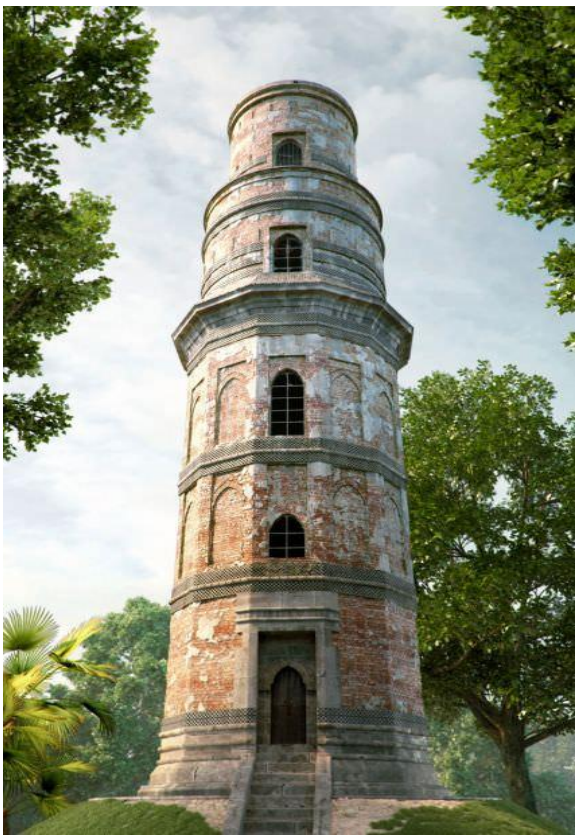


RECLAMATION

ABSTRACT

Description of the design and creation of Reclamation: A 2D Zombie Shooter Game.

JACK ECUYER

Submitted as part of the OCR Computer Science coursework requirements:H446/03

Candidate No: 1056
Centre number: 16444

Contents

Analysis

Introduction

For my project I intend to make a web based multiplayer 2d zombie shooter game for the browser, called Reclamation.

In Reclamation, the objective is to progress through horizontally scrolling levels across a large map, reaching towers at points each level to reclaim by guarding the tower from oncoming waves of zombies. The towers will have a certain amount of health, and can be damaged by zombies if they come into reach. Killing zombies will reward the player with cash, which can be used to purchase new items from a market once a level has been completed such as weapons, armour, ammo and grenades.

As the player progresses through the levels, the difficulty will increase by spawning new, more powerful types of zombies in larger numbers as well as tower health being reduced. The player will earn more experience points (XP) the harder the level is, and levelling up your player will unlock a larger variety of upgrades available for purchase in the market.

Core elements of the game that will make it suitable for a computational approach are: the use of object oriented programming (OOP) to design different types of zombies with different appearances, abilities and attributes; the algorithmic calculations behind player progression including level and money; database design to enable profiles/progress and level to be stored as well as compared (high score tables); client/server side architecture design to enable the game to become multiplayer (co-op mode).

The entire project will be managed in stages by a method of problem decomposition to identify modules that can be programmed independently from one another e.g. At this point it is clear that getting the single player base game of level development, zombie spawning and shooting is essential for a first stage, before progressing onto an item market with money and XP, then networking features such as server storage of player profiles and developing an online co-operative mode.

Stakeholders

The primary stakeholder and originator of the idea for this project is myself, I have always been a fan of competitive shooter games and wanted to choose this project as a way to create a new game that I will be able to compete with friends in to achieve the highest points on a leaderboard but also play with for fun. In addition, I have chosen this project to advance my ability in compiling all different areas of my technical programming skills into one game and specifically to improve my network programming skills.

I expect to use a range of friends from classmates to friends that I play games with online to be my secondary stakeholders. They don't have a great stake in the project, however they would like the project to be successful so that they can play with each other and compete for

the highest scores on the leaderboards. My friends will be able to test early versions of the game, reporting any bugs that they find which will give me more time to work on programming and make the development more manageable.

Finally, once the game is completed and released to the public, I hope that the competitive nature will incentivise players to offer me feedback on improvements that could be made with gameplay as well as offering ideas for new additions.to the game. Also, it is possible however hopefully unlikely that players discover bugs in gameplay that they will report and I can fix.

Research

There are 3 main threads to my research:

1. Analyse similar games with 2d shooting mechanics and level based gameplay
2. Discover features from other games that I would like to implement into my project
3. Technical research looking at the essential programming techniques I'm going to use

Commando 3

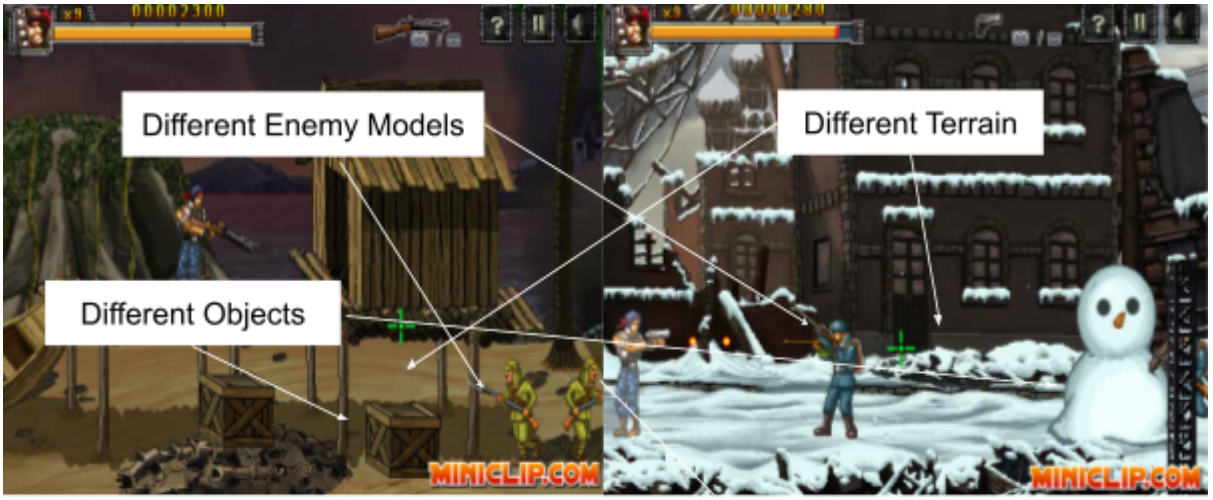
One game that has many similarities in format and design to the project I intend on creating is **Commando 3**, a game playable on http://miniclip.com/. Therefore, I used this game as a tool to research common usability features and mechanics which I will also be developing into my game.

Level Generation

There are two possible methods for approaching level generation within my game which include:

- Random level generation using generation algorithms
- Predetermined level generation using a hard-coded approach

However, the method that I believe suits my specific goals the best is a predetermined approach to level development. This is because I would like the theme of the game to evolve as the player progresses through the levels, and this will be easier achieved if I have the ability to manually control the theme of each level. Furthermore, my game has less of a focus on level variety and scale (with the main objective being taking over towers) therefore I don't believe that the hard-coding of each level will take up an overly large amount of time that will hinder other areas of game development.



As seen from the screenshots above, the colours, terrain, objects and even enemy models included within different levels of commando 3 differ drastically, which I believe gives the game a more interesting feel that incentivises the player to continue playing, rather than a continuous theme that could get repetitive and bore the player.

I specifically like the use of different enemy models, and this is a feature that I am going to implement into my game to give the zombies more variety and make it easier for the player to determine the difficulty level of the zombies.

Player Movement

Player movement is one of the key features in the game that the player must control, and in my game, there will be four types of movement that the player will be able to do which include:

- Moving Left
- Moving Right
- Jumping
- Crouching

To control these movements, the player will use the traditional movement keys that the majority of games on the market use which are W (jump) A (left) S (crouch) and D (right) because I think this will increase the appeal of the game to a larger player base, as gamers can use their muscle memory from other games and incorporate it into Reclamation.

Commando 3 incorporates these same movement mechanics well, because the mechanics can be used to strafe and dodge incoming attacks or projectiles.

The ability to dodge these attacks is a mechanic which is great for a game like mine, specifically because it adds to the skill based nature that I am aiming for, and consequently makes the game more competitive.

Heads up Display (HUD)

The heads up display of a game is intended to offer the player an insight into their statistics whilst they are playing. These statistics can include their current health, ammo and money as well as any other key information which could help them play.

The HUD implemented into Commando 3 includes the players health, score, current weapon, ammo and a pause button which are all positioned at the top of the screen.

I am also going to position the HUD of my game at the top of the screen, because the gameplay is going to be on the lower half of the screen, And I don't want the HUD to block the vision of any gameplay.

I would like to implement a similar healthbar to the one used in commando 3: where the bar depletes or increases depending on how much health the player has, however I am going to use a different colour scheme of green and red as well as include an exact value for the players health overlaying the bar to be even more specific. In addition, in the co-operative gamemode of my game I intend to also display the health of teammates that are playing alongside the player.

I also really like how Commando 3 displays an image of the current weapons the player is using, and overlays the image with the current ammo next to the max ammo. This approach is simple yet effective because the player knows without reading any text which weapon is equipped and I would like to use this same format in my game.

Shooting Mechanics

The shooting mechanics are going to be relatively simple for my game. The player will have a crosshair positioned wherever their mouse is on the screen, and when the player shoots (left click) the weapon will fire. The weapon and arm of the player will point in the direction of the crosshair.

Commando 3 follows this same method of aiming and shooting, however one feature that I am going to implement is the changing of crosshair appearance depending on the weapon

the player has equipped. I believe this will add some more variety to the game in general however will also make it easier for the player to identify which weapon they are using.

I am leaving the actual shooting mechanics for the game relatively simple, and instead adding the complexity within the weapon variety which will affect the bullet speed, ammo type and damage.

Other Games

There are also some other games which are not similar to the plan of my game such as **Fortnite** and **League of Legends**, however I would like to incorporate some of the features from them into my game.

Medal Punch Card

One feature that I would like to include in my project is similar to the medal punch card from **Fortnite: Battle Royale**. This feature enables the player to gain XP everyday for actions that they complete whilst in game e.g play for one hour to be rewarded with 10,000XP.

However, unlike Fortnite, I would like the medal punchcard to be available for each level that the player attempts, but have one specific challenge that can only be completed once daily. I would like to take this approach because I believe that it rewards players who play the game for a longer period of time with a greater amount of XP. Some of the challenges I would like for the punchcard to include are:

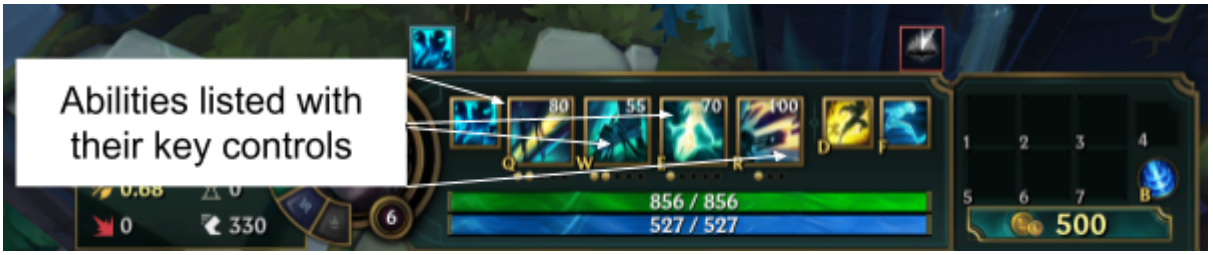
- Playtime (10 mins, 30 mins, 1hr, 2hr, etc.)
- Speedrun (e.g complete this level in under 2 minutes)
- Kills
- Weapon (e.g complete this level using only a pistol)

I really like the presentation of the medal punch card used within Fortnite, and how it is shown how much XP can be achieved through incomplete challenges.

After a match has been completed, a count is displayed at the bottom of the screen which displays statistics for the XP gained within the match, and where the XP has come from. I would like to include this counter in my game after a level has been completed, because it would not block vision of any gameplay, and would inform the player on how well they performed in the select level.

Abilities

A feature from **League of Legends** that I would like to include in Reclamation are abilities. Abilities can be used after a certain amount of time and affect the gameplay in a variety of ways. Some abilities enable the player to get a health or damage boost, whereas other abilities damage enemies instantly with splash damage.



As seen in the picture above, the player has the option to press different keys to activate these abilities (Q,W,E & D), and the abilities vary depending on which hero they are using.

Unlike League of Legends, I would like to include only one special ability for the players in my game, with a large cooldown. However, the player should be able to swap this ability with another by purchasing others in the in-game market.

The options for abilities that can be purchased from the market are going to be focused on different areas that could help with co-operative gameplay. For example, one player could choose a healing ability to support their teammates, another player could choose a damage buff ability, and the other player could use a high damage ability, this way the abilities compliment each other.

Technical Research

Client Server Communication

In order for my game to be multiplayer, a large amount of client server communication is necessary, and therefore researching how the client server communication is going to work is very important.

My project is going to be developed for the browser, therefore the main programming language that I am going to use is JavaScript because of its versatility in all areas of development, as well as being the language that I have the most experience programming in. In order to set up client server communication with JavaScript, I have decided on using a JavaScript Runtime called NodeJS (https://nodejs.org/).

NodeJS essentially enables code to be executed outside of the browser, by creating a server that can execute code instead. This means that the user will be unable to modify anything that is executed on the server side, and therefore will not be able to 'cheat' and change values that could give them an unfair advantage in the game.

Furthermore. NodeJS will be able to receive data from any clients that are connected and relay this data to other clients whenever necessary e.g when a client moves their player, this movement will be displayed on the other client's game also.

As well as having a server that can execute code remotely, NodeJS is going to require further support from another library to simplify the communication between client and server. The library that will assist this communication is called Socket.io (https://socket.io/).

