing code to alter the players health from certain events.

All code design was done in the same way, pseudocode was written down on paper and was used as the only step of code planning/design. The pseudocode written was quite extensive and thought over and the code rarely changed after being written because of overlooking aspects of the process being planned. There was refactoring done at a later stage but that isn’t due to the lack of thoroughness in the pseudocode. This thoroughness in the planning stage allowed for the implementation of the code to be a very smooth process in general, there were bugs in some parts of the pseudocode but that is to be expected in development and were ironed out quickly.

Pen and paper was also used for the design of graphical assets, this includes things such as user interfaces, images/icons and heads up displays. These were then sketched on Microsoft Paint to then be used the game. Again, most designs stayed true from design to implementation but some were changed due to miscalculations in pixel measurements or change of opinion

A lot of planning for the games background, lore and interactions was also done on paper, planning the games map, character background and writing down interaction ideas for example.

# Design

* Front end: HTML, CSS and JavaScript
* Game logic: JavaScript
* Code design: hand written pseudocode on pen and paper
* Graphical design: hand drawn on pen and paper
* Graphical assets: drawn in MS Paint

## High Level Overview

### Player Object

The Player object wasn’t implemented at the start of the project, but later with the implementation of the rest of the objects and OO methodologies, most of the functionality had been implemented but not in an efficient manner. The Player object was intended to contain everything concerning the player, setting up the player’s inventory, changing the player’s health and gold, their stats (health, attack, defence etc.) and handling the player’s death.

//Setting up players stats  
health: {  
 //'base' refers to the amount of that stat the player starts with  
 base: 100,  
 //'current' refers to how much of that stat they currently have  
 current: 100,  
 //'bonus' refers to the stats the player gets from items  
 bonus: 0,  
},  
attack: {  
 base: 10,  
 current: 10,  
 bonus: 0,  
},  
defence: {  
 base: 10,  
 current: 10,  
 bonus: 0,  
},  
charisma: {  
 base: 10,  
 current: 10,  
 bonus: 0,  
},  
intelligence: {  
 base: 10,  
 current: 10,  
 bonus: 0,  
},  
gold: 1000,  
//Instantiating inventories  
inventory: [],  
equipped: [],  
  
setUpPlayer: **function** () {  
 //Called on setup, instantiates/resets player  
 **this**.resetHealth();  
 **this**.changeGold(0);  
 **this**.setUpEquipment();  
 **this**.updateStats();  
},  
  
setUpEquipment: **function** () {  
 //Called on setup  
 //Creates an inventory of dummy items  
 **this**.equipped = [];  
 **for** (**var** i = 0; i < 6; i++) {  
 //Creates a blank item template to be modified later  
 **var** item = **new** Item;  
 item.consumable = **false**;  
 item.regen = 0;  
 item.health = 0;  
 item.attack = 0;  
 item.defence = 0;  
 item.charisma = 0;  
 item.intelligence = 0;  
 item.power = 0;  
 //Gives the blank item a type so there is an item in each inventory slot  
 **switch** (i) {  
 **case** 0:  
 item.type = "helmet";  
 **break**;  
 **case** 1:  
 item.type = "amulet";  
 **break**;  
 **case** 2:  
 item.type = "armour";  
 **break**;  
 **case** 3:  
 item.type = "leggings";  
 **break**;  
 **case** 4:  
 item.type = "boots";  
 **break**;  
 **case** 5:  
 item.type = "sword";  
 //The equipped sword is given some small base stats to make the early game slightly  
 //easier to get through as there were instanced where the player couldn't get past fight 1  
 item.attack = 5;  
 item.power = 5;  
 **break**;  
 }  
 **this**.equipped.push(item);  
 }  
},  
changeHealth: **function** (health) {  
 //this is called whenever the player's health needs changing, be it from a fight, an interaction etc  
  
 //This is the return value, there are bits of error handling on some of the function calls, this value returns:  
 //true for success and false for failure  
 **var** healthChanged = **false**;  
  
 **if** (health > 0) {  
 //If health is being changed, health is 0 in certain occasions just to update the HUD  
 **this**.health.current += health;  
 healthChanged = **true**;  
 }  
 //Health can be a negative value, if current health goes below 0, set it to 0  
 **if** (**this**.health.current < 0) **this**.health.current = 0;  
  
 //If current health is greater than max health, set it to the max health  
 **if** (**this**.health.current > (**this**.health.base + **this**.health.bonus)) **this**.health.current = (**this**.health.base + **this**.health.bonus);

//Update HUD  
 document.getElementById("healthBar").style.width = ((**this**.health.current / (**this**.health.base + **this**.health.bonus)) \* 100) + "%";  
 document.getElementById("healthLabel").innerHTML = **this**.health.current + "/" + (**this**.health.base + **this**.health.bonus) + "hp";  
  
 **if** (**this**.health.current == 0) **this**.die();  
 **return** healthChanged;  
},  
  
changeGold: **function** (gold) {  
 //called from whenever the player loses/gains gold, be it from shopping, loot or an interaction  
 **if** (gold < 0) {  
 gold = -gold;  
  
 //If you don't have enough gold  
 **if** (gold > **this**.gold) **return false**;  
  
 **this**.gold -= gold;  
 //Update HUD  
 document.getElementById("gold").innerHTML = **this**.gold + " gold";  
  
 //The return value is for the same reason as in changeHealth, the are instances  
 //that the return value is used for error handling  
 **return true**;  
 } **else if** (gold > 0) {  
 **this**.gold += gold;  
 }  
 //For the same reason as in changeHealth, gold can be 0 just to update the HUD  
 document.getElementById("gold").innerHTML = **this**.gold + " gold";  
},  
resetHealth: **function** () {  
 //called on startup/reset  
 //Sets players max and current health to 100  
 **this**.health.current = 100;  
 **this**.health.bonus = 0;  
 **this**.health.base = 100;  
},  
  
updateStats: **function** () {  
 //called whenever the players stats are modified, whenever they equip an item for example  
 //recalculates all stats bonus and current values according to appropriate equipped items  
 **this**.health.bonus = **this**.equipped[0].health + **this**.equipped[2].health + **this**.equipped[3].health + **this**.equipped[4].health;  
 **this**.attack.bonus = **this**.equipped[1].attack + **this**.equipped[5].attack;  
 **this**.defence.bonus = **this**.equipped[0].defence + **this**.equipped[2].defence + **this**.equipped[3].defence + **this**.equipped[4].defence;  
 **this**.charisma.bonus = **this**.equipped[1].charisma;  
 **this**.intelligence.bonus = **this**.equipped[1].intelligence + **this**.equipped[5].intelligence;  
  
 //current health isn't calculated otherwise every time a stat changes, health would go back to max  
 **this**.attack.current = **this**.attack.base + **this**.attack.bonus;  
 **this**.defence.current = **this**.defence.base + **this**.defence.bonus;  
 **this**.charisma.current = **this**.charisma.base + **this**.charisma.bonus;  
 **this**.intelligence.current = **this**.intelligence.base + **this**.intelligence.bonus;  
  
 //updates HUD for health (discussed in its function)  
 **this**.changeHealth(0);  
},  
die: **function** () {  
 //called when the player loses all health, from changeHealth  
 //opens the defeat screen and cleans up the screen  
 nav.open('defeat');  
 document.getElementById('enemyHealthBarBG').style.display = 'none';  
 document.getElementById('enemyHealthLabel').style.display = 'none';  
 inBattle = **false**;  
}

### Navigation

This was an object, like the Player object, that was implemented later in the projects life. Before this object, the ‘open()’ was a standalone function with some slight changes from function to method but the main functionality has stayed the same. It closes the currently open div, and opens the screen passed in via the first argument. ‘back()’ is a new method made easier by it being included in an object because the ‘prevOpenDiv’ can be completely encapsulated within the object rather than having to use a global variable.

prevOpenDiv: "",  
  
open: **function** (ID, completedInteractionID) {  
 //called whenever a different has to be opened  
 **var** list = document.getElementsByClassName('outputDiv'), interactionID = "", count = 1;  
  