Socket.io enables the client and server to communicate via events e.g the server can listen for an event and when the client sends this event the server can respond. Furthermore, socket.io enables broadcasting where the server can emit the same data to several clients at once. This will be useful for when a game has more than two players in it. Here is an example of event handling within socket.io:

```
//Listen 'connection' event, which is automatically sent by the web
client
io.on('connection', function(socket){
 //Listen 'chat message' event, which is sent by the web client while
sending request
 socket.on('chat message', function(msg){
   //Emit event to all connected users, The second parameter should be
the message
   io.emit('chat message', msg)
 })
})
```

The ‘io’ is a global identifier that includes all clients connected to the server, and the ‘on’ is the function that waits for a certain event to occur (the event type is defined in the parameters of the function). In the image above, the event is named ‘connection’ a default event that occurs when a client connects to the server.

The ‘socket’ refers to the specific client which has communicated the event, and this enables the server to differ between each client because variables can be stored within the ‘socket’ object e.g I would give each ‘socket’ a ‘socket.name’ and add these sockets into an array of server connections.

Finally, in the example above the server waits for a ‘chat message’ event to be received from a specific client, and when received the server relays the data from the event to every client connected using the io once again.
Using this research, I believe the easiest method of organising the communication within my game is to create many different events for the actions of all the players. Some of these events will include movement, shooting, and ability usage. Once the event is relayed to the server, it will be emitted to all the clients within the game.

Prototyping

V0.1 Player Movement

The first thing that I would like to implement into a prototype of the game is player movement for the player that the user is controlling.

I will heavily be utilising the p5 js library (https://p5js.org/) throughout the development of my project, because it simplifies the graphical programming inside of a browser and therefore will enable me to make the game more visually pleasing.

To implement movement mechanics into my prototype (left, right and jump), I created a class named 'Player' that enables the game to create an object which stores the player's data such as their position (x,y) on the screen, and contains player specific functions.

The programming for player movement left and right was relatively simple. I used the keyIsDown() p5 function to detect whenever a player has pressed a key that causes movement, in this case it was 'a' for left and 'd' for right. After detecting the player input, a function is called from inside the player object that increases or decreases the object's x value depending on the direction the user wants to move.

However, the programming for movement became harder once I had to program a jump() function for the player. This part was more difficult because the jump of the player must include an animation, whereas the movement left or right is just modifying an x value.

My first attempt at implementing the jump() function I tried to use the built in p5 timer function millis() which stores the number of milliseconds the program has been running for, to animate the jump over several seconds.

```
jump(){
  //stores the current time of the program
  let startTime = millis();

  //loops until the time of the program is 200 milliseconds further
than the startTime
  while( (millis() - startTime) < 200) {
    this.y -= 1; // increases the y value of the player by 1 each
iteration
  }
}
```

Unfortunately, this method was unsuccessful because it did not enable me to manage how high the player was going or the speed in which the y value was increasing (the y value was changing each time the while loop was iterating, and the speed of the loop cannot be

modified). When tested, the player disappeared off the screen instantly on the press of the jump button.

To fix this problem, I realised I had to sync the player jumping animation with the p5 draw() loop timing, much the same as the movement to the left and right was synced. To do this, I added a different internal function into the player object called update().

Within the update() function, there are several if statements that check:

- Whether or not the player has reached the maximum height of their jump and if so stop the jumping animation
- whether or not the player is in the process of jumping and if they are, continue to change the y value
- If the player has completed their jump, begin to simulate the effect of gravity (changing back of the y value until the player hits the floor).

```
update(){

   //checks to make sure that the player jump is not exceeding the max
jump limit
   if(this.y <= floorHeight-200){
     this.jumping = false;
   }

   //decreases the player y value if the player is jumping
   else if(this.jumping == true){
     this.y -= this.speed;
   }

   //creates the effect of gravity if the player is not still jumping
   if(this.jumping == false && this.y < floorHeight-100){
     this.y += this.speed;
   }
 }
```

This attempt was much more successful, and the player is now able to jump to a certain height before falling back down to the floor. However, there was one more additional error that needed fixing, and this was that the player was able to jump whilst in midair, therefore enabling the player to never hit the ground.

To fix this error, I added an extra if statement into the keyPressed() function that waits for the input of the player, and the jumping value of the player is now only changed to true if the player is on the floor level.

```
function keyPressed(){
  if(keyCode == 87){
   if(player.y == floorHeight-100){
     player.jumping = true;
   }
 }
}
```

After these modifications and fixes to the code, I had a working movement mechanics system which responds to the inputs of the player. This prototyping has shown me that the usage of the draw() function will be very beneficial in the actual project development for when I need to execute an action multiple times whilst also continuing to run other game logic.

V0.2 Scrolling Level

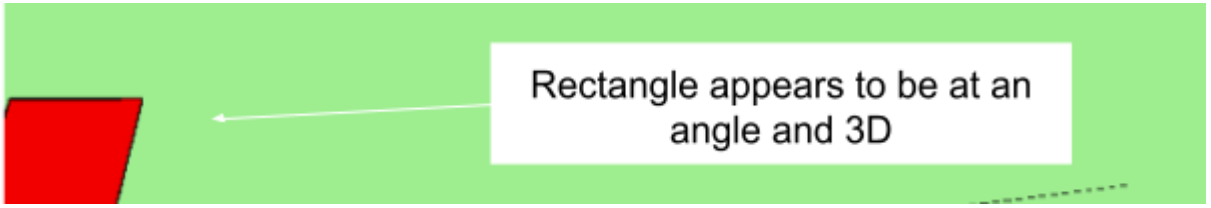
The next feature that I implemented into my prototype was a scrolling level design. In order for this to work, the player must always be centered on the canvas of the game, therefore the objects must move around them.

The first approach I took towards implementing the scrolling level was using a ‘camera’ that locks onto the player and follows their x coordinates. To do this, I researched the camera() function that is built into the p5js library.

Unfortunately, this approach was not viable because in order for the camera() function to work in the browser, the project must use a library named WebGL (web graphics library). This is because the camera must have a different z coordinate to the objects to ‘look’ towards them, and the canvas is made 3d using this library.

My first solution was to implement the graphics library into my project, however because the library causes the environment in the browser to become 3d, the traditional p5 drawing functions do not work correctly. The reason the drawing functions no longer worked correctly is because the WebGL 3d environment causes other factors to come into play within the drawing such as lighting and material.

```
canvas = createCanvas(windowWidth-100, windowHeight-20, WEBGL);
 camera(player.x, player.y, (height/2.0) / tan(PI*30.0 / 180.0), 0, 0,
0, 0, 1, 0);
```



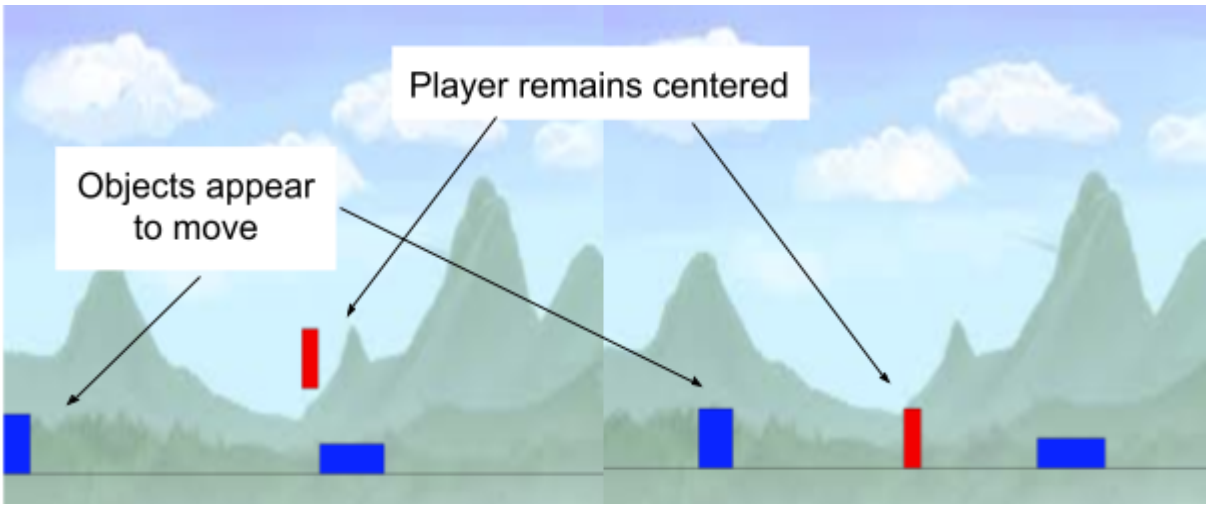
As seen above, when the camera function is called and the canvas is loaded with WEBGL, the player object and floor of the level appear in incorrect positions and are given a ‘3D’ effect.

The second approach was to use an extension of the p5js library called p5.play, which includes a different camera function that instead stores the center of the canvas as a reference to move the position of all other objects in the level. However, I decided against this approach because I realised that it was unnecessary, adding a lot of functions to the code through the library that I would not use.

I finally decided that the only approach would be similar to the p5.play library approach, but instead using the player's position as a reference to the center of the canvas, and moving the other objects depending on the player's position. I set the static position of the player to the center of the canvas and created some objects that moved depending on the x variable that was stored in the player object.

```
//adds some objects to the level
fill("blue");
rect(player.x-300,floorHeight-100,100,100);
rect(player.x+700,floorHeight-50,200,50);
```

The result was successful, and now when the movement keys are pressed the player appears to be moving through a level, even if the actual x coordinate of the player is a constant.



This prototyping has shown me that I am going to have to adopt a different approach to level design and generation during the actual development, where instead of the objects having set positions they are offset by the player's x position. It has also made me consider that there may be an increased difficulty with the collision detection development, because no positions in the game will have their own fixed locations.

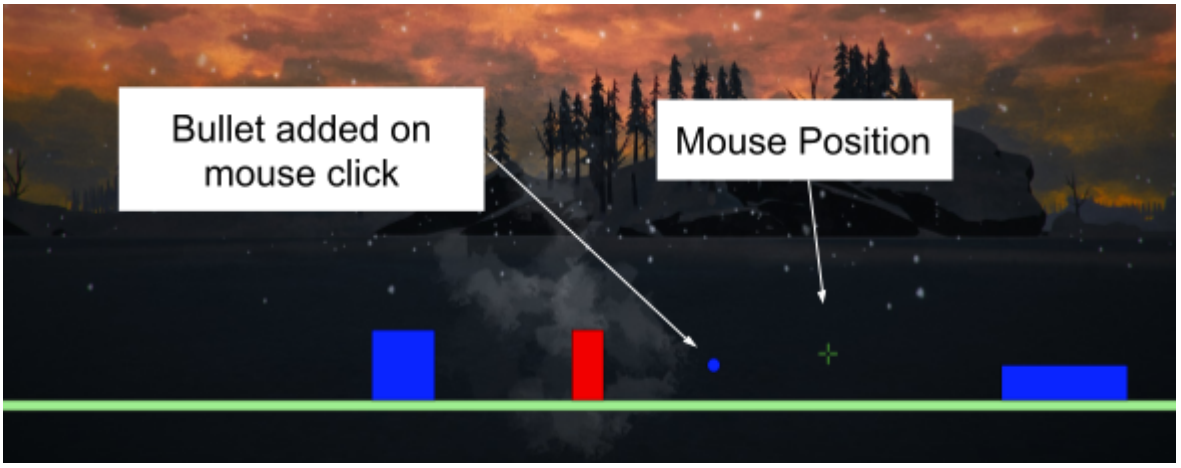
V0.3 Weapons & Shooting

Now that movement and level scrolling has successfully been implemented into the prototype, the next step was adding in the shooting mechanics and weapon types.

In order to add weapon types into the game, I decided on adding a 2d array called bulletStats which stores the name and statistics for each bullet type for the different weapons. I decided on this approach because it enables me to easily modify bullet statistics without having to create different object scripts with different values for each type of bullet. The array currently stores the name and speed of the bullet, however once different modules are added into the prototype, I can add values to the bullets such as their damage and corresponding sprite image.

```
//creates array that has statistics for all bullet types
 bulletStats = [
   ["pistol",20],
   ["rifle",30],
   ["rocket",50]
 ]
```

To implement shooting into the game, I added a crosshair onto the screen which draws at the player's mouse position and when the mouse is clicked, a new bullet object is added into a bullet array that is drawn onto the canvas each frame and travels towards the direction that the player clicked the mouse at.



As seen above, the blue circle is a bullet object that has been added when the player has clicked on the screen at the current position of the crosshair, and the bullet is travelling towards (and will go past) the crosshair.

The most difficult part of developing the shooting mechanics was the bullet movement after the bullet had been fired. The first attempt at implementing the bullet movement was successful, and the bullets travelled towards the target direction, however the bullets would travel towards the target at different speeds depending on the crosshair's position when the player fired the bullet. For example, if the crosshair was closer to the bullet's starting position it would travel slower than if the crosshair was far away.

To fix this problem, I had to find a way of calculating the vector that the bullet must travel in and then make this vector the same no matter how far away the crosshair was. I solved this using a built in p5 function called normalise(), This function used the vector that was calculated from the crosshair position and made it use a ratio of 1 between the x and y values. For example, if x=80 and y=100 then the new x would be 0.8 and the new y would be 1. After this normalised vector was found, it could be multiplied by the bullet speed to travel at different speeds.

```
//gets the normalised vector for the position of the mouse.
function getMouseVector(){
 let mouseXalt = mouseX - width/2;
 let mouseYalt = mouseY - player.y;
 let mouseDir = createVector(mouseXalt, mouseYalt);
 mouseDir.normalize();
 return mouseDir;
}
```

This method worked, and now the bullets travel towards and past the location of the crosshair when the bullet was fired at the same speed no matter how far away the crosshair was.

In order to link the bulletStats array to each individual bullet object, I made a function in the bullet object script that matches the bullet type given to the specific object to the name inside the bulletStats array, and then once the type is matched, the statistics are written to that object.

```
//calculates the bullet stats by locating the stats in the bullet
statistics array
 calculateTypeStats(){
   //loop through array until finds bullet with same type
   for(var bullet of bulletStats){
     console.log("called");
     if(bullet[0] == this.bulletType){
```

```
        //assigns speed to this object depending on bullet type
        this.speed = bullet[1];
        //for different bullet sprites;
        // this.img = bulletStats[x][2];
    }
}
```

This function is called only once upon the creation of the object, and as seen above is used to set the speed of travel for the bullet.

This prototyping has shown me that I have the ability to separate bullet data from the actual creation of each bullet object. I can now use this knowledge in my actual development to create links between different weapons and different bullet types. For example, a weapon does not have to be bound to a single bullet type, and instead could be switched to a different bullet type just by referencing its name before a bullet object is created.

Feature List

At this stage in the project development, I have researched and planned every individual feature that I would like to implement into the game. Here is a list of all the features (in no particular order) that would be included in the game regardless of whether they will actually be implemented or not.

- Player Mechanics
- Menu Functionality
- Level Selection Menu
- Level Design
- Enemy AI & Wave spawning
- Player Health & Weapon HUD
- Persistent Storage
- Sprites & Graphics
- Bullet Collision Detection
- Weapon Properties and Usage
- Weapon Selection Menu
- Sound Effects
- Player Economy & Shop
- Enemy Variety
- Multiplayer Compatibility
- Variety of levels
- Challenges & Additional objectives
- Settings Menu
- Health Packs
- Player Leaderboard
- Speed Runs
- Versus Gamemode
- Abilities
- Zombie Drops
- Microtransactions

- In game voice chat
- Controller compatibility

Success Criteria & Limitations

Throughout the research for the development of my project, I have listed a large amount of features that I would like to implement within Reclamation. However it is also important to set realistic and appropriate goals before the actual development. These goals involve establishing which features should be added first in order to make the game playable, which features should be added after the initial core of the game to give the game the most beneficial addition to user experience and which features could be added after both these goals have been met. This is my MOSCOW analysis – I have ordered features according to whether they Must, Should, Could or Won't be included in the program (the initial letters form the acronym MoSCoW).

- MUST (M1,2, etc.) be completed for the project to be a success (playable).
- SHOULD (S1,2, etc.) be completed for the project to feel like more than just a basic project.
- COULD (C1,2, etc,) be included if I have the time to code and test them.
- WON'T (W1,2, etc,) be incorporated into the game at this stage, because I know they will take too much time to code and test, although it would be nice in a future version of the game to implement them or if all other goals have been completed.

M1	Player Mechanics	The player mechanics are the most important feature that must be added to create a user experience within the game. The user will be able to control their player by moving around, jumping and shooting their weapon. They will move with the W,A and D keys and will shoot using left click.
M2	Menu Functionality	The player will be able to navigate through different menus using buttons that are displayed on the canvas. If a button is clicked, the menu is switched to the target menu of the button, and a new series of buttons are displayed on the screen as well as any other content from that menu.
M3	Level Selection Menu	The player will be able to choose between a screen containing buttons for all the levels in the game. If the player has unlocked the level and clicks a level button, the level will be selected and generated and the player will be moved to a weapon selection screen before they play the level.

M4	Level Design	At least a single level must be designed within the project in order for the player to have an objective to complete. Depending on the size of each level, there must be a reasonable amount of levels implemented for the user to play for a decent period of time. The levels will include objects that the player will have to destroy to move further on, and will have towers that when the player approaches will spawn waves of zombies. Zombies may also be spawned at certain points when the level is loaded initially.
M5	Enemy AI & Wave spawning	There will be different types of enemy AI however all of these types will function in the same manner: they will attack the player when they are within a certain radius, and will move towards the player when they are within a larger radius. Some AI will attack at close range with melee attacks, whilst other AI will shoot projectiles at the player. The AI will have health bars above their sprites and will take damage when hit with a player attack.
M6	Player Health & Weapon HUD	The player health will be stored in a player object during gameplay and will be displayed on a HUD (Heads up Display). This HUD will include a health bar and the current weapons the player is using with the equipped weapon's current ammo/max ammo.
M7	Persistent Storage	Player's progress and statistics will be stored within a database, so that when the player returns to play again they don't have to start from scratch. Game progress should include unlocked levels, kills, and player level.
M8	Sprites & Graphics	Basic sprites will be added into the game before further progress is made in order for objects and players to be distinguishable. The sprites will be loaded from sprite sheets and each sprite will be cropped into a variable and loaded onto specific objects when they are drawn.

M9	Bullet Collision Detection	Bullets that are fired by the player or the enemies will be destroyed once they collide with their specified target. Once collided, they should deal damage to the target. The damage dealt will be based on the bullet type damage
M10	Weapon Properties and Usage	Different weapon types will be implemented into the game that have different ammo types, reload speeds, mag sizes and fire rates. These weapons objects will be stored in object arrays and their properties will be used to calculate the behaviour of the player's shooting.
M11	Weapon Selection Menu	The player will be able to select between a list of all their owned weapons and customise which 5 weapons they would like to have equipped in their slots before they enter into a level. The screen will display their current equipped weapons and the player will be able to click weapons in and out of these slots.
S1	Sound Effects	Simple sound effects should be added to the game for when the player is shooting, for when zombies are attacking or spawning and for when the player completes a level. These sounds will make the game feel more reactive and enjoyable.
S2	Player Economy & Shop	The player should be able to earn money from their progress through kills and levels completed. This money will be stored persistently with their other progression in a database and can be used to purchase new items within an in game store. The in game store will be accessible once the player has returned to the menu and will include new weapons, ammo and grenades.
S3	Enemy Variety	A larger amount of enemy types should be added into the game to increase the difficulty as the player progresses. These enemies will have different attack types, statistics and spawn rates.
S4	Multiplayer Compatibility	Multiplayer compatibility will be added by creating a client/server side architecture. The project will be moved to NodeJS as a server

		and Socket.io will be used for the clients to communicate with the server. Each action each client makes must be communicated with the server and the server must emit this event to all other clients so that the same things are happening on each client's displays.
S5	Variety of Levels	Once all the other features above have been implemented and work successfully, a portion of development time should be put towards the development of new levels because it will enable the player to progress for longer and give the user more options when they are playing. New levels must include different wave spawning algorithms to incorporate new AI types.
S6	Challenges & Additional objectives.	Additional challenges and objectives should be added whilst the player is progressing through a level because it will give them a new way to earn XP and money that they can use for the shop. These challenges will track player kills and other statistics throughout a level and at the end of the level if the objective has been completed a page is displayed showing rewards.
S7	Settings Menu	A settings menu should be added so that the player can choose between turning sound on and off as well as for backing up account progress and exiting a level.
S8	Health Packs	Health packs should be able to spawn in game to help the player restore their health. These could spawn at specific points in the game and be coded into the level generation. If the player goes over the health pack a collision check is made which causes the player to gain health and the health pack to disappear.
C1	Player Leaderboard	A player leaderboard could be added into the game where real users can compare their stats for kills, levels completed, and money earned. This will make the game more competitive and encourage players to keep playing to beat each other's scores. The leaderboard will use statistics fetched from a player database and compare these values before displaying ordered stats on a table of contents that the player can view.

C2	Speed Runs	A timer could be added to time how long a player takes to complete a level, this could further be added to a leaderboard system where players could compete for the fastest time.
C3	Versus Gamemode	A versus gamemode could be added where players can 1v1 each other instead of teaming to fight zombies. Events would have to be modified to enable player attacks to damage other players.
C4	Abilities	Abilities could be added that players can purchase in the in game shop. This could help diversify combat and create a higher skill boundary for choosing when to use the abilities. Abilities would need new events that cause different actions depending on the nature of the ability. Abilities could heal players, damage zombies, or make players invulnerable for a short time.
C5	Zombie Drops	As well as allowing the player to gain money when they kill zombies, the zombies could have a drop rate where there is a small chance that players can pickup a new rare weapon. An event would need to be added every time a zombie dies and a probability check would need to be carried out.
W1	Microtransactions	I will not attempt to add a system into the project where the player can purchase in-game items with real money because the development process would be very complex and there is a large margin for error. This could cause issues if a player doesn't receive the item after purchase.
W2	In game voice chat	Voice chat will not be added into the game because it would be extremely difficult to implement due to it requiring the use of different protocols as well as needing additional resources to function. I don't believe it would be the best way to spend my development time due there being many different methods to communicate online (discord, skype, teamspeak etc.)
W3	Controller compatibility	I won't add additional compatibility for the controller to play as an input because of the initial platform for the game being in a browser

		and therefore the majority of users being on a computer with a mouse and keyboard.

Hardware & Software Requirements

My goal is to develop a web based game that is run in a browser environment therefore I am going to be using JavaScript with multiple well known js libraries to code the majority of the game logic and graphical display. JavaScript is the most common language executed by web browsers, therefore the game will be able to run on every up to date web browser available.

In order for potential players to access and play the game, the files which store the game code must be served to the player via a web server that the player can connect to in the browser. It is possible for me to develop the game on my local machine within an IDE (Integrated Development Environment) and then upload the final product to a hosting service, however I have developed previous projects in an online text editing environment called repl.it, and this is the approach I would like to take for Reclamation. Repl.it allows me to host a node.js web server, which is key for the server side development of the game (web sockets and database hosting).

A web server enables me to make changes to the game code at any point in time and the users will not have to perform any software updates because the updated code will be served to them each time they reconnect. The one negative to a web server approach however is that the user will not be able to access the game whilst they are offline, and if the game is loaded and they lose internet connection, their progress will not be saved unless they reconnect.

I will be using several JavaScript libraries to simplify the development process. These libraries include p5.js for the graphics, socket.io for the client server communication, and express to setup a web server which will serve files to connected users.

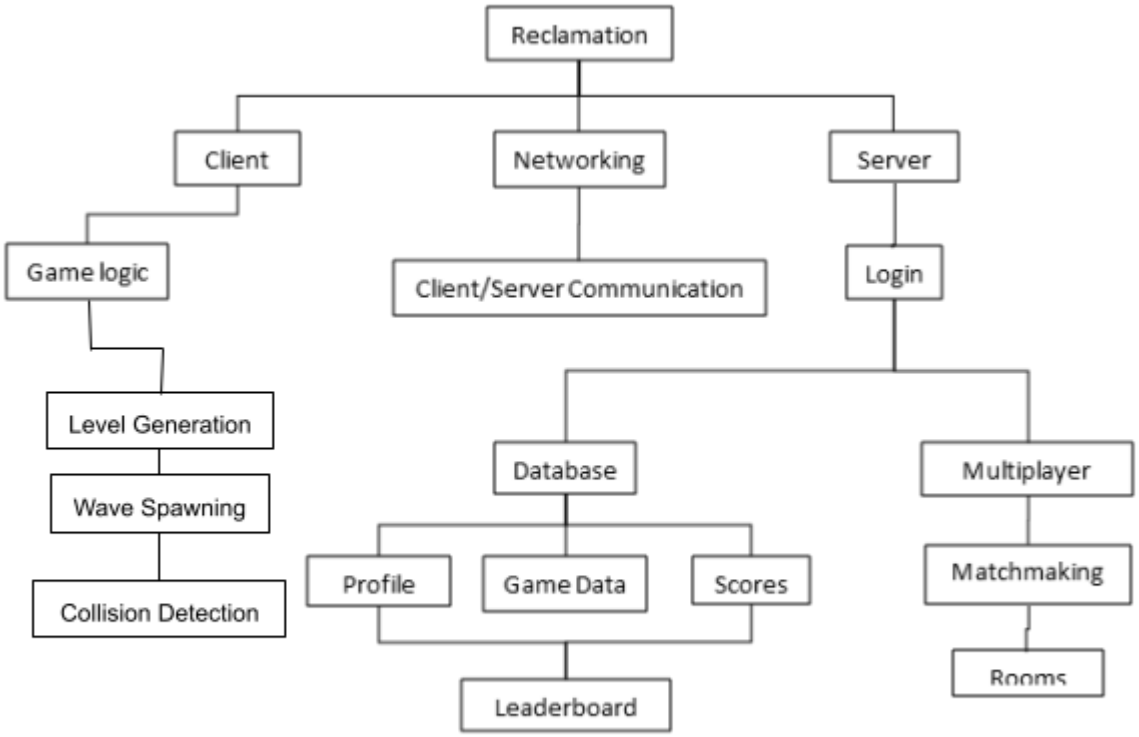
There are no specific hardware requirements for the development process of Reclamation. Repl.it provides a free web server to host the game from, and the programming of the game can be done from any computer with a browser, because repl.it stores my game code online and I can access it from my account on any device. When I am at home, I will be developing the game on my personal computer, and when I am at college I will be developing the game on a college computer. Furthermore, the game is not going to be graphically intensive (because of the pixelated style) and therefore the user will not require a powerful graphics card to run the game with a high performance.

Design

Module Decomposition

The project can be separated into several different modules, and there are sub-modules within each one of these modules. Understanding where these modules separate and how they relate with each other will help me to simplify the design and development process.

The decomposition design diagram below offers a visual representation of the project.



If the game meets it's desired success criteria, it will have a client and server model. The server side will be responsible for the database management (storing the player progress and login information). Whilst the client will be responsible for everything else in the project (graphic display, game logic, input detection, etc).

Internal & External Data Storage

Client Side Storage

Player Class

Property name	Datatype	Comment
Health	Integer	The player health is displayed using a health bar drawn outside of the player class. Value is changed when the player is attacked or uses a health pack. The player has a maximum health value of 1000 and a minimum health value of 0.
Speed	Integer	Player speed will affect how fast the player moves in the x direction.
jumpSpeed	Integer	jumpSpeed will affect how quickly the player jump animation is completed by changing the speed that the player moves in the y direction.
jumping	Boolean	The jumping boolean will change depending on whether or not the player jumping animation has been called or completed.
x,y	Float	Player x value will affect every other object within the game other than the player object, so that the player appears to be moving. Y value of the player will only change when they are jumping or on an elevated obstacle.