 //if the screen being opened is an interaction (eg. output10 will open interaction 10)  
 **if** (ID.includes('output')) {  
 //extracts the interaction number from the end of the argument  
 **while** (parseInt(ID.slice(ID.length - count)) || parseInt(ID.slice(ID.length - count)) === 0) {  
 interactionID = parseInt(ID.slice(ID.length - count));  
 count++;  
 }  
 //if there is an interaction being opened  
 **if** (interactionID !== "") {  
 interactionID = (interactionID === 0) ? "Boss" : interactionID;  
 //if its complete, don't do anything  
 **if** (mapObj[interactionID].complete) {  
 **return**;  
 }  
 //the div has an ID of 'output', removing the number from the argument essentially  
 ID = "output";  
 //changes the contents of the 'output' div  
 setOutputDiv(100, 'openingScr' + interactionID);  
 }  
 }  
 //hides the open div  
 **for** (**var** i = 0; i < list.length; i++) {  
 **if** (list[i].style.display == 'block') {  
 **if** (list[i].classList.contains('interaction') && (ID == 'inventory') && (list[i].id != 'openingMenuScr2')) {  
 **return**;  
 }  
 list[i].style.display = 'none';  
 **this**.prevOpenDiv = list[i];  
 **break**;  
 }  
 }  
 //if a point has been completed (eg. after a fight, going onto the loot screen or the last screen of an interaction)  
 **if** (**typeof** completedInteractionID != "undefined" && completedInteractionID != "null") {  
 //regen the shops  
 genShop('merchant');  
 genShop('aMerchant');  
 genShop('smithy');  
 pointsCleared++;  
 mapObj[completedInteractionID].complete = **true**;  
 document.getElementById("coord" + mapObj[completedInteractionID].Y + "," + mapObj[completedInteractionID].X).style.backgroundImage = "url(img/mapIconDone.png)"  
 }  
  
 **if** (ID == "loot") genLoot();  
 //clean up stray divs and then open the output  
 document.getElementById('shopAnnouncement').style.display = 'none';  
 document.getElementById('backButton').style.display = 'none';  
 document.getElementById('doneButton').style.display = 'none';  
 document.getElementById(ID).style.display = 'block';  
},

back: **function** () {  
 //called when the back button is clicked, back button is only shown on certain screens  
 //namely shops and the town   
 **var** list = document.getElementsByClassName('outputDiv');  
  
 //hides the open div  
 **for** (**var** j = 0; j < list.length; j++) {  
 list[j].style.display = 'none';  
 }  
  
 //opens the div open before it  
 **this**.prevOpenDiv.style.display = 'block';  
 document.getElementById('backButton').style.display = 'none';  
 document.getElementById('shopAnnouncement').style.display = 'none';  
}

### Fight Screen

The fighting was designed and implemented later than the player and navigation objects so it was designed to be an object from day one. ‘genEnemy()’ was the only exception, it was a separate function before I made this object but I later moved it in as it is only used when fighting. The first thing called from this object is always ‘setUpFight()’, this is called from ‘buildFight()’ which is an intermediate function between the onClick of an interaction that leads to a fight. ‘setUpFight()’ then goes on set up the fight and then waits for user input.

playerTurn: **true**,  
enemy: **null**,  
interactionNo: **null**,  
  
setUpFight: **function** (enemyType, interaction) {  
 //Called as a prerequisite for a fight beginning  
 **this**.interactionNo = interaction;  
 nav.open('fight');  
 **if** (interaction != "bossFight") {  
 //if its not a boss fight, generate a normal random enemy  
 **this**.enemy = **this**.genEnemy(enemyType);  
 } **else** {  
 //if it is the boss fight, use this set enemy  
 **this**.enemy = **new** Enemy(enemyType, 200, 100, **null**, 100, 100);  
 }  
 //this flag is used for when the player is consuming something from their inventory  
 //the 'equip' function see the flag is set to true and makes it so it takes a game turn  
 //so that its the enemy's turn after  
 inBattle = **true**;  
 **this**.changeEnemyHealth(0);  
 //sets up the few divs that aren't contained the output div and turns the player's  
 //name yellow to indicate it's their turn  
 document.getElementById('enemyHealthBarBG').style.display = 'block';  
 document.getElementById('enemyHealthLabel').style.display = 'block';  
 document.getElementById('playerName').style.color = 'yellow';  
 document.getElementById('enemyName').innerHTML = **this**.enemy.name;  
  
 //determines enemy's first turn, talked about more in enemyTurn()  
 **if** (Math.random() > 0.8) {  
 **this**.enemy.intention = 'block';  
 } **else** {  
 **this**.enemy.intention = 'attack';  
 }  
 document.getElementById('enemyIntention').innerHTML = **this**.enemy.intention;  
  
 **this**.outputStats()  
},  
  
outputStats: **function** () {  
 //called at the start of the fight, concatenates players stats and  
 //enemy's stats and displays them in two separate divs  
 //also called when an enemy defends, talked about in enemyTurn()  
 **var** output = "";  
 output += "Attack: " + player.attack.current + "<br>";  
 output += "Defense: " + player.defence.current + "<br>";  
 output += "Charisma: " + player.charisma.current + "<br>";  
 output += "Intelligence: " + player.intelligence.current + "<br>";  
 document.getElementById('playerStats').innerHTML = output;  
  
 output = "Attack: " + **this**.enemy.attack + "<br>";  
 output += "Defense: " + **this**.enemy.defence + "<br>";  
 document.getElementById('enemyStats').innerHTML = output;  
},  
  
attack: **function** (playerAttacking) {  
 //playerAttack is a boolean argument that is true if the player is attacking and false if enemy is attacking  
 //determined by how its called  
 //called from the onClick function of the player's attack button in the fight screen  
 //argument is true in this case

//if it's not the players turn, and the player is trying to attack, do nothing  
 **if** (!**this**.playerTurn && playerAttacking) **return**;  
  
 **var** damage = 0, enemyDead = **false**, dmgUp, dmgDown, x;  
  
 //if its the player's attack  
 **if** (playerAttacking) {  
 x = **this**.enemy.defence;  
 //effectively capping the enemy's defence at 75  
 **if** (x > 75) x = 75;  
  
 //determines which weapon type is equipped and uses that as the attack type  
 //damage formula takes into account attack/intelligence and defence to come up with a number for damage  
 **if** (player.equipped[5].type == 'sword') {  
 damage = Math.floor(player.attack.current - (player.attack.current \* (x / 100)));  
 } **else** {  
 damage = Math.floor(player.intelligence.current - (player.intelligence.current \* (x / 100)));  
 }  
  
 //this creates 2 bounds that the damage can vary from, 10% either way  
 //this gives the battles some sort of variation in damage rather than the same every time  
 dmgUp = Math.round(damage + (damage \* 0.1));  
 dmgDown = Math.round(damage - (damage \* 0.1));  
 damage = Math.floor(Math.random() \* (dmgUp - dmgDown)) + dmgDown;  
  
 //changes enemy health whilst checking if the enemy has been slain  
 enemyDead = **this**.changeEnemyHealth(damage);  
 } **else** {  
 x = player.defence.current;  
 //caps player defence as well as enemy defence  
 **if** (x > 75) x = 75;  
  
 //this is the same damage formula used for the player, just against them this time  
 damage = Math.floor(**this**.enemy.attack - (**this**.enemy.attack \* (x / 100)));  
 dmgUp = Math.round(damage + (damage \* 0.1));  
 dmgDown = Math.round(damage - (damage \* 0.1));  
 damage = Math.floor(Math.random() \* (dmgUp - dmgDown)) + dmgDown;  
  
 //changes the player's health, checking whether the player dies is done within the function  
 player.changeHealth(-damage);  
 }  
 **if** (playerAttacking && !enemyDead) {  
 //highlights the enemy's name to indicate that it's their turn  
 document.getElementById('playerName').style.color = 'white';  
 document.getElementById('enemyName').style.color = 'yellow';  
 **this**.playerTurn = **false**;  
  
 //waits a second and then enemy takes their turn so it’s not done in an instant  
 setTimeout(**function** () {  
 fight.enemyTurn();  
 }, 1000);  
 }  
},  
  
enemyTurn: **function** () {  
 //only called from the end of the players turn  
 //depending on the enemies intention (determined later) takes different actions  
 **if** (**this**.enemy.intention == 'attack') {  
 //calls attack with false, meaning it’s not the player attacking  
 **this**.attack(**false**)  
 } **else if** (**this**.enemy.intention == 'block') {  
 //increases the enemy's defence permanently by 10% (with a minimum of 1) and then updates the stats  
 **this**.enemy.defence = (Math.floor(**this**.enemy.defence \* 1.1) == **this**.enemy.defence) ?