Constructor function

The constructor function will take 2 parameters for the player's x and y initial coordinates.

Render function

The render function is executed every p5 draw cycle (60 fps) and draws the player sprite onto the canvas.

Update function

The update function is also executed every p5 draw cycle and is used to handle the jumping animation of the player as well as handling collision checks between objects and the player.

Left and right functions

These functions increase or decrease the player x direction when the player has pressed the arrow keys.

Enemy Class

Property name	Datatype	Comment

Health	Integer	The enemy health is displayed as a bar above the enemy sprite. Health is decreased when the enemy is hit by a player attack. Enemy health can vary depending on the type of enemy.
x,y	Float	The enemy x value is offset from the player x value, the player y value is individual to each enemy object.
maxHealth	Integer	Stores the max health value for the enemy.
enemyType	String	The enemy type is a value that is used to calculate what the properties of the enemy will be initialised as.
spriteSheet	PNG	The sprite sheet will contain all the images of the enemy that can be displayed on the canvas for different actions that the enemy performs.

Constructor function

The constructor function will take 3 parameters for the enemy x value, y value, and enemy type.

PropertyInitialise function

This function will take the type value for the enemy that was inputted during the constructor function and use this type value to initialise corresponding values for the enemy maxHealth and spriteSheet.

Attack function

The attack function will use the enemy type value to call a specific attack method that the enemy will use, these attack methods will also be hard coded methods stored in the enemy class.

DistanceCheck

This function will compare the x value of the player with the x value of the enemy and if they are within a certain range of each other depending on the enemy type the enemy will be able to attack the player.

Render function

The render function is executed every p5 draw cycle (60 fps) and draws the enemy sprite onto the canvas.

Update function

The update function is also executed every p5 draw cycle and is used to handle movement of the enemy by updating the x and y values of the enemy.

Bullet Class

Property name	Datatype	Comment

x,y	Float	Stores the x and y values of the bullet object
xSpeed,ySpeed	Float	These values are calculated for the bullet before the bullet is created depending on where the player has clicked on the screen. They hold the directions for the bullet to travel.
speed	Float	This value stores the speed of the bullet depending on the type of projectile, It is used to multiply the directional vector of the bullet so that the bullet travels to a point slower or faster.
bulletType	String	This stores the type of bullet - this is used to calculate the speed of the projectile and the bullet sprite
sprite	PNG	Stores the bullet sprite

Constructor function

The constructor function will contain the values for the xSpeed, ySpeed and type of bullet.

CalculateTypeStats

This function will use the bullet type value from the constructor function to initialise the values for the speed and sprite of the bullet.

Render function

The render function is executed every p5 draw cycle (60 fps) and draws the bullet sprite onto the canvas.

Update function

The update function is also executed every p5 draw cycle and is used to change the x and y values of the bullet whilst it travels towards its aimed position.

Server Side Storage

The server side storage enables the player to have persistent storage of their progress through making an account on the game. This account will contain all the details for their progress such as their statistics and unlocked features. This data will be stored on an SQL database and will be accessed via a login page when the player starts the game.

SQL Player Profile Database

Field	Datatype	Comment
Username	String	Stores the username of the individual user
Password	String	Stores the password for the

		player's account so that the player can verify their credentials before logging into their account.
Level	Integer	Stores the level of the player
Kills	Integer	Stores the amount of kills the player has accumulated throughout their playtime.
Weapons	Array	Stores all the weapons that the player has unlocked throughout their playtime.
Credits	Integer	Stores all the coins that the player currently has on their account that they can spend to buy in game items.
LevelsUnlocked	Array	Stores a list of all the levels that the player has completed and has unlocked.
Score	Integer	Stores the accumulated score of the player over their entire playtime.
LevelTimes	Array	Stores the players fastest time for each completed level. This will be used for the speedrun highscores table.
Items	Array	This will contain any other items that the player has unlocked in the game including any unlocked abilities.

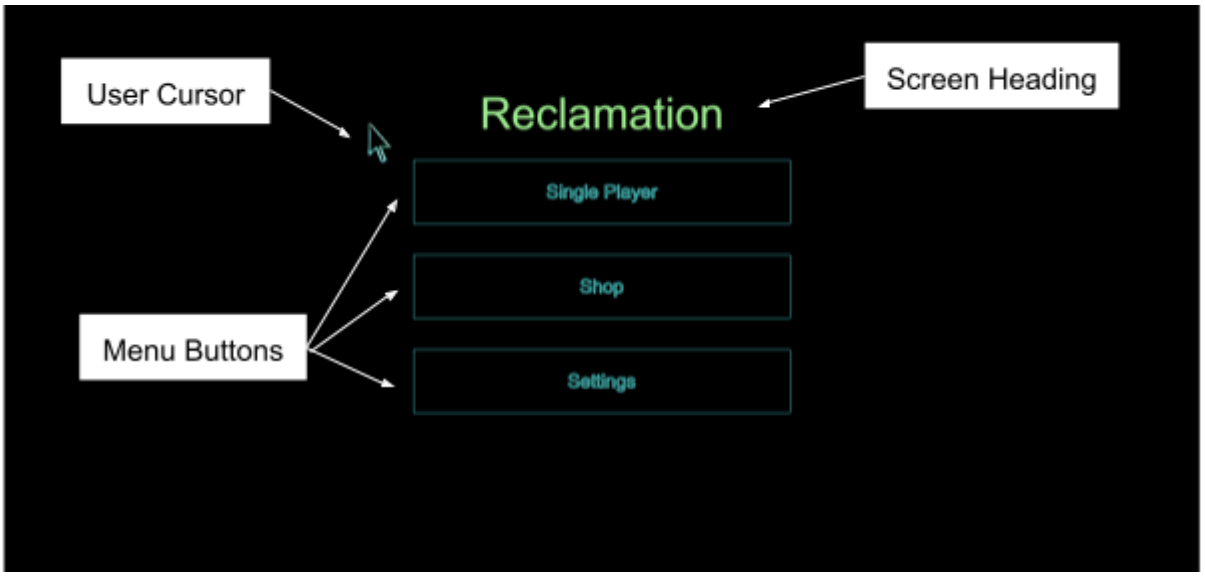
Screen Designs

Core Screen Display

The game is designed for browsers with a monitor resolution of 1920x1080p, with the p5 canvas (where all graphics are displayed) having an exact size of 1905x917 pixels. However, the game can easily be played on monitors of varying sizes because the user can simply use the browser zoom in / zoom out functionality to change the screen size if it is too small / too large.

Menu Screens

The menu screens will have a black background and will contain buttons with a blue border and blue outlined text. The screens will have a light green heading above the buttons describing the menu and the user will have a special neon cursor that they can use to click the buttons.



Gameplay Screen

The main gameplay screen will contain a player HUD at the top of the screen that displays the statistics necessary for the player to play, as well as the actual gameplay (level graphics, enemies, and player) below. There will be a button in the top right that the player can use to pause the game.

Algorithms

Several of the MUST objectives require prior planning of the algorithms which will be used before the development process begins in order to ensure that the algorithms that are implemented will work as intended and without issues.

Bullet Collision Detection

This algorithm will detect if a bullet object has collided with it's target type by comparing the x and y positions of the bullet and the target.

Parameters: *Target type, Bullet x position, Bullet y position*

- Confirm Target Object
    - ➢ Use Target type parameter in a switch case statement with cases for player and enemies
    - ➢ If enemies target type the enemies array is looped through one by one for collision checks with each enemy position being check
    - ➢ If player target type player position is used for collision check
- Perform Collision Check
    - ➢ Does the top of the bullet y overlap with the bottom of the target y
    - ➢ Does the bottom of the bullet y overlap with the top of the target y
    - ➢ Does the farthest right bullet x position overlap with the farthest left target x position
    - ➢ Does the farthest left bullet x position overlap with the farthest right target x
    - ➢ If any of the above are true the bullet has collided and perform collided code
- Collided Code
    - ➢ Destroy the bullet (remove bullet from bullets array)
    - ➢ Decrease target health by the damage of the bullet
    - ➢ If the target health is equal to or below zero destroy target (remove target from target array or if player was target game over screen appears)

To test the algorithm I will initiate different actions in the game to ensure that the collision detection doesn't have any of the below errors:

- Bullet not being destroyed after a collision
- Damage not being dealt to target after collision
- Bullet colliding with objects that aren't specified target type
- Bullet not colliding with target type

Weapon Reloading

This algorithm will verify the necessary parameters are in place before reloading a player's weapon.

Parameters: *Weapon current ammo, Weapon mag size, owned spare ammo*

- If Weapon Mag Full
  - ➢ Reload can't take place

- If Weapon Mag Partially Full
  - ➢ Check if enough spare ammo to reload mag to full capacity
  - ➢ If enough spare ammo perform reload at reload speed of weapon
  - ➢ If not enough spare ammo perform reload with last remaining spare ammo into mag at reload speed of weapon
  - ➢ If no spare ammo reload can't take place
  - ➢ Take away spare ammo by the amount of bullets loaded into the mag

- If Weapon Mag Empty
  - ➢ Check if player has any spare ammo
  - ➢ If player has spare ammo fill mag up to full capacity if owned ammo is sufficient and if not enough owned ammo to fill to full capacity fill magazine partially with last remaining spare ammo
  - ➢ If no spare ammo reload can't take place
  - ➢ Take away spare ammo by the amount of bullets loaded into the mag

To test this algorithm I will:

- Try to reload weapon with no remaining spare ammo
- Try to reload weapon with only a few spare bullets remaining
- Try to reload weapon with lots of spare ammo to see if mag fully reloads

Development Log

V0.1 Menu Functionality

The first feature that I have implemented into the official version of the game is menu functionality. I created a button class that draws a button onto the screen with a specific title and function, and these buttons have been separated into different arrays to create button menus that are drawn into different scenes. These scenes can now be navigated between by clicking the appropriate buttons, and this is how different sections of the game are going to be separated.

```
mainMenuButtons = [
  new Button("Single Player", 952.5, 300, 600, 100, () => { gameMode =
'singlePlayerMenu'; buttons = singlePlayerMenuButtons }),
  new Button("Shop", 952.5, 450, 600, 100, () => { gameMode =
'playerShop'; buttons = playerShopButtons }),
  new Button("Settings", 952.5, 600, 600, 100, () => { gameMode =
'settingsMenu'; buttons = settingsButtons })]
```

The code above creates a menu array with the buttons specific to that menu, and the code below in the main draw function contains a switch-case statement that will execute a specific scene depending on the one specified in a ‘gameMode’ variable.

```
switch(gameMode){
  case 'mainMenu': mainMenu();
    break;
  case 'singlePlayerMenu': singlePlayerMenu();
    break;
}
```

The next step of development will be to create the single player gamemode scene, transferring over player movement mechanics and level generation from my prototype and improving the drawing functionality onto the canvas.

V0.2 Player Movement and Basic Level Generation

In order to initialise the necessary objects and variables for the single player environment, I had to implement a separate function within the single player game scene called singlePlayerInitialise(). This function is executed once the player has clicked the play button from the single player menu, and this was very simple to do as I was able add the function to the button code that is executed after it is clicked.

```
new Button("Play", 952.5, 300, 600, 100, () => { gameMode =
'singlePlayerGame'; buttons = inGameButtons; singlePlayerInitialise()
})
```

Next, I had to transfer the game loop code from my prototype into the singleplayer game scene. I created a player class with the same properties as the prototype, and after the player object has been initialised in the initialise function, the game loop checks for the movement of the player through the appropriate key presses as well as generates a few basic objects to simulate a level. The game loop is executed each draw cycle after the scene is selected.

To simplify more complex level development and collision detection later down the line, I modified the default method of drawing onto the p5 canvas by changing the settings to CENTER rather than CORNER. This will mean that objects will draw directly out from the coordinate I specify in all directions, instead of the coordinate being in the top left corner.

```
//settings for the alignment of anything drawn on the screen using p5
rectMode(CENTER);
textAlign(CENTER);
imageMode(CENTER);
```

The next step of development will be to create a level select menu, where the player can choose the level they would like to play before it is generated.

V0.3 Level Select Menu

In order for the level select menu to work, there must be a way for the player to interact with the main single player game loop. To implement this, I created a new variable called ‘currentLevel’. This variable is updated each time the player clicks a different level button from the level select menu which was created using the usual button functionality. This currentLevel variable is now used at the beginning of the single player game scene to call a specific generate level function.

```
/* checks the current level that the player has selected and generates
the level from the corresponding level function */
switch(currentLevel){
  case 1:  generateLevel1();
    break; }
```

Separating the generation of different levels into individual functions in the single player game scene will be useful because it will make it much easier to debug specific levels and also enable me to add more and more levels whenever I want and only need to add a new button to the level select menu.

The next target for development will be to create an enemy object that can have several different attack types and properties which can be fetched from an enemies array. The enemy will be able to track the player and move towards them and attack the player when they are within a specific range.