(**this**.enemy.defence + 1) : Math.round(**this**.enemy.defence \* 1.1);  
 **this**.outputStats();  
 }  
 **this**.playerTurn = **true**;  
 document.getElementById('playerName').style.color = 'yellow';  
 document.getElementById('enemyName').style.color = 'white';  
  
 //determines what the enemy will do next turn, 80% chance of attacking, 20% of blocking  
 **if** (Math.random() > 0.8) {  
 **this**.enemy.intention = 'block';  
 } **else** {  
 **this**.enemy.intention = 'attack';  
 }  
 document.getElementById('enemyIntention').innerHTML = **this**.enemy.intention;  
},  
  
changeEnemyHealth: **function** (damage) {  
 **var** enemyDead = **false**;  
   
 //changes enemy health, if its 0 or below, set the enemy dead flag to true and makes sure health doesn't fall below 0  
 **this**.enemy.hp -= damage;  
 **if** (**this**.enemy.hp <= 0) {  
 enemyDead = **true**;  
 **this**.enemy.hp = 0;  
 }  
   
 //updates HUD  
 document.getElementById("enemyHealthBar").style.width = ((**this**.enemy.hp / **this**.enemy.health) \* 100) + "%";  
 document.getElementById("enemyHealthLabel").innerHTML = **this**.enemy.hp + "/" + **this**.enemy.health + "hp";  
   
 **if** (enemyDead) {  
 **this**.victory();  
 }  
 **return** enemyDead;  
},  
  
genEnemy: **function** (type) {  
 //called as part of the prerequisite to a fight, if not a boss fight  
 **var** enemy = **new** Enemy;   
   
 //scalar is used to adjust enemies difficulty throughout the playthrough  
 //to keep up with the player   
 **var** scalar = (pointsCleared / pointNo);  
 scalar = (scalar < 0.2) ? 0.2 : scalar;  
  
 enemy.name = type;  
 enemy.attack = Math.floor(((Math.random() \* 15) + 30) \* scalar);  
 enemy.defence = Math.floor(((Math.random() \* 10) + 15) \* scalar);  
 enemy.health = Math.floor(((Math.random() \* 100) + 150) \* scalar);  
 enemy.intention = 'attack';  
 enemy.hp = enemy.health;  
 **return** enemy;  
},  
  
victory: **function** () {  
 //only called when the enemy dies, from changeHealth()  
 document.getElementById('enemyHealthBarBG').style.display = 'none';  
 document.getElementById('enemyHealthLabel').style.display = 'none';  
   
 **if** (**this**.interactionNo != "bossFight") {  
 //if the player beat a normal enemy  
 //it will open the vitcory screen and mark interaction as complete   
 mapObj[**this**.interactionNo].complete = **true**;  
 nav.open('victoryScreen', **this**.interactionNo);  
 } **else** {  
 //if the player beat the boss, will output the final victory screen  
 nav.open("output");  
 setOutputDiv(100, "victory");  
 }  
 inBattle = **false**;  
}

### Loot System

The loot system was one of the last major features to be implemented into this project, it turned out to be a lot smaller than intended but it’s still functioning as it needs to. ‘genItem()’ can work independently to ‘genLoot()’ but not vice versa, ‘genLoot()’ uses ‘genItem()’ multiple times. ‘genItem()’ is called from multiple different places, ‘genLoot()’ is the main place it will be called from but it is also used to generate items for the shop in ‘populateShop()’, and for some interactions that generate loot as a result of a certain encounter choice. ‘genLoot()’ is only ever called at the end of an interaction when the player gets some loot.

**function** genItem(inShop, type) {  
 //this is called whenever a random item needs to be generated  
 //this can be for loot, shops or interactions  
 //inShop is an argument that determines whether this item is going in a shop or not  
 //type is used when a specific item type needs generating, normally undefined  
  
 //scalar works the same as for enemy generation  
 //except that there's a modifier for if the item is in a shop  
 **var** scalar = pointsCleared / pointNo;  
 scalar = (scalar < 0.2) ? 0.2 : scalar;  
 scalar = (inShop) ? scalar - 0.1 : scalar;  
  
 **var** item = **new** Item();  
 //cases of type being undefined handled here, a random type is generated  
 **if** (**typeof** type === "undefined") {  
 type = (Math.floor(Math.random() \* 9) + 1);  
 }  
  
 //power is an item stat and is also used to generate the rest of the item stats  
 **var** power = 100 \* scalar;  
 //like the enemy generation, the items have a variant so not every item is the same  
 **var** variant = Math.floor(Math.random() \* (power \* 0.2));  
 **if** (Math.random() >= 0.5) {  
 variant = variant \* -1;  
 }  
  
 //empty item template  
 item.consumable = **false**;  
 item.regen = 0;  
 item.health = 0;  
 item.attack = 0;  
 item.defence = 0;  
 item.charisma = 0;  
 item.intelligence = 0;  
 item.power = 0;  
  
 //the items stats are all generated from the power and the variant calculated above  
 **switch** (type) {  
 **case** 1:  
 item.type = "sword";  
 item.attack = Math.floor(power) + variant;  
 **break**;  
 **case** 2:  
 item.type = "staff";  
 item.intelligence = Math.floor(power + variant);  
 **break**;  
 **case** 3:  
 item.type = "helmet";  
 item.defence = Math.floor(power / 4) + variant;  
 **break**;  
 **case** 4:  
 item.type = "amulet";  
 item.attack = Math.floor(power / 3) + variant;  
 item.charisma = Math.floor(power / 2) + variant;  
 item.intelligence = Math.floor(power / 3) + variant;  
 **break**;  
 **case** 5:  
 item.type = "armour";  
 item.health = Math.floor(power \* 3.5) + variant;  
 item.defence = Math.floor(power / 3) + variant;  
 **break**;  
 **case** 6:  
 item.type = "leggings";  
 item.health = Math.floor(power \* 2.5) + variant;  
 item.defence = Math.floor(power / 4) + variant;  
 **break**;  
 **case** 7:  
 item.type = "boots";  
 item.defence = Math.floor(power / 4) + variant;  
 **break**;  
 **case** 8:  
 item.type = "potion";  
 item.consumable = **true**;  
 item.regen = 20;  
 **break**;  
 **case** 9:  
 item.type = "food";  
 item.consumable = **true**;  
 item.regen = 10;  
 }  
 item.power = power;  
  
 //cost is also randomised slightly so that it’s not always obvious which item is best value for money  
 item.cost = (type < 8) ? (2 \* power) + variant : 2 \* power;  
 **return** item;  
}  
  
**function** genLoot() {  
 //called whenever the player is given the opportunity to gather loot  
 //after certain interactions or after battles  
 **var** itemNoSel = Math.random(), itemNo, output = "", items = [], div;  
  
 //determines how many items will be generated  
 **if** (itemNoSel < 0.6) {  
 itemNo = 1  
 } **else if** (itemNoSel < 0.9) {  
 itemNo = 2  
 } **else** {  
 itemNo = 3;  
 }  
  
 **for** (**var** i = 0; i < itemNo; i++) {  
 output = "";  
 //makes it so a consumable item isn't generated, only equipment  
 **do** {  
 items[i] = genItem(**false**);  
 } **while** (items[i].consumable === **true**);  
  
 //creates the div's for the items generated  
 div = document.createElement('div');  
 div.className = 'lootItem';  
 div.id = 'lootItem' + i;  
 div.setAttribute('itemId', "" + i);  
 div.style.backgroundImage = "url(img/" + items[i].type + ".png)";  
 **for** (**var** y **in** items[i]) {  
 **if** (y != 'consumable' && y != 'cost' && y != 'inShop' && items[i][y] != "") {  
 output = output + y + ': ' + items[i][y] + '<br>';  
 }  
 }  
  
 div.onclick = **function** (evt) {  
 **var** itemId = parseInt(evt.target.getAttribute('itemId'));  
 player.inventory.push(items[itemId]);  
 document.getElementById('loot').removeChild(**this**);  
 };  
 div.style.top = (i + 1) \* 120 + "px";  
 document.getElementById('loot').appendChild(div);  
 document.getElementById('lootItem' + i).innerHTML = output;  
 }  
 //creates the div for the gold generated at the end of the fight  
 div = document.createElement('div');  
 div.className = 'lootItem gold';  
 div.id = 'lootItem' + i;  
 div.setAttribute('itemId', "" + i);  
 div.style.backgroundImage = "url(img/gold.png)";  
 div.style.top = (i + 1) \* 120 + "px";  
 div.style.lineHeight = "100px";  
 **var** gold = Math.floor(Math.random()\*20) + 10;  
 div.innerHTML = gold + " Gold";  
 div.setAttribute("gold", "" + gold);  
 div.onclick = **function** (e) {  
 player.changeGold(parseInt(e.target.getAttribute("gold")));  
 document.getElementById('loot').removeChild(**this**);  
 };  
 document.getElementById('loot').appendChild(div);  
}

# Implementation

Here a copy of both pseudocode from my notebook and the actual implementation of the same function will be shown for comparison:

## buyItem()

Pseudocode

buyItem  
if (enough gold)(shopItems[pos] != “empty”)  
 shopItems[pos].inShop = false  
 player.inv.push(shopItems[pos])  
 player.gold -= shopItems[pos].cost  
 doc.getId(shopItem…).bgimage = none  
 .innerHTML = “sold”  
 doc.getId(shopPrice).innerHTML = “Sold out”  
 doc.getId(announcement).inner = “Purchased”  
  
elif (shopItem[pos] == “empty”)  
 announcement.inner = “This item is sold out”  
else   
 announcement.inner = “You don’t have the gold to purchase this”

Actual Implementation

**function** buyItem(shop, pos, y, x) {  
 **var** item = eval(shop.id + 'Items[' + pos + ']');  
 **var** price = item.cost;  
 **var** sold = player.changeGold(-price);  
  
 document.getElementById('shopAnnouncement').style.display = 'block';  
 **if** (sold && item != "") {  
 player.inventory.push(item);  
 **switch** (shop.id) {  
 **case** 'merchant':  
 merchantItems[pos] = "";  
 **break**;  
 **case** 'aMerchant':  
 aMerchantItems[pos] = "";  
 **break**;  
 **case** 'smithy':  
 smithyItems[pos] = "";  
 **break**;  
 }  
  
 document.getElementById(shop.id + 'Item' + y + ',' + x)

.removeChild(document.getElementById

(shop.id + 'tooltiptext' + y + ',' + x));  
 document.getElementById(shop.id + 'Item' + y + ',' + x)

.style.backgroundImage = 'url(img/sold.png)';  
 document.getElementById(shop.id + 'Price' + y + ',' + x)

.innerHTML = 'Sold out!';  
 document.getElementById('shopAnnouncement').innerHTML = 'Item purchased';  
  
 } **else if** (item == "") {  
 document.getElementById('shopAnnouncement').innerHTML = 'This item is sold out';  
 } **else** {  
 document.getElementById('shopAnnouncement')

.innerHTML = 'You dont have the gold to purchase this';  
 }  
 setTimeout(**function** () {  
 document.getElementById('shopAnnouncement').innerHTML = "";  
 }, 2000);  
  
}

Break lines have been used in this code sample to make cade more readable   
in this document, if copying into own text editor, will have to delete them

While there are some clear deviations from the pseudocode, these were mainly to do with the later implementation of objects into the code, the other one being using a case statement to wipe the item from the shop. This was because the original plan for shops and how they work and are set out is different to the current implementation of them.

## genItem()

Pseudocode

genItem()  
pointNo – global variable  
pointsCleared – global variable  
var percentCleared = (cleaed/pointNo)  
var item = new Item()  
var type = Math.floor(Math.random()\*9)+1  
switch (type)  
1  
-  
9  
  
**Item Example**1 type – sword  
 consumable – false  
 instantHealth – 0  
 health – 0  
 attack – rand 3 – 5  
 defence – 0  
 charisma – 0  
 intelligence – 0

**Item types**sword – 1  
staff – 2  
helmet – 3   
amulet – 4  
armour – 5  
leggings – 6  
boots – 7  
potion – 8  
food – 9

Actual Implementation

**var** scalar = pointsCleared / pointNo;  
scalar = (scalar < 0.2) ? 0.2 : scalar;  
scalar = (inShop) ? scalar - 0.1 : scalar;  
**var** item = **new** Item();  
**if** (**typeof** type === "undefined") {  
 type = (Math.floor(Math.random() \* 9) + 1);  
}  
**var** power = 100 \* scalar;  
**var** variant = Math.floor(Math.random() \* (power \* 0.2));  
**if** (Math.random() >= 0.5) {  
 variant = variant \* -1;  
}  
item.consumable = **false**;  
item.regen = 0;  
item.health = 0;  
item.attack = 0;  
item.defense = 0;  
item.charisma = 0;  
item.intelligence = 0;  
item.power = 0;  
**switch** (type) {  
 **case** 1:  
 item.type = "sword";  
 item.attack = Math.floor(power) + variant;  
 **break**;  
 **case** 2:  
 item.type = "staff";  
 item.intelligence = Math.floor(power + variant);  
 **break**;  
 **case** 3:  
 item.type = "helmet";  
 item.defense = Math.floor(power / 4) + variant;  
 **break**;  
 **case** 4:  
 item.type = "amulet";  
 item.attack = Math.floor(power / 3) + variant;  
 item.charisma = Math.floor(power / 2) + variant;  
 item.intelligence = Math.floor(power / 3) + variant;  
 **break**;  
 **case** 5:  
 item.type = "armour";  
 item.health = Math.floor(power \* 3.5) + variant;  
 item.defense = Math.floor(power / 3) + variant;  
 **break**;  
 **case** 6:  
 item.type = "leggings";  
 item.health = Math.floor(power \* 2.5) + variant;  
 item.defense = Math.floor(power / 4) + variant;  
 **break**;  
 **case** 7:  
 item.type = "boots";  
 item.defense = Math.floor(power / 4) + variant;  
 **break**;  
 **case** 8:  
 item.type = "potion";  
 item.consumable = **true**;  
 item.regen = 20;  
 **break**;  
 **case** 9:  
 item.type = "food";  
 item.consumable = **true**;  
 item.regen = 10;  
}  
item.power = power;  
item.cost = (type < 8) ? (2 \* power) + variant : 2 \* power;  
**return** item;