V0.4 Enemy Initialisation and Player Tracking

The first step to adding enemies into the game was to create a specific enemy class with a function that fetches its properties from an array. This function is called propertyInitialise() and locates the corresponding enemy type from an array of enemy objects. It loops through each enemy object and compares the enemy object type to the type inputted on creation of the enemy object.

```
//initialises array that stores each enemy object type with its values
var enemyTypes = [
{ type: "basic", speed: 0.8, health: 30, spriteSheet: null,
attackDistance: 700, attackType: 'projectile'},
{ type: "speedy", speed: 5, health: 20, spriteSheet: null,
attackDistance: 300, attackType: 'projectile'} ]
```

```
for(var enemy of enemyTypes){
    if(enemy.type == this.type){
      this.health = enemy.health }}
```

After the enemy properties have been initialised, the enemy needs to have functionality that enables it to track the location of the player and move towards it depending on its attack range. To add player tracking, I created a function that executes each draw loop called travelDistanceCheck().

```
//travel closer to player if not within the attack range
  if( (this.x+player.x) - width/2 > this.attackDistance +
this.attackDistanceOffset){
    this.x -= this.speed ;
  }
  else if( (this.x+player.x) - width/2 < -this.attackDistance -
this.attackDistanceOffset){
    this.x += this.speed;
  }
```

The code above causes the enemy to move closer to the player if they aren't within attack range. attackDistanceOffset is a random value between 0 and 100 that enables enemies who have the same attack distances to be separated on the canvas a small amount, so that they don't stack up and cause the player to only see one enemy. I solved this issue after

executing the game and finding that no matter how many enemies I added into the game, after a period of time there would only appear to be one.

```
//checks attack method and performs correct attack type once enemy
comes into attack range
      switch(this.attackType){
         case 'projectile':
            //calculates the direction the bullet has to travel in
            let bulletDir = createVector( ( width/2 - (this.x +
player.x) ), (player.y-this.y) );
            bulletDir.normalize();
            currentBullets.push(new Bullet((this.x+player.x), this.y,
bulletDir.x, bulletDir.y, 'pistol', 'player'));
```

Once in attack range, the enemy should stop and begin to attack the player with a specific attack type. The code above checks the enemy attack type once it is in the attack range, and if it has the projectile attack type it fires a bullet in the direction of the player.

The next step of development will be to add collision detection for the bullets that the player fires as well as the projectiles fired by the enemy at the player.

V0.5 Bullet Collision Detection

To implement bullet collision detection, I had to add a function to the bullet object that executes each draw loop which checks the x and y values of the bullets edges in comparison to the x and y values of the bullet target. This function was called collisionCheck(). In order to distinguish between different target collision checking, I added a new constructor parameter to the bullet class called ‘target’, which differs depending on if an enemy or the player shot the bullet, and when the collisionCheck function is called it executes different checks depending on the target.

```
switch(this.target){
      case 'enemy': //checks if the bullet x and y is colliding with
any enemies currently in the game
      for(var i=0; i < currentEnemies.length; i++){

        if( (this.x+this.size/2) >= (currentEnemies[i].x + player.x) -
currentEnemies[i].width/2 && (this.x - this.size/2) <=
(currentEnemies[i].x + player.x) + currentEnemies[i].width/2 &&
(this.y+this.size/2) >= currentEnemies[i].y- currentEnemies[i].height/2
&& (this.y-this.size/2) <= currentEnemies[i].y +
currentEnemies[i].height/2){
          currentEnemies[i].health -= this.damage;
```

```
            this.destroyed = true;
}
```

The main difficulty with collision detection was for bullets fired at the enemy. This was because enemies are displayed on the canvas with an offset of the placeholder value of the player so they appear to be moving. This meant that when I first attempted to implement the enemy hitbox the bullet would disappear and damage the enemy when it reached a different location to the actual location of the enemy on the screen. Therefore, player.x had to be taken into account for the coordinates of the boundaries of the enemy hitbox.

To remove the bullet from the game once it has collided with its target, I created a variable for the bullet object called ‘destroyed’, this value is checked within the single player game loop and if true, the corresponding element location for the bullet that needs to be removed is spliced from the currentBullets array.

```
//removes bullets from the game if they have collided with their target
  else if(currentBullets[i].destroyed){
    currentBullets.splice(i,1); }
```

The next goal for development will be to differentiate between rates of fire for the enemies depending on their type as well as having different rates of fire for the player depending on which weapon they currently have equipped.

V0.6 Weapon & Enemy Specific Rates of Fire

The easiest method of implementing fire rate functionality into the game is using the built in p5 function millis(). millis() displays the current amount of milliseconds that have passed since the program has been executed. This can be used to create time frames between each bullet that is fired.

This was relatively simple to implement for the enemies. The first step was to assign a new property to each enemy type called attackSpeed. This property manages the rate of fire for projectile enemies and the rate of melee attack for melee enemies, and is stored in seconds. The next step was to add a new variable to each enemy object that would store the time in which the last attack was carried out, this variable was called timeLastAttacked. Finally, I added an if statement into the attack distance check function, and if the time since the last attack had been longer than the assigned attack speed, the enemy would attack again and update its timeLastAttacked value.

```
//checks if enough time has passed since the last attack and attacks
again depending on attack speed
 if( millis() - this.timeLastAttacked >= this.attackSpeed * 1000 ||
this.timeLastAttacked == 0){
  // code to attack player with either projectile or melee
     //updates the time last attack happened
```

```
    this.timeLastFired = millis();
  }
```

The process of implementing specific weapon types for the player was more difficult. Firstly, I created a new array called weaponTypes which contains a list of all weapon objects that are available to be unlocked in the game. These weapon objects contain properties necessary for the use of the weapon, as well as linking each weapon to a specific bullet type and image.

```
var weaponTypes = [
{ type: "Glock", bullet: "9mm", magSize: 7, fireRate: 1, sprite: null},
{ type: "Magnum", bullet: "44 Magnum", magSize: 10, fireRate: 1.5,
sprite: null},
{ type: "AK-47", bullet: "7.62mm", magSize: 30, fireRate: 0.4, sprite:
null}
```

The next step was to create several new variables that the player would use to store their currently equipped weapons and currently owned ammo.

The currently equipped weapons are stored in a static array named equippedWeapons with 5 elements for 5 different possible weapon slots. The array includes objects with the names of the weapons equipped and the amount of ammo in the mag for the weapon. The name will be matched to the correct weapon object later when needed. I also created an integer variable called currentWeaponSlot, which stores the location of the weapon slot they are currently using in the equippedWeapons array.

```
var equippedWeapons = [
{name: "AK-47", ammoInMag: 30},
{name: null, ammoInMag: null},
{name: null, ammoInMag: null},
{name: null, ammoInMag: null},
{name: null, ammoInMag: null}
]
```

For the currently owned ammo, I created an array which contains an object for each type of bullet that contains the bullet name and the amount of bullets that the player currently owns.

```
var ownedAmmo = [
{ type: "9mm", amount: 'unlimited'},
{ type: "44 Magnum", amount: 0}
];
```

The final step of the weapon development process was to use all of these new variables to calculate what should happen when the player is trying to shoot. I swapped the method of detecting the input of the player from using the p5 mousePressed() function to using the p5 boolean mouseIsPressed in an if statement. This meant that I was able to detect if the player was holding the shoot button rather than just detecting whether or not the player had pressed it. By making this swap, I was then able to make the weapon of the player shoot at a specific fire rate without the player having to re-press the shoot button.

After detecting that the player was trying to shoot, I had to carry out multiple checks for which weapon the player is trying to shoot, if they have enough ammo in the magazine to shoot the weapon, and if enough time has passed since the last bullet has been fired.

The first check just required matching the player's currently equipped weapon slot to the correct weapon object in the weaponTypes array, and then using the located weapon to find the correct type of ammo needed to make the weapon shoot.

```
//finds the corresponding weapon type from the currently equipped
weapon
  for(weapon of weaponTypes){
    if( weapon.type == equippedWeapons[currentWeaponSlot].name ){
    //finds the correct ammo type for the weapon the player has
equipped
      for(ammo of ownedAmmo){
        if(ammo.type == weapon.bullet) {
```

The next check required checking if the weapon had more than 0 ammo in it's magazine or if it was a special weapon that has unlimited ammo.

```
//checks the player has enough ammo to shoot weapon
          if(equippedWeapons[currentWeaponSlot].ammoInMag > 0 ||
equippedWeapons[currentWeaponSlot].ammoInMag == "unlimited") {
```

The final check required a similar method to the enemy attack rate management, using the millis() function to see if enough time had passed since the last bullet had been fired, and if enough time had passed, the player was able to shoot a bullet with the type that the weapon they had equipped carried.

```
//checks if enough time has passed since last bullet was fired for
weapon to shoot
      if(millis() - timeLastFired >= weapon.fireRate * 1000 ||
timeLastFired == 0){
```

After all the checks were implemented, the result was a success, and the player is now able to shoot different types of bullets at different fire rates depending on the weapon that they have equipped and if they have ammo left in their magazine.

The next goal of development will be to implement a reloading system for the weapons, where if the player has less than maximum ammunition left in their weapons magazine, they will be able to press a reload button that will check if they have spare ammo that can be loaded into the magazine at a weapon specific reload time. I would also like to finalise the weapon development by adding a HUD section for the player where they can see their currently equipped weapon as well as it's ammo information

V0.7 Weapon Reloading

The first step to implementing a reloading mechanic into the game was to give each weapon a new reloadSpeed property. This property specifies the amount of time in seconds it takes each weapon to reload their weapon magazine, and this was added to the weaponTypes array objects.

```
{ type: "Glock", bullet: "9mm", magSize: 7, fireRate: 1, sprite: null,
reloadSpeed: 3},
```

The next step was to alert the player to when it is necessary to reload. I implemented this using a simple if statement and if the mag is empty then text is drawn onto the screen above the player that says reload with the correct key they must press to reload their weapon (R).

```
//alerts the player if they need to reload
  if(equippedWeapons[currentWeaponSlot].ammoInMag <= 0){
    fill("white");
    textSize(20);
    text("Press 'R' to Reload", width/2, player.y - 150);
  }
```

Once the player was able to be alerted that they needed to reload, I needed to detect when the player had pressed reload, and to do this I used the same p5 function previously used to detect when the player pressed the movement keys, the keyIsDown() function. I used the website keycode.info to find the correct keycode for 'r' which is 82.

After it is detected that the reload button has been pressed, several checks have to be carried out before reloading the player's weapon magazine. The first check was for if the equipped mag wasn't full so that a reload could occur.

```
if( equippedWeapons[currentWeaponSlot].ammoInMag < weapon.magSize ){
```

The second check was to see if the player had enough ammo left to fully reload the magazine, if they do, a full reload is carried out.

```
if( weapon.magSize - equippedWeapons[currentWeaponSlot].ammoInMag <=
ammo.amount){
              //takes away the ammo from the players spares that is
going to be used
              ammo.amount -= weapon.magSize -
equippedWeapons[currentWeaponSlot].ammoInMag;
              //fully reloads the magazine
              equippedWeapons[currentWeaponSlot].ammoInMag =
weapon.magSize; }
```

If the player did not have enough ammo left to fully reload the magazine, another check is made for if the player has a little bit of ammo left to partially reload the magazine.

```
  else if( ammo.amount > 0 ){
      //reloads the magazine with the remaining spare ammo
      equippedWeapons[currentWeaponSlot].ammoInMag += ammo.amount
      //takes away the final ammo from the players spares that was used
      ammo.amount = 0; }
```

After all these checks were implemented, the player was now able to shoot their weapon and reload it at any time.

The final step to implementing reloading was to combine the checks made with the reloadSpeed property to add a delay between when the player presses reload and when the player is able to use their magazine again.

To implement this delay, I used the same method of delay that I used with fire rates, incorporating the millis() function.

```
reloadStartTime = millis();
if( millis() - reloadStartTime > weapon.reloadSpeed * 1000 ||
reloadStartTime == 0){
```

However, upon testing these changes, the player would become stuck reloading their weapon for an unlimited amount of time. I found that this was happening for 2 reasons, one because the reloadStartTime was constantly getting updated each time the if statement was executing, and two because the if statement would only execute if the player was holding down the r key.

To fix these problems, I initialised a new boolean variable in the game called currentlyReloading which would change when the player started and finished a reload. This

value was then taken into account for when the reload mechanic code was to be executed, and when the reloadStartTime value was to be changed.

```
if(keyIsDown(82) || currentlyReloading){
//executes the reload code if the r key is being pressed or the player
is currently in the process of reloading.
```

```
//sets the start time of the reload for reload delay reference when the
player first begins to reload
      if(!currentlyReloading){
         reloadStartTime = millis();
      }
      currentlyReloading = true;
```

The next step of development will be to display the weapon information on the player HUD with its corresponding weapon sprite.

V0.8 Weapon & Ammo HUD

The first step to creating a weapon and ammo HUD was to initialise a sprite sheet array that contains the locations for individual sprite images on a sprite sheet source image. To do this I created a new script called spritesheet.js with defined arrays for each sprite sheet.

A defined sprite sheet array contains objects for each sprite. These objects store data for each individual sprite frame of that sprite. However the weapon sprite sheet array was simpler because there is no animation needed for weapon sprites. The first element contains the width and height of the sprite frame on the source image, and each object contains the location of the sprite frame on the source image.

```
var weaponSprites = [
{name: "Glock" sx: 80, sy: 50, sWidth: 100, sHeight: 100}
]
```

The next step was to match the correct weapon sprite to the player's equipped weapon. To do this I wrote a function that loops through the sprite array and matches the weapon name to the sprite name.

```
function matchWeaponSprite(name){
  for(sprite of weaponSprites){
    if(sprite.name == name){
      //sets default width and height of sprite frame
      sprite.sWidth = weaponSprites[0].sWidth;
      sprite.sHeight = weaponSprites[0].sHeight;
```

```
      return sprite;
    }
  }
}
```

Now that the correct weapon sprite can be fetched, I added the first part of the weapon HUD by drawing the weapon sprite next to the player's health bar.

```
  let weaponSprite =
  matchWeaponSprite(equippedWeapons[currentWeaponSlot].name);
  text(weapon.type, 500, 50);
  //image of weapon
  image(weaponSpriteSheetImg, 500, 100, sprite.sWidth*1.5,
sprite.sHeight*1.5, sprite.sx, sprite.sy, sprite.sWidth,
sprite.sHeight);
```

This gave the result of showing the player's weapon correctly.

Finally, all that was left to do was to display the players remaining ammo in their mag, and their remaining ammo they could use to reload their equipped weapon. Using a previously made matchWeaponType and matchBulletType functions, I retrieved the weapon and owned ammo info and used it to display the mag size and spare ammo next to the weapon sprite.

```
//Display weapon name, ammo left in mag and spare ammo remaining
    textSize(30);
    text(equippedWeapons[currentWeaponSlot].name, 610, 45);
    text("Spare: " +
matchBulletType(equippedWeapons[currentWeaponSlot]).amount , 610, 75);
    textSize(40);
    text(equippedWeapons[currentWeaponSlot].ammoInMag + "/" +
weapon.magSize, 480, 90);
```

The final weapon HUD worked as intended and displayed the weapon info correctly for the players equipped weapon.

Using the spritesheet.js file, the next step of development will be to create sprite arrays for enemy types and the player, then use these arrays to animate the objects.

V0.9 Player and Enemy Sprite Animation

The first step of creating sprite animation for the enemies was to create a sprite sheet array for all enemy types. I created this array with a similar format to the weapon sprite sheet, however each object had to contain data about individual sprite frames, and therefore the objects contain nested objects with the sprite frame data. Furthermore, to simplify the animation process, the frames are stored in a specific order (left facing frames to right facing frames).

```
var enemySprites = [
{name: null, sWidth: 45, sHeight: 72},
{name: "basic", frame1: {sx: 434, sy: 668}, frame2: {sx: 504, sy: 665},
frame3: {sx: 516, sy: 489}, frame4: {sx: 442, sy: 492} }
}
```

To animate the enemy sprites, I decided to use the x position and state of the enemy as a reference point for frame changes. The state of the enemy is which direction they are moving in or if they are stationary, and the x position is used to swap frames after the enemy has passed a certain distance.

The first step was to create a new property for the enemy object called currentDirection and calculate the value of this property.

```
this.currentDirection //used for sprite animation, shows direction of
enemy travel
```

To calculate the value of the property, I used the enemy travelDistanceCheck() function which already calculates where the enemy needs to travel. If the enemy is stationary, the currentDirection property checks if the enemy is to the left or right of the player, and chooses it's facing direction accordingly.

```
if( (this.x+player.x) - width/2 > this.attackDistance +
this.attackDistanceOffset){
    this.x -= this.speed ;
    this.currentDirection = "left";
```

```
  }
```

```
   //if enemy is not moving
   if(this.x + player.x >= width/2){
     this.currentDirection = "leftstatic";
   }
   else if(this.x + player.x < width/2){
     this.currentDirection = "rightstatic"; }
```

The next step was to match the correct enemy sprite object to the enemy object. To stop an unnecessary match function executing each game loop and possibly lowering frame rate, I included this match within the propertyInitialise() enemy function that is only executed once upon the spawning of the enemy.

```
//matches correct enemy sprite object to this enemy
for(var sprite of enemySprites){
 if(this.type == sprite.name){
   this.spriteSheet = sprite;
 }
}
```

Finally, to render the enemy sprite frames I created a switch case statement in the render function of the enemy object that checks for the current direction of the enemy, and if they are stationary the stationary frame is drawn straight away.

```
switch(this.currentDirection){
    case "leftstatic":
      image(enemySpriteSheetImg, this.x + player.x, this.y, this.width,
this.height, this.spriteSheet.frame3.sx, this.spriteSheet.frame3.sy,
enemySprites[0].sWidth, enemySprites[0].sHeight);
      break;
    case "rightstatic":
      image(enemySpriteSheetImg, this.x + player.x, this.y, this.width,
this.height, this.spriteSheet.frame4.sx, this.spriteSheet.frame4.sy,
enemySprites[0].sWidth, enemySprites[0].sHeight);
      break;
```

For the case in which the enemy is moving, there are 2 options for frames to create a walking effect. These frames are chosen between using a currentFrame variable that is switched between 1 and 2 based on the x position of the enemy.

```
//change frame every time width of sprite has passed
        if( Math.floor(this.x) % Math.floor(this.width/3) == 0){
```

```
          if(this.currentFrame - 1 == 0){
            this.currentFrame = 2;
          }
          else{
            this.currentFrame = 1;
          }
        };

       //display one of the two moving sprite frames
        if(this.currentFrame == 1){
          image(enemySpriteSheetImg, this.x + player.x, this.y,
this.width, this.height, this.spriteSheet.frame1.sx,
this.spriteSheet.frame1.sy, enemySprites[0].sWidth,
enemySprites[0].sHeight);
        }
        else{
          image(enemySpriteSheetImg, this.x + player.x, this.y,
this.width, this.height, this.spriteSheet.frame2.sx,
this.spriteSheet.frame2.sy, enemySprites[0].sWidth,
enemySprites[0].sHeight);
        }
```

To implement the sprite animation of the player I used the exact same method as seen above in the enemy animation. I created a playerSprites array in the spritesheet.js script, which enables me to add different player sprites in the future easily, perhaps purchasable by the user.

The next goal of development will be to implement the usage of the player's weapon slots, where they will be able to switch weapons using the number keys and their weapon slots will be displayed on their HUD.

V1.0 Weapon Slot Usage

The first step to developing the weapon slot usage was to display each weapon slot onto the player HUD, and highlight the slot which they currently have equipped. To display the weapon slots on the player HUD I used a for loop which loops through the player's equipped weapons array and draws a box with the weapon sprite inside each slot. This loop was successful, and displayed the weapon slots like this:

The next step to creating weapon slot usage was to enable the player to change weapon slots using the corresponding key numbers (1,2,3,4,5). To save repeating code, I wrote a function which checks that the weapon slot the player has selected contains a weapon, and if it does the weapon slot is equipped.

```
//function that changes the player's equipped weapon slot
function changeWeaponSlot(slot){
 if(equippedWeapons[slot].name != null){
   currentWeaponSlot = slot;
 }
}
```

After writing this function all I needed to do was write keyIsDown p5 checks for each number key and include this function in the code that is executed upon a key press.

```
if(keyIsDown(49)){
   changeWeaponSlot(0);
 }
```

This implementation was a success and the player was now able to switch between their weapon slots and equip the weapon of their current slot. The correct highlighted slot was also updated on the player HUD. However, there was one bug which was occurring when the player switched weapons at the same time as reloading where the weapon the player switched to would begin to reload after the swap. To fix this bug, I made sure that the currentlyReloading variable was set to false when the player swapped weapons.

```
if(equippedWeapons[slot].name != null){
   currentWeaponSlot = slot;
   currentlyReloading = false;
 }
```

The next step of development will be to add a weapon selection menu screen before the player enters a level, where they can choose which of their owned weapons they would like to equip in their weapon slots to play with for that specific level.

V1.1 Weapon Selection Menu

To implement a weapon selection menu I first created a new script for the menus folder called weapon-selection-menu.js. This script will contain the code for the weapon selection page and its functionality.

In order for the player to be able to select from a list of their owned weapons, there must be an array that stores their owned weapons in the game. Therefore, I created an array that contains a list of all the weapons the player owns that can be added to when the player purchases a new weapon. For a new player, the default owned weapons are just the glock.

```
//stores a list of all the weapons owned by the player
var ownedWeapons = [ "Glock" ];
```

The next step was to create a screen that graphically displays the slots of the player and below the slots a list of the weapons that they can select to equip with their sprite. To simplify the functionality needed for the player to be able to click a weapon in and out of a slot, I created a weaponButton class that will draw a box with the weapon object as well as perform a click check for if the player has clicked within it. If the button is clicked, the weapon will either be equipped (if the player has a spare slot) or unequipped if it is already in a player slot.

Each game loop, the buttons are removed, rebuilt, and pushed into arrays based on if the weapon is equipped or if it is unequipped. After they are pushed into arrays, each array is looped through and rendered onto the screen.

```
unequippedButtons = [];
 //creates unequipped weapon buttons based off weapons unequipped and
owned by the player
 for(var i=0; i< ownedWeapons.length; i++){
    //if this owned weapon is not equipped
    if(!checkIfEquipped(ownedWeapons[i])){
     //create a button for the owned weapon
     unequippedButtons.push(new weaponButton(ownedWeapons[i],
"lightblue", (500 + unequippedButtons.length*100 +
unequippedButtons.length*20), 390, "equip", undefined));
    }
 }
//draws unequipped weapon options
 for(b of unequippedButtons){
   b.render();
 }
```

The result of this was a success, and the player is now able to equip/unequip weapons from their slots based off of their owned weapons.

However, upon adding more weapons to the list of weapons owned by the player, I discovered a bug which caused the new weapon buttons to continue to be displayed along one row of the available weapon buttons, eventually being positioned out of view on the right. To fix this bug, I created an if statement in the build loop of the buttons which caused the y position of buttons to be increased every 8 buttons, therefore being displayed in rows.

The next step of development will be to add a shop menu to the game, where the player can use cash that they accumulate from killing enemies to buy new weapons and ammo.

V1.2 Shop Menu

The development of the shop menu had many similarities to the development of the weapon selection menu. The first step to the development was to find a way in which to display a list of all the weapons on a screen with their price and below these weapons an option to buy their ammo type. In order to achieve this, I used the same method as for the weapon selection screen: creating button objects from a weapons array. In this case, I wanted there to be a button for each weapon available in the game, so the buttons are created from the weaponTypes array.

```
//creates shop buttons for weapons that the player doesn't own
 for(weapon of weaponTypes){
   if(ownedWeapons.includes(weapon.type)){
     //weapon already owned
     weapon.owned = true;
   }
   //create shop button
   shopButtons.push( new shopButton(weapon,500,200));
 }
```

As you can see above, the loop also marks down whether or not the player owns the weapon type. This will be helpful in preventing the player from buying the same weapon more than once.

The one major difference between the buttons in the shop menu and the weapon selection menu is that the shop menu requires each button to have several different actions (buy

weapon or buy ammo) whereas in the weapon selection menu each button only performs one action. To enable the button to perform several different actions, the shop button object renders 2 different button sections and performs 2 different click checks for the 2 rendered positions.

```
renderAmmoButton(){
   //draw box for ammo buy option
   fill("lightyellow");
   rect(this.x+60, this.y + 100, 220, 90);
   fill("blue");
   text(this.ammo.type, this.x+60, this.y + 95);
   text("Buy x30 £" + this.ammo.price, this.x+60, this.y + 130);
 }
```

```
renderWeaponButton(){

   //draw weapon image in a box
   fill("lightgreen");
   rect( this.x, this.y, 100, 100);
   image(weaponSpriteSheetImg, this.x, this.y, this.sprite.sWidth/1.8,
this.sprite.sHeight/1.8, this.sprite.sx, this.sprite.sy,
this.sprite.sWidth, this.sprite.sHeight);

    //draw buy box option
   fill("lightblue")
   rect( this.x + 110, this.y, 120, 100);
   fill("blue");
   if(this.weapon.owned == undefined){
     text("BUY", this.x + 110, this.y-10);
     text("£" + this.weapon.price, this.x + 110, this.y+20);
   }
   else{
     fill("red");
     text("OWNED", this.x + 110, this.y);
   }
   fill("lightgreen");
 }
```

If the user clicks the ‘buy’ button for a weapon, the game checks whether the player already owns the weapon, and if they don’t and have sufficient money to buy the weapon, the

weapon is added to the player's ownedWeapons array and the cost of the weapon is taken from their overall balance which is stored in a currentMoney variable.

```
//buy weapon if player has enough money and they don't already own the
weapon
     if(this.weapon.owned == undefined){
       if(currentMoney >= this.weapon.price){
         ownedWeapons.push(this.weapon.type);
         currentMoney -= this.weapon.price;
       }
     }
   }
```

If the user clicks the 'buy' button for ammo, the game makes sure the player has sufficient funds to buy 30x the ammo type, and the ammo is added to the player's ownedAmmo amount.

```
if(currentMoney >= this.ammo.price && this.ammo.type != "9mm"){

       for(var i=0; i < ownedAmmo.length; i++){
         if(ownedAmmo[i].type == this.ammo.type){
           ownedAmmo[i].amount += 30;
           currentMoney -= this.ammo.price;
         }
       }
     }
```

Testing

Development Testing

In order for the project to be successful upon completion, it is necessary to test each objective of the success criteria before the project is launched. Each objective from the success criteria has a series of tests that will be carried out to ensure that the objective is working as intended. Evidence of successful testing will be provided through screenshots, console logs or within the final video.

M1 Player Mechanics

The user should be able to make their player move left, right and jump around the level when they press certain keys on their keyboard

#	Description of Test	Expected Result	Result
1.1	Press A key on keyboard	Player will move left	Success
1.2	Press D key on keyboard	Player will move right	Success
1.3	Press W or Space key on keyboard	Player will jump and fall back to the floor	Success
1.4	Press left click on mouse	Player will fire a bullet if their weapon has ammo and fire rate timeout has passed	Success

M2 Menu Functionality

The user should be able to navigate through different menus by pressing buttons to select what they would like to view in the game.

#	Description of Test	Expected Result	Result
1.1	Open project page in browser	Main menu screen will be displayed	Success
1.2	Click mouse whilst hovered over a button on the screen	Menu screen will be switched and display a new screen depending on the button the player has pressed	Success
1.3	Click button with path of returning to main menu screen	Menu screen will be returned to the main menu screen	Success

M3 Level Selection Menu

The user should be able to choose between a screen of buttons displaying all levels in the game and if they have unlocked the level, play the level

#	Description of Test	Expected Result	Result

1.1	Load level selection menu screen	A screen with a heading describing the menu objective and buttons below displaying a list of the game levels	Success
1.2	Click on an unlocked level button	The screen will be switched to a weapon selection menu and the currentLevel is changed to the number of the selected level	Success
1.3	Click on a locked level button	No action will occur	Success

M4 Level Design

The game should include several levels with different wave sizes and enemy types.