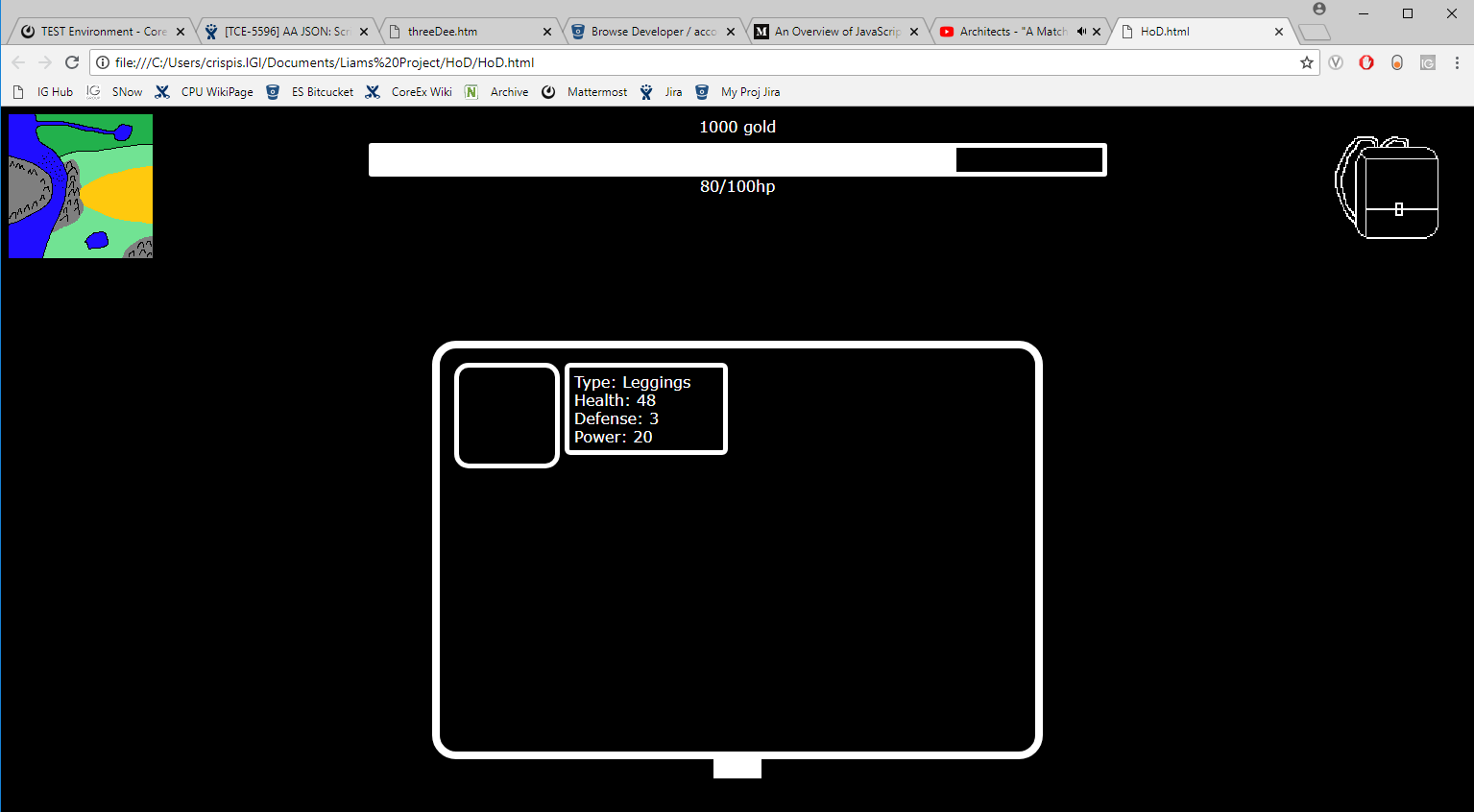
Like in the previous example, there is a good amount of variation from the original plan to the actual final implementation.

A big change to how the function is written was to do with the realisation that initialising all the objects fields to 0 would save a lot of repeating code later on. This reduced the size of my case statement dramatically and also made it much more readable. Another change that was made was the ability to be able to choose what item type that would be generated if and when the need arises, this is done by introducing a new argument into the function. The argument is read, if nothing was passed in to that argument, it will be undefined and an item type will be chosen at random. This was originally going to have be separate function but with this argument there is no need for it.

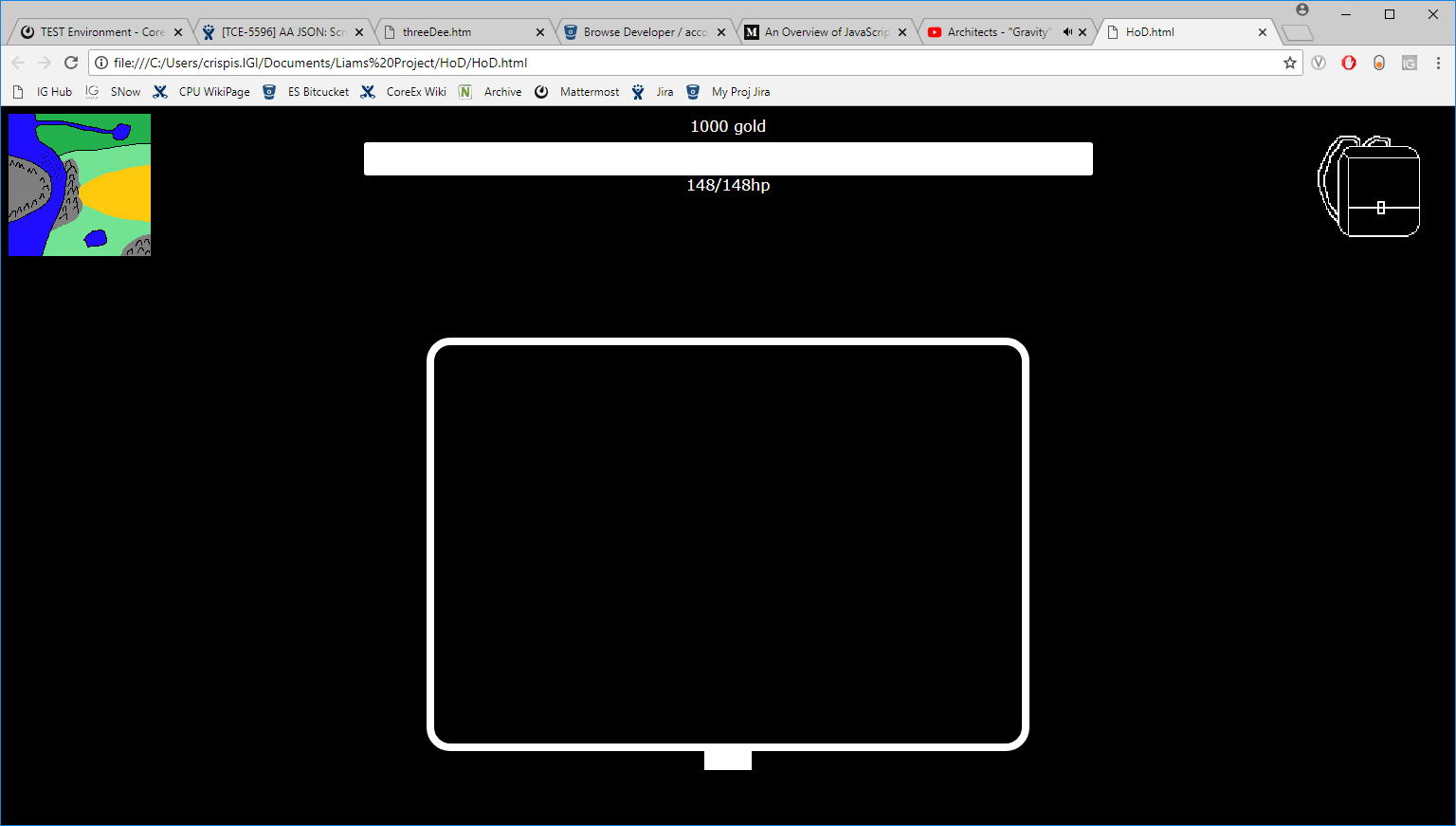
The biggest change to this functions actual functionality was the implementation of a variant. This variable allows a deviation in the cost and power of equipment generated, rather than all being the same like they previously were. This variant takes into account the amount of points cleared so far and weather the item is in the shop or not, shop items are less powerful as to make them a last resort when improving equipment, if you are very unlucky with a certain item not dropping yet for example. The variant was also given a minimum value of 0.2 as to not make values 0 or too close to it.

# Testing

The best way to test a game is play testing, the developer played through the game as much as is possible with this current version to try and find any bugs or features not working as intended.

The first bug encountered was to do with equipping loot that was obtained after a battle:

Here you can see that health has been lost after being damaged in a fight and some loot has been obtained from the battle. There is one immediate problem, the div for the loot item doesn’t have a background picture like it is supposed to. The next problem arises when the item is equipped



The tool tip for the item says that it will give the player 48 health, meaning maximum health should go up to 148 and current health should increase by 48 as well. Clearly that hasn’t happened. Maximum health has increased by 48 correctly but my health also restored to full which wasn’t the intended outcome.

The problem for the health was in updateStats() and changeHealth(), both methods in the player object. The updateStats() function was setting the players current health to the base health (100) + the players total bonus health, in this case: 48. This meant that whenever updateStats() was called, the players health was set to the maximum that it could be.

updateStats: **function** () {  
 player.health.bonus = player.equipped[0].health + player.equipped[2].health

+ player.equipped[3].health + player.equipped[4].health;  
 player.attack.bonus = player.equipped[1].attack + player.equipped[5].attack;  
 player.defense.bonus = player.equipped[0].defense + player.equipped[2].defense

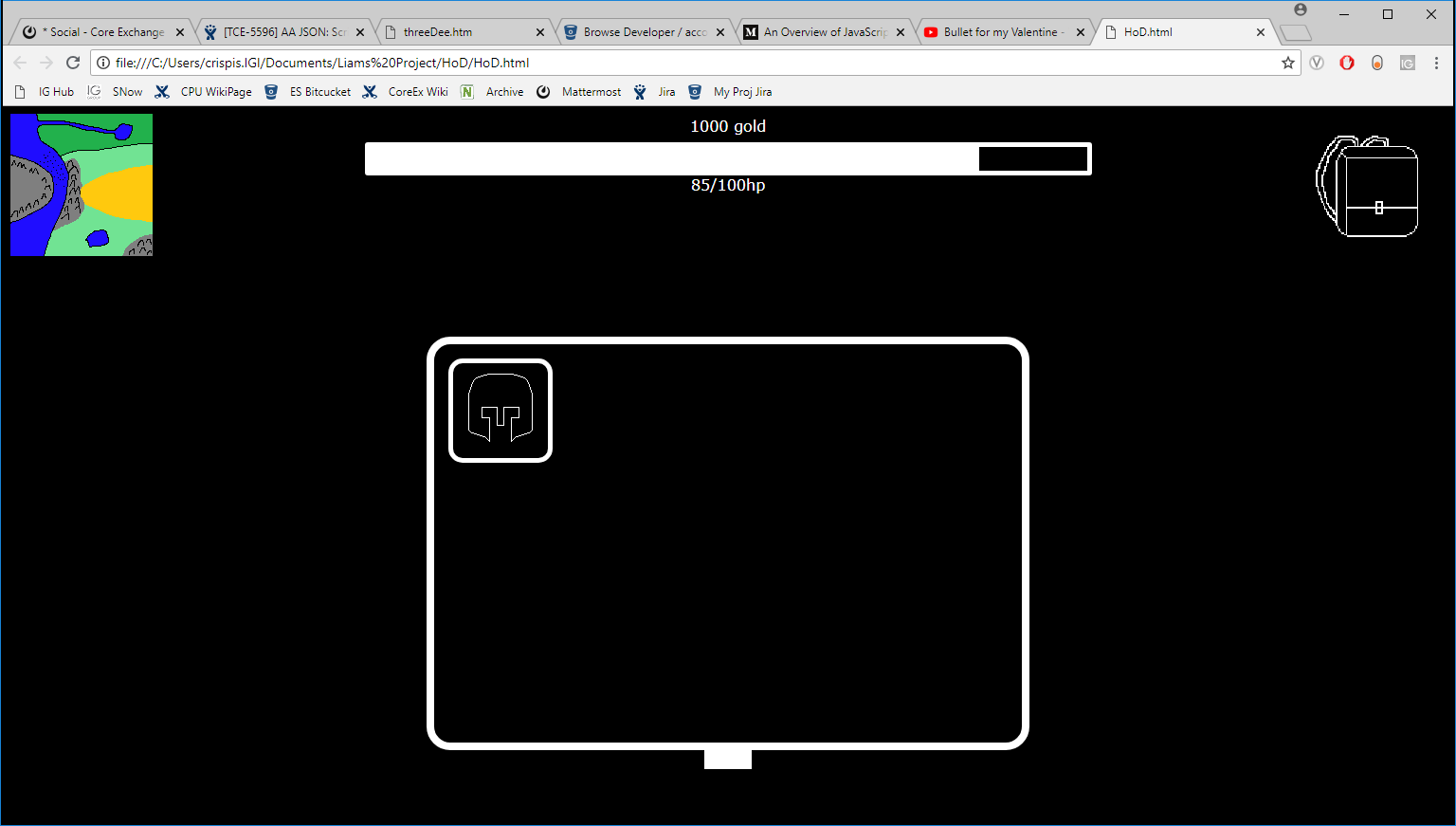
+ player.equipped[3].defense + player.equipped[4].defense;  
 player.charisma.bonus = player.equipped[1].charisma;  
 player.intelligence.bonus = player.equipped[1].intelligence

+ player.equipped[5].intelligence;  
  
 player.health.current = player.health.base + player.health.bonus;   
 player.attack.current = player.attack.base + player.attack.bonus;  
 player.defense.current = player.defense.base + player.defense.bonus;  
 player.charisma.current = player.charisma.base + player.charisma.bonus;  
 player.intelligence.current = player.intelligence.base + player.intelligence.bonus;  
  
 **this**.changeHealth(0);  
}

To fix this bug, the line setting the players current health in the second part of the function shown above, was removed. This meant that the players current health was only ever modified within the changeHealth() function, as intended.

As for the error regarding the fact that the div had no background image. The error shown above, was being logged for every instance of a broken background image, the name differing respectively for the image that is trying to be shown.

The fix for this only required a small amount of change. The project structure had previously been changed so that the images were contained within their own folder named ‘img’. The error arouse due to the fact that not all code instances of referencing an image had been changed to include the prefix ‘img/’, therefore the code was being pointed to the wrong location and not seeing the image.

The fix only required to search for all instances of ‘url(’ and to check where the ‘img/’ prefix had been left out, and then add it in. Proof for background images working in thumbnails working shown below.

# Refactoring

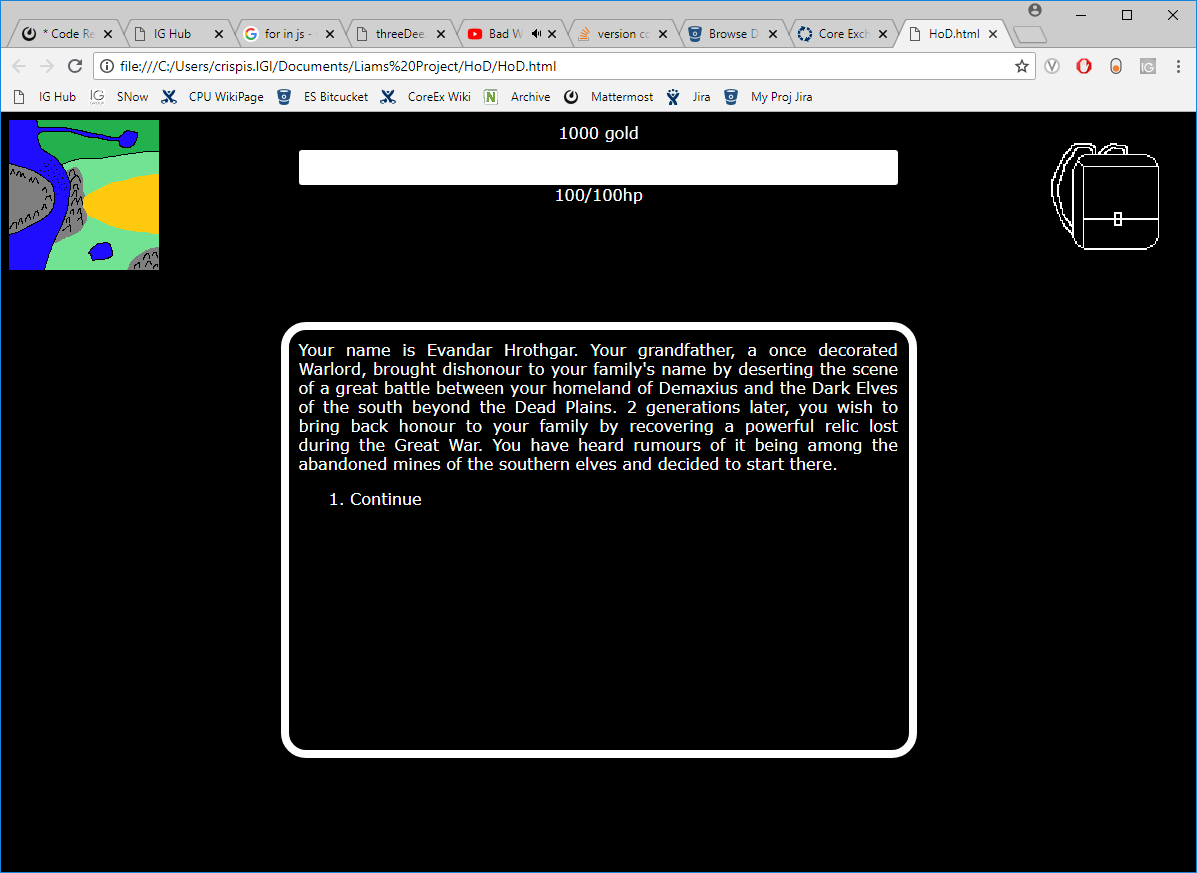
As mentioned previously, there was a good amount of refactoring during the lifetime of this project, the majority being to how interacts are handled. There was also other refactoring done to include objects elsewhere in the code. The biggest object (aside from the interactions, talked about later) is the player object but there are also fight and navigation objects.