#	Description of Test	Expected Result	Result
1.1	Enter into level 1	The player will be spawned into a level containing a tower that they can press e whilst next to to begin enemy wave spawn.	Success
1.2	Enter into each level	The player will be spawned into levels with different backgrounds	Success
1.3	Press e on the tower in each level	A wave of zombies will be spawned with harder enemy types and sizes the higher the level	Success

M5 Enemy AI and wave spawning

The enemy AI in the game should be able to track the player and follow them until they are within attack range, once they are within attack range they will shoot a projectile at the player or melee attack the player. Enemy waves will be spawned with specific player types and amounts at specific spawn locations.

#	Description of Test	Expected Result	Result
1.1	Player presses e on tower in level 1 to test enemy player tracking	Enemies will be spawned off the screen and begin to walk towards the player	Success
1.2	Wait for enemy to stop when it reaches its attack range from the player	Enemy will fire bullets at the player at its fire rate until the player moves out of the enemies attack range.	Success
1.3	Shoot bullet at the enemy and wait for the bullet to collide with the enemy	The enemies health bar above the enemy will be reduced by the amount of damage the bullet causes	Success
1.4	Call spawnWave function in console with enemy type parameter equal to basic and	Wave of enemy type specified will be spawned off screen and begin to target the player, once all enemies are	Success

	enemy amount parameter equal to 7	on the screen there should be 7 basic enemies	

M6 Player Health and Weapon HUD

The player should have a health value that can be changed when they are hit by a bullet. The player's health and equipped weapons should be displayed in the player HUD.

#	Description of Test	Expected Result	Result
1.1	Observe health bar on HUD whilst in game	The top left of the screen should display a health bar for the player with a health text heading and text on the bar showing the exact player health out of their max health	Success
1.2	Modify player.health value	The health bar should change to show that the player has taken damage	Success
1.3	Observe weapon slot HUD whilst in game	The top of the screen should contain 5 slots with images of the player's equipped weapons inside each slot	Success
1.4	Observe current weapon HUD whilst in game	The top of the screen should contain an image of the player's current weapon. The HUD should display the name, current ammo and max ammo of the weapon	Success
1.5	Press buttons 1-5 on keyboard whilst in game	When each key is pressed on the keyboard, the new current slot of the player will be highlighted and the current weapon data will be switched to the data of the weapon in the slot	Success

M7 Persistent Storage

Objective not attempted.

M8 Sprites and Graphics

The player and enemies in the game should have sprites that change depending on their movement and direction.

#	Description of Test	Expected Result	Result
1.1	Move player to the left	Player sprite will face left and switch between two walking sprites to make a walking animation	Success
1.2	Move player to the right	Player sprite will face right and switch between two walking sprites to make a walking animation	Success

1.3	Observe enemies travelling towards the player from the left and right	Enemy sprites will face towards the player and switch between two walking sprites to make a walking animation	Success
1.4	Observe enemies within attack range which are standing still	Enemy sprites will face towards the player and have a static sprite	Success

M9 Bullet Collision Detection

Bullets can be fired by the player and by the enemies, the bullets should only collide when they hit their specified target type.

#	Description of Test	Expected Result	Result
1.1	Shoot a bullet from player at an enemy	Bullet will be destroyed when it collides with any enemy and the enemy health bar will decrease	Success
1.2	Wait for enemy to fire a bullet at the player	Bullet will be destroyed when it collides with the player and the player health will decrease	Success
1.3	Shoot a bullet from the player towards the edge of the screen	The bullet will be destroyed (removed from the currentBullets array) once off the screen	Success

M10 Weapon Properties and Usage

Different weapons should have different behaviours when shot by the player. These behaviours include fire rate, reload speed, magazine size and bullet types.

#	Description of Test	Expected Result	Result
1.1	Reload “Glock” weapon type whilst equipped to check reload times	The weapon will reload in the reload time of the Glock and the weapon mag will be filled with ammo	Success
1.2	Hold down shoot button with “Glock” weapon type equipped to check fire rate	The weapon will fire bullets at the fire rate of the Glock and for each bullet fired the mag ammo will decrease by 1	Success
1.3	Fire a bullet from the Glock at the enemy to check bullet damage	The bullet will deal the amount of damage to the enemy equal to the damage of the Glock bullet type	Success
1.4	Attempt to reload “AK-47” weapon type with 0 spare AK ammo remaining	The weapon will not begin reloading because there is no spare ammo to reload	Success
1.5	Attempt to reload “AK-47”	The weapon mag will be reloaded with	Success

	weapon type which has a currently empty mag with 12 spare AK ammo remaining	the remaining spare bullets (12) and the spare AK ammo will go to 0	
1.6	Attempt to reload “AK-47” weapon type which has a half full mag with 1000 spare AK ammo remaining	The weapon mag will be reloaded up to its maximum mag size and the bullets that were added to the mag will be removed from the spare bullets	Success
1.7	Attempt to reload “AK-47” weapon type which has a half full mag with 5 spare AK ammo remaining	The weapon mag current ammo will be increased by 5 and the spare ammo will be changed to 0	Success
1.8	Switch between weapon slots 1-5 with 5 different weapon types equipped to check mag size	When the current weapon slot is changed the weapon HUD info will display the new equipped weapons mag size	Success

M11 Weapon Selection Menu

The player should be able to customise their equipped weapons slots and choose between their owned weapons before they enter into a level.

#	Description of Test	Expected Result	Result
1.1	Load weapon selection menu screen	A screen with 5 weapon slots displayed and below the weapon slots a list of the players owned weapons as buttons with the weapon images inside	Success
1.2	Click a weapon button from the owned weapons list with a spare weapon slot available	The weapon will be displayed in one of the 5 empty equipped weapons slots and will disappear from the owned weapons list	Success
1.3	Click a weapon button from the owned weapons list with no spare slots available	None	Success
1.4	Click a weapon slot which has a weapon equipped	Weapon slot will become empty and the weapon will appear in the owned weapons list	Success
1.5	Click an empty weapon slot	None	Success
1.6	Fill all 5 weapon slots with weapons	Play button will appear above the weapon slots and when clicked player will be transferred into the game with correct weapons equipped	Success
1.7	Use all owned weapons in weapon slots	Play button will appear above the weapon slots	Success

S1 Sound Effects

To improve the overall user experience and gameplay sound effects should be added when the user is navigating the menus and using different weapons in game. This objective was not attempted, however if the effects were to be implemented, the tests below would be appropriate.

#	Description of Test	Expected Result	Result
1.1	Load up the main menu screen of the game	Reclamation theme music plays in the background	Not attempted
1.2	Click menu button	Play a button sound effect	Not attempted
1.3	Switch between different weapons in game and shoot them each	Play different shooting sound effects for the different weapons	Not attempted
1.4	Reload a weapon in game	Play a reloading sound effect	Not attempted
1.5	Wait for player to run out of health in game	Play game over themed music	Not attempted

S2 Player Economy and Shop

The player should be able to earn in game currency through killing zombies and completing levels. This currency can be used in an in game shop to buy new weapons and more ammo.

#	Description of Test	Expected Result	Result
1.1	Kill a zombie in game	Reward player with the amount of cash that the zombie rewards and display earned cash message on screen	Success
1.2	Select shop from the game menu	Switches to shop menu screen which has buttons for each weapon with their price, a buy option (if not owned) and a button below where you can buy their ammo type.	Success
1.3	Press buy button on a weapon or ammo with insufficient cash	No action occurs	Success
1.4	Press buy button on a weapon or ammo with sufficient cash	if weapon the weapon is added to ownedWeapons for the player and if ammo their ownedAmmo is increased by the amount they bought, their current cash is decreased by the cost of the item	Success

S3 Enemy Variety

The game should have a decent amount of enemy types that have different speeds, fire rates, damage values, attack ranges and different sprites.

#	Description of Test	Expected Result	Result
1.1	Call the spawnWave function inside of a level for different enemy types and observe their appearance	Each wave should have different enemy sprites as the enemy type specified was different	Success
1.2	Call the spawnWave function and observe enemy speeds	Each wave should have different enemy speeds when they travel towards the player	Success
1.3	Call the spawnWave function and wait for the enemies to enter their attack range	Each wave should have a different attack range for the enemies in the wave	Success
1.4	Call the spawnWave function and observe enemy fire rate	Each wave of enemies should fire bullets at the player at different rates	Success
1.5	Call the spawnWave function and observe the damage dealt by each enemy attack	Each wave of enemies should deal different amounts of damage to the player per attack	Success

S4 Multiplayer Compatibility

Objective not attempted.

S5 Level Variety

The player should be able to play through several levels in the game to make the game playable for longer without being repetitive.

#	Description of Test	Expected Result	Result
1.1	Enter the level selection menu	A series of levels should be selectable	Success
1.2	Enter into each level from the selection screen	The levels should be functional and playable	Success

S6 Challenges and Additional Objectives

Objective not attempted.

S7 Settings Menu

Objective not attempted.

S8 Health Packs

Objective not attempted.

C1 Player Leaderboard

Objective not attempted.

C2 Speed Runs

Objective not attempted.

C3 Versus Game Mode

Objective not attempted.

C4 Abilities

Objective not attempted.

C5 Zombie Drops

Objective not attempted.

W1 Microtransactions

Objective not attempted.

W2 In Game Voice Chat

Objective not attempted.

W3 Controller Compatibility

Objective not attempted.

Evaluation

Now that the development of the project has been completed and I have finished testing the different features, It is necessary to get feedback from the stakeholders of my project where they test and evaluate it. The testing of the stakeholders is not intended to be as thorough and detailed as my personal testing, but is more centered towards gaging how they feel about the specific features, whether they enjoyed them, and if they think any improvements could be made.

To get the stakeholders feedback, I will design a simple test plan for the stakeholders to follow and provide their comments inside as well as providing them with a link where they can access the finished project.

Stakeholder Testing

Stage 1 of development for 'Reclamation' has now been completed and thank you so much for your contributions towards the project! It is now important for me to get your feedback on how you feel about the current state of the project so I can work on future improvements to gameplay.

You can access the game at the following link:
https://reclamation-singleplayer.jackecuyer.repl.co/
As you play along the game, please provide your feedback for the following topics shown below and once you have finished playing the game as much as you want, please email me your responses. Your comments are very valuable towards the evaluation of the project and will ultimately be used to help plan out future steps of development and improve features of the game. I welcome your criticisms of the project, so feel free to leave a comment about a feature that you didn't necessarily like or enjoy and the reasoning behind your opinions.

Thanks so much for working with me through this journey of the Reclamation project and I hope to remain in contact with you for the possibility of future endeavours.

Jack.

Starting a Game

The game is laid out with multiple menu screens that should be easy to follow and get you into your first level of the game.

Action	Comment
Navigate the menu screens to select level 1 of the game	
Do you think that the menu screens are intuitive to navigate and the weapon selection screen is understandable?	

Overall Comment:

Level Gameplay

The core gameplay occurs in levels where you must fight oncoming waves of zombies that get progressively stronger the further through the game you progress.

Action	Comment
Enter a level of the game	
Is the player HUD easily readable and clear?	
What do you think of the graphics?	
Was the gameplay balanced (not too easy or difficult)	
Do you feel the core mechanics of the game (movement of the player) are well made?	
Do you feel the weapons are fun to use?	

Overall Comment:

Using the Shop

The main way the player progresses through the game is by playing increasingly difficult levels, however to be able to compete with the enemies at the higher levels, you must have stronger weapons that you have to save up money for and buy in the in game shop.

Action	Comment
Kill enough zombies and accumulate enough money to buy a new weapon in the item shop	
Do you feel the shop is well laid out?	
Do you feel that having to buy ammo for stronger weapons is a good idea?	

Overall Comment:

Results of Stakeholder Testing

I received the responses of the stakeholders via email and have created a table below which collates the results of the testing for all of the stakeholders.

Test	Response
Impressions of menu functionality and layout	1. The menus are easy to navigate but they would be more interesting with some images in the background 2. I liked the buttons but I think it would be nice if the buttons changed colour when i hovered over them. 3. I clicked through the screens quickly, the buttons describe the next menus well.
Is the weapon selection screen easy to use?	1. It's awesome that I can customise what weapons I want to use for different levels! 2. It took me a second to figure out how to unequip a weapon after I put it into a slot. 3. It's a nice feature that I can change the order of weapons in the slots.
GAME NAVIGATION SUMMARY	1. The menus for the game are very easy to follow and I especially like the weapon selection screen. I would like them to be a bit more colourful though. 2. The menus switch instantly and easily, I had no trouble finding which page I wanted to go to. 3. I thought that the menus were very simple to navigate, I think they could be more interesting at times.
Entering a level - clear and intuitive?	1. After the selection menu I'm put straight into the level without any difficulties. 2. I followed the menus easily and am spawned into the level straight away. 3. I got put straight into level 1, but there was no description for the objective.
Player HUD impressions - simple and easy to read?	1. I really like the player HUD. The health bar is easy to interpret, weapon info is clear to read and I like how the slot I have equipped is highlighted. 2. The weapon image is helpful because it lets me see what I'm using straight away. The rest of the HUD is easily understandable. 3. I found it easy to read the HUD, it's a good

	size and doesn’t block the actual gameplay at all.
What do you think of the graphics?	1. The graphics are relatively simple but I like the pixelated feel of the game, the background to the level is well fitted to the page. 2. The movement of the enemies and the player are cool, there aren't many objects in the foreground though. 3. The graphics are basic but I like how distinct the enemies are from each other.
Is the gameplay balanced?	1. The levels are definitely a challenge, but I think that the difficulty is well balanced. 2. The first level is really easy, but they definitely get harder the further along you get. 3. The weapon prices are fair for their strength, and the enemies aren’t impossible to beat.
Impressions of the core mechanics - player movement	1. I like the WAD movement controls because they are the same as other games that I play. 2. It’s cool that I can also use space to jump because I don’t normally use W for it. 3. I wish there was a way I could change the keys to the arrow keys because I normally use my other hand to move around.
Weapon impressions - fun to use?	1. The variety of weapons to choose from is so cool, it definitely encourages me to play more to unlock the different guns. 2. Some of the assault rifles are quite similar in their usage, but I think the variety of other options definitely makes up for it. 3. I like how the stronger weapons are more expensive and require more playtime, it definitely makes me want to play more to unlock and use them.
LEVEL GAMEPLAY SUMMARY	1. I love the gameplay of the game, the weapons are great and stop the game from getting repetitive. There are a nice variety of enemies to fight and the game isn’t too easy. 2. The overall playing experience is fun and interesting, I want to continue playing to find and fight the harder enemies. 3. The levels are sometimes quite hard but I know that if I keep playing I can buy better weapons to make it easier.
Economy impressions - accumulating money	1. This is my favourite feature in the game, it definitely keeps me playing knowing that I still have different weapons to unlock! 2. Sometimes it feels like a bit of a grind having to play the same level again to earn more money, but I overall like this feature. 3. I think it’s cool how even if I’m struggling to finish a level, it has still benefitted me to play it as it means I’m saving up money to buy new guns.