How the interactions were handled changed multiple times over the course of this project. Originally, the interactions were all stored in a large switch statement. This had many disadvantages, including readability, customisability and speed. Using a switch statement meant that if changes needed to be made to the structure of the statement to include more options or to change the way a certain interaction works, a lot of work would have to be done to accommodate that. Where speed isn’t a major factor in this kind of project as there isn’t a game loop happening 100 times a second for example, it is still best practise to make the game run as fast as possible. Switch statements will check each option up to and including the one that is selected, meaning that if the option selected is the very last option, it will check each option before reaching the intended output, wasting a lot of time at the same time.

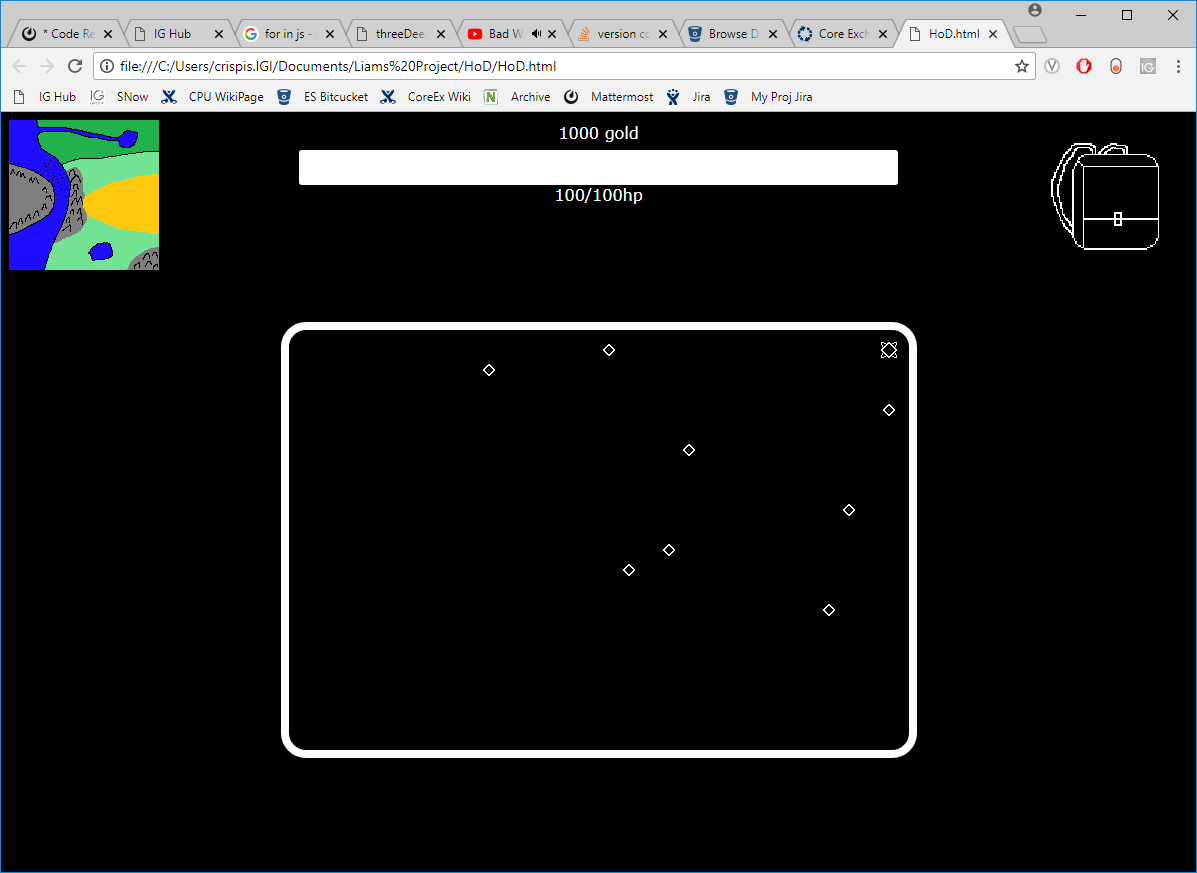
Another solution to the problem of handling interactions was to use an object to store each interaction with each frame of an interaction being another object nested within it. Originally, the onclick function of an option was a function call with all the arguments being passed in via the function call. A few problems reared their heads when using this method including the handling of undefined arguments. Not all interactions would need to use all arguments so they wouldn’t be defined which led to a lot of messy code handling that when it happened. One way to get around this was using attributes. Instead of passing in arguments via the function call, a preliminary function was written which extracted the arguments from the divs attributes which were set from the previous function call. This made handling of arguments a lot easier, first of all, if an argument wasn’t used, it wasn’t accessed and didn’t cause a problem and also made adding extra attributes/arguments a lot easier. The rest of the object wouldn’t have to be touched, just adding in the extra attributes where needed.

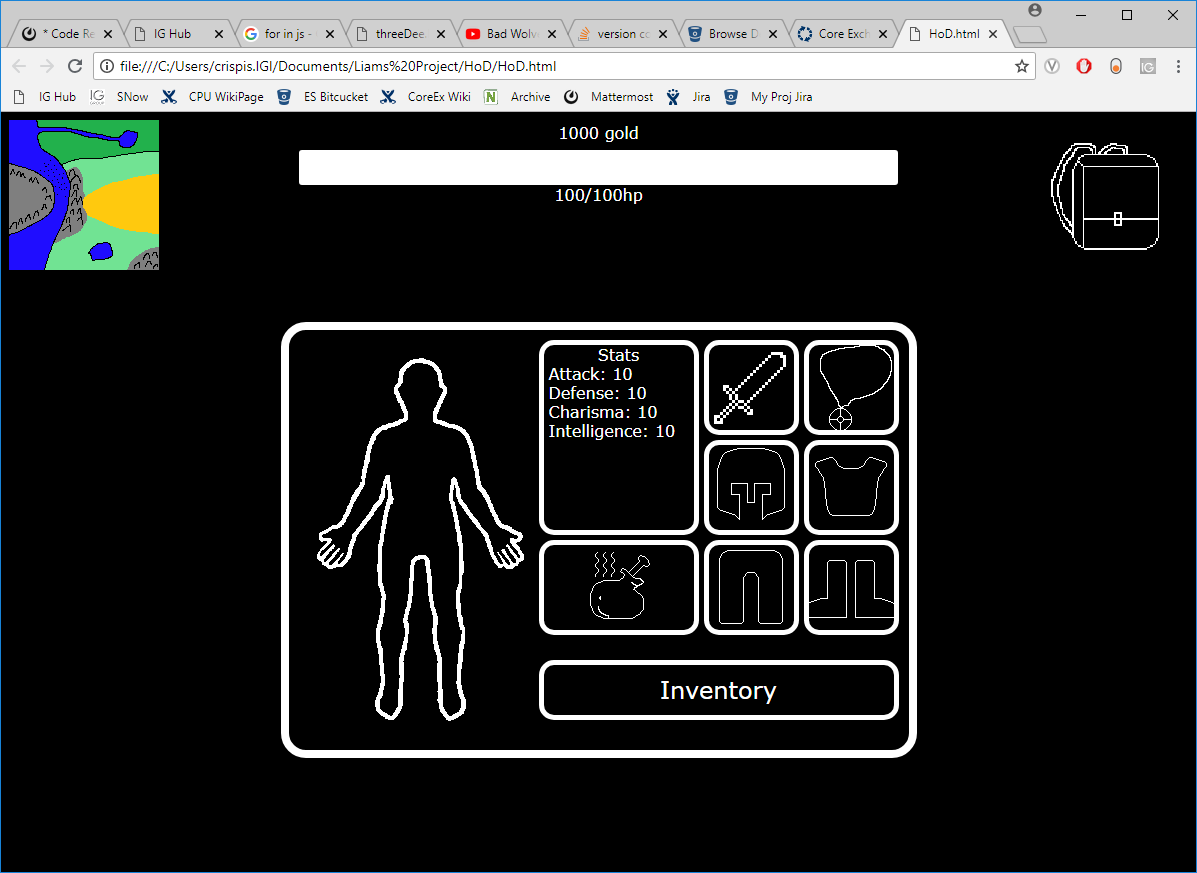
# Screen Shots

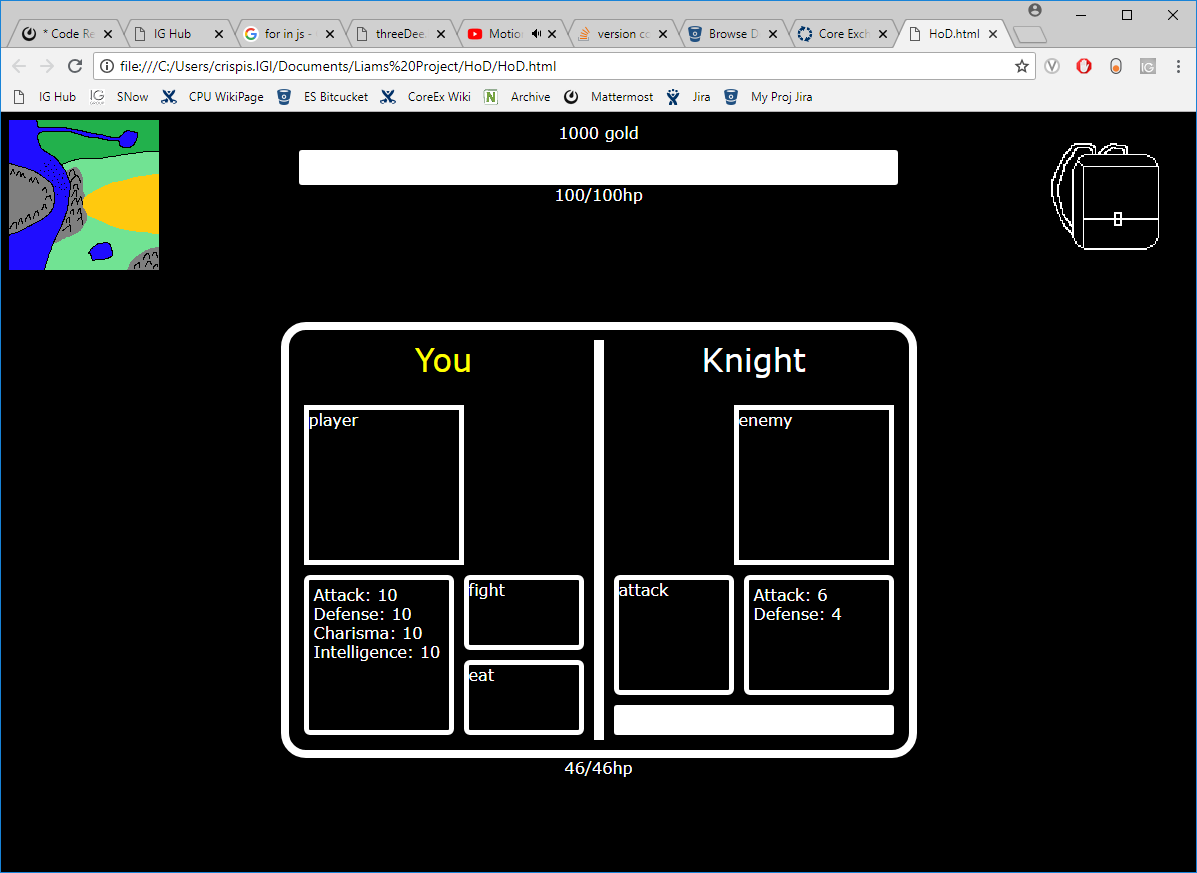
Opening Screen



Map Screen



Inventory Screen

Fight Screen

# Conclusion

In conclusion, where the project could have gone much better, there were still aspects of it that went well. Progress was initially quick even if the early code was rewritten and refactored at a later point. Quick progress was due to the fact that the developer had no other commitments at the time and could focus completely on this project. There was a few hiccups in the process, mainly the creation and then refactoring of the interactions object. There were about 3 or 4 different iterations of the object which was spoken about earlier.

There were plenty of concepts and techniques that were learnt about during this project, the main one is the use of object lookup. This was used in tandem with the later versions of the previously mentioned interactions object. Again, with my interactions object, there were a few other concepts like event listeners and other bits related to it like using target and function pointers a lot of which I hadn’t touched before.

Although this project is closer to being finished than was expected at this point, it is ultimately unfinished. The main bit that was unfinished was the infamous interactions object. However there were other smaller bits and finishing touches that I would have liked to make. There are a few undrawn assets and unbalanced parts of the game that make the game feel unfinished.

The interactions object was unfinished in more than one way, it still needed to be populated completely, it only had about ¼ of the amount of interactions that were originally planned. The creation of scenarios that were both not too farfetched and not too dull as to bore the player, was a much more time consuming task than originally planned for therefore less time than needed went into them.

There were features that were in the original plan but never made it into the final implementation. The main one being a save and load game function that would use local storage and JSON to save the games state so you can carry on with a play through. This would only be accessible outside of an interaction, meaning that a person couldn’t just save and keep reloading for the optimal outcome to an interaction.

There were other bits as well, the previously mentioned ability to navigate with the keyboard using event handlers would be a nice to have if it was planned from the start rather than knowing how to do it half way through the project. It’s not particularly complex but I think it would be done much easier if planned from day 1.

Character creation and an experience and levelling system was a feature that was only thought about if the project was completed ahead of time. At the start of the game, you would be prompted to create the character you would keep throughout the playthrough. There wouldn’t be anything cosmetic about it, you would only be able to edit the characters starting stats. You would be given a base character and some points that you could assign to your characters stats as you see fit. You would earn experience from killing enemies of other special outcomes from interactions and when you level up, you would be able to increase one of your stats like in the character creation screen.

In the same vein, leaderboard for keeping track of player’s scores would been implemented if the game was finished well before time. The player would earn points from completing interactions and killing enemies, points earned would depend on the enemy’s difficulty, more for harder enemies and less for easier ones.

As for things that would have been done differently if the project was restarted now, the biggest part is the combat engine, or lack thereof more accurately. If this project would be rewritten without time constraints, the fight engine as a whole would have a complete overhaul. Combat would likely take more inspiration from other RPG’s like the Pokémon series where fighting was ability/move based. There would be more distinct types of weapon, rather than just sword and staff, and these would have different types of basic attack which would do different types of damage (a mace would do blunt damage and a spear would do piercing damage for examples). More powerful weapons might also have abilities that give you other unique moves or just buff certain stats in different situations. This differentiation in weapons would add an element of strategy in which weapons you would equip, a weapon might have higher base stats than another one but the other might have a powerful ability that you want. This also introduces an aspect of personal preference, one person could have different opinions on a certain weapon type or ability that would cause them to use a different weapon than you, rather than just only using the weapon with the highest stats. At the moment, combat is very linear, you either attack or heal up by eating, there isn’t anything else you can do within a combat and this makes the game both extremely boring and extremely luck based.

There was a plan to implement different types of damage along with the two different types of weapon in the game at the moment, with swords doing physical damage and staffs doing magic damage. Enemies would also have two types of resistances, armour: for physical attacks, and magic resistance: for magic attacks. The player would then have an option within a battle to switch weapons, consuming a game turn but then being able to use a more efficient weapon for the remaining duration of the fight. This would at least add a small amount of strategy to a battle which is currently non-existent.

Another change would be a redo of the loot system. Where the current loot system is good and it fits the needs of the current implementation of the game, if the above changes to the fight engine were to be made, major changes to the loot system would be required to accommodate that. A whole new stat would have to be included on armour, magic resist and there would be a whole array of new weapons being implemented to make the combat much more expansive.

Another plan that was for the current loot system but never saw the light of day was categorisation for the loot generated, much like the *Borderlands* and the *Diablo* series. These games both use a colour coded loot system which was also the plan for this game, if a weapons power was over a certain threshold, it would be classified as uncommon (green), then rare (blue), then very rare (purple) and then legendary (orange). In these games, orange weapons were often specific named drop that were the same every time (base stats differed dependant on level but the special effect was the same every time the weapon with that name was generated), there was also the plan to do this but, again, time constraints didn’t allow it.