Impressions of shop layout	1. The ammo is clearly shown under each weapon so I know which type I need to buy for each weapon. 2. I like how I can't make the mistake of buying the same weapon twice as it is clear when I already own a weapon. 3. I think it would be cool if there was some type of animation that happens when you buy a new weapon, it would definitely make the process more rewarding.
Impressions of ammo purchase feature - fair?	1. It is sometimes a bit frustrating if I can't afford much ammo for one of my stronger weapons, but I understand that it makes the gameplay more balanced. 2. I like how I can't just buy a strong weapon and use it to destroy all the enemies easily because I have to take notice of how much ammo I have left. 3. I wish there were more weapon types with unlimited ammo or if there was an option to buy unlimited ammo eventually. The grind is fun but after a while I wanted to just be able to use each weapon as much as I want.
SHOP SUMMARY	1. The shop is great, it makes the game so much more interesting and makes me want to play for longer. 2. I really like the shop, it definitely creates a completely different challenge and objective within the game. 3. The shop is cool, but sometimes I feel like it can stop me from just playing and enjoying the gameplay because I have to worry about my weapons and their ammo.

Success of the Project

To evaluate how successful the development of the project was, I have evaluated each MOSCOW feature of the game below.

M1 Player Mechanics

This core objective of the game was met completely as evidenced by tests M1.1 to M1.4. The stakeholders were generally happy with the keys used to control the player ["I like the WAD movement controls because they are the same as other games that I play", "It's cool that I can also use space to jump because I don't normally use W for it."] because they were the same controls they had used previously in other games therefore they didn't have to learn different controls.

However, the 3rd stakeholder said "I wish there was a way I could change the keys to the arrow keys because I normally use my other hand to move around." This shows that although the majority of players would be familiar with the controls, some players have

different preferences. To solve this issue, I could implement an option in the settings menu of the game where the player could choose whatever keys on the keyboard they would like to play with. This would be relatively simple to develop, I could detect a keypress from the player when they choose to change a specific control, and this key would be stored in a variable that is used in keypress checks instead of the default keys of the game.

M2+M3 Menu Functionality

These objectives were completed successfully (shown in tests M2.1 to M2.3 and M3.1 to M3.3). The stakeholders were generally happily with the menu screens and thought that they were easy to navigate [“The menus are easy to navigate but they would be more interesting with some images in the background”, “I liked the buttons but I think it would be nice if the buttons changed colour when i hovered over them”, “I clicked through the screens quickly, the buttons describe the next menus well.”] The only improvement suggested by the stakeholders was to improve the graphics of the menu screens by adding some more images and animations to the buttons, which would be a simple addition to the game using the image() p5 function. The menu functionality was satisfying to program because the gameMode variable simplified the modularisation of the project (made it easy to separate code sections into different scripts and functions).

M4+S5 Level Design & Variety

This objective was completed successfully (shown in tests M4.1 to M4.3 and tests S5.1 to S5.2). Although level design is definitely a section of development I struggle with, I believe that the variety of enemy types simplified the development process and gameplay because it allowed me to essentially increase the difficulty (using harder enemy types) and amount of enemies as the levels progress.

The only comment suggesting an improvement in level design was from the 3rd stakeholder: “I got put straight into level 1, but there was no description for the objective.” The best way of responding to the stakeholders issue would be to add a timed sequence at the beginning of each level, which scrolls through headings describing what the player must do within the level before they begin playing. This could be implemented using the millis() p5 function.

M5+S3 Enemy AI and Wave Spawning

These objectives were met successfully (shown in tests M5.1 to M5.4 and S3.1 to S3.5) and the stakeholders believe that the enemy spawning and difficulty was balanced throughout the levels. [“The levels are definitely a challenge, but I think that the difficulty is well balanced.”, “The first level is really easy, but they definitely get harder the further along you get.”, “The weapon prices are fair for their strength, and the enemies aren’t impossible to beat.”].

The development of enemy AI was one of the sections of programming I was most proud of because of the simplification it enabled when adding to the game further down the development process. Coding the enemy AI script into one specific class enabled me to add many different enemy types to the game with just one line of code each that declared all the properties of the new enemy type. The variety of property types for the enemy allow each

enemy to feel unique, and these properties included speed, fire rate, attack range, attack type and bullet type.

M6 Player Health and Weapon HUD

This objective was met completely (evidenced by tests M6.1 to M6.5) and the feedback from the stakeholders towards the player HUD were very positive. [“I really like the player HUD. The health bar is easy to interpret, weapon info is clear to read and I like how the slot I have equipped is highlighted.”, “The weapon image is helpful because it lets me see what I’m using straight away. The rest of the HUD is easily understandable.”, ”I found it easy to read the HUD, it’s a good size and doesn’t block the actual gameplay at all.”].

I was satisfied with the feedback I received from the stakeholders because it described perfectly the goal of the player HUD, which was to make it easy and intuitive for the player to view their statistics and options whilst they are playing a level. The process of creating a sprite sheet script that contained details of each individual weapon sprite was quite tedious, however was definitely worth the end result.

M7 Persistent Storage

This objective was not attempted however I believe that one of the most important next stages for the project would be to implement persistent storage.

Basic persistent storage would be relatively simple to implement, because the user could create an account and the data which has previously been stored in variables on the client’s device could simply be transferred to a database and updated at regular intervals of gameplay.

On the other hand, if persistent storage were to be implemented in the future I believe it would pose several major issues for the user because it comes with the risk of a user forgetting their details and having to play the game from the beginning on a new account.

This would be one of the most difficult problems to tackle in the future because it would require the player to add another parameter when they create their account such as a security question or email verification for the event of needing to recover or change their account details. As the game is relatively simple and would not take too long for a player to regain their progress in its current state,I believe that this problem would be a low priority in future development as the current solution of creating a new account is already a possibility.

M8 Sprites and Graphics

This objective was met successfully (shown by tests M8.1 to M8.4) and the stakeholders were generally happy with the look of the game. [“The graphics are relatively simple but I like the pixelated feel of the game, the background to the level is well fitted to the page.”, “The movement of the enemies and the player are cool, there aren't many objects in the foreground though.”, “The graphics are basic but I like how distinct the enemies are from each other.”].

Overall, I was happy with how the graphics turned out, and the ‘pixelated feel’ is the exact look I was going for. I concluded that a 2d game shouldn’t be too focused on graphic details, and should instead feel ‘basic’, which I believe is a target that I met successfully judging by the feedback of the stakeholders.

The one criticism by the 3rd stakeholder of there being a lack of objects in the foreground of gameplay is an issue that could be relatively easily addressed by creating or downloading additional object sprites, creating a sprite sheet script, and coding these sprites into the generateLevel() functions using the image() function.

M9 Bullet Collision Detection

This objective was met completely (shown by tests M9.1 to M9.3) and the development of this feature was relatively easy for me because I have had previous experience in other projects using the same methods of collision detection. The one difference with collision detection in this project that made coding slightly more difficult was the necessary inclusion of the player.x variable offset for each position in the game.

M10 Weapon Properties and Usage

This objective was met successfully (shown by tests M10.1 to M10.8) and the feedback from the stakeholders towards the ability to use different weapon types was overwhelmingly positive [“The variety of weapons to choose from is so cool, it definitely encourages me to play more to unlock the different guns.”, “Some of the assault rifles are quite similar in their usage, but I think the variety of other options definitely makes up for it.”, “I like how the stronger weapons are more expensive and require more playtime, it definitely makes me want to play more to unlock and use them.”].

This was another area of development I believe was very successful because similarly to the programming of the enemy AI, the format of weapon usage coding enabled me to implement many different weapon types into the game with just one line of code each, and each weapon is able to be distinguished from the others due to a variety of different properties that the weapons can have including fire rate, reload speed and bullet type.

In addition, the method of development I chose which involved separating bullet types and weapon types into different arrays that are independent of each other worked really well because it allowed the modification of each section exclusively from the other.This meant that properties of bullets or weapons could be changed without one affecting the other.

M11 Weapon Selection Menu

This objective was met completely (shown by tests M11.1 to M11.7) and the feedback from the stakeholders was generally positive [“It’s awesome that I can customise what weapons I want to use for different levels!”, “It’s a nice feature that I can change the order of weapons in the slots.”].

The only criticism by a stakeholder was that there were no instructions on how to use the menu and it took them a little while “to figure out how to unequip a weapon after I put it into a

slot.” This would be an easy problem to solve by adding a few headings in the free space of the menu screen using the p5 text() function which describe how to use each feature.

I believe that the method of development I chose for this specific menu was a success because the creation of button objects based off the player’s ownedWeapons array enables the program to automatically generate the correct buttons for when the player equips or unequips a weapon, and the equipped slots are also updated successfully.

S1 Sound Effects

This objective was not successfully met. However, if I were to implement the different music which would play on the main menu screen or after the player dies and enters the game over menu in the future, I would either have to play the sound data for the music when the player enters the menu and stop it when they exit the menu, these changes would be detected by waiting for a change in the gameMode scene management variable. The other method I could use would be to create a cutscene effect using the p5 millis() function which puts the player into a set sequence of text headings and stop the music once the sequence finishes.

S2 Player Economy and Shop

This objective was met successfully (shown by tests S2.1 to S2.4) and the feedback received from the stakeholders was positive [“The shop is great, it makes the game so much more interesting and makes me want to play for longer.”,”I really like the shop, it definitely creates a completely different challenge and objective within the game.”, “The shop is cool, but sometimes I feel like it can stop me from just playing and enjoying the gameplay because I have to worry about my weapons and their ammo.”].

The one issue raised by the 3rd stakeholder was that the economic side of the game prevented them from having the freedom to “enjoy the gameplay”. To address this issue in the future, I could implement a ‘freeplay’ feature into the game that when selected would temporarily disable updates to the player’s account data and they would be given unlimited money to purchase and use all the weapons as well as having “unlimited” ammo values for every ammo type. This would also resolve the issue raised by the stakeholder of wishing “there were more weapon types with unlimited ammo or if there was an option to buy unlimited ammo eventually”.

Assessment of Usability Features

The game is intended to be simple and intuitive to navigate, with the main focus of the project being the actual level gameplay. Therefore, the user interface is kept relatively basic so that it does not cause players any confusion and they can access the gameplay easily.

There are 3 core usability components including:

- Menu Functionality
- Player Movement
- Weapon Usage

However, the menu functionality branches into multiple usability features, because the player is able to perform more complex actions on the weapon selection and shop menus.

Basic Menu Functionality

The basic menu functionality consists of buttons objects rendered from different button menu arrays depending on the current gameMode scene of the game. Extensive testing has shown that these buttons work as intended, with every button click taking the user to the menu screen described by the button and the scene changing accordingly.

The stakeholders were generally happy with the menu screens, and every stakeholder made a comment about the simplicity of the navigation saying it is “easy to navigate” and “quick”. The only improvement suggested was an aesthetic addition (stated in the success of the project: M2 + M3). Therefore, in conclusion the menu navigation was a success because it achieved its primary goal of being simplistic and intuitive.

Weapon Selection Menu Functionality

The weapon selection menu generated a button with a slot like appearance for each available equippable weapon. The buttons automatically format next to each other and move onto new rows after a certain amount of buttons to ensure a symmetrical appearance to the menu.

The only suggested improvement to this menu was for there to be a distinct description of what the player is able to do in the menu (i.e click on an unequipped button to equip it and click on an equipped weapon slot to unequip the weapon in that slot) because a stakeholder struggled to “figure out how to unequip a weapon after I put it into a slot.” This issue was addressed in the success of the project: M11 section.

Shop Menu Functionality

The shop menu generated buttons for each purchasable weapon type with a buy option listing the price of the weapon. Below the weapon buy option, an ammo purchase button is also generated matching the weapon type above. The purchase buttons are formatted into rows once a certain amount of buttons have been generated to prevent buttons from being off the screen and inaccessible by the user.

The stakeholders were generally happy with the shop menu usage saying “The ammo is clearly shown” and that the menu is designed in a way that prevents the user from “making the mistake of buying the same weapon twice.”

The only improvement to the menu suggested by a stakeholder was the addition of an “animation when you buy a new weapon” because they believe that it would “make the process more rewarding”. This is a great idea and could be implemented into the game using a timed sequence with the millis() function. For example, the weapon image could get progressively larger by using the map() p5 function to calculate an image scale value and after the sprite reaches a certain size it could fade away using the tint() p5 function.

Player Movement

The objective for player movement usability was for the controls to be simple and intuitive. To achieve this, I decided on using WAD + Space controls because they are the controls most commonly used in other games and the user can transfer over their muscle memory. These controls are detected in the main game loop through keycode detection, and execute player object functions.

Although this method was successful for some stakeholders, saying that they “like the WAD movement controls because they are the same as other games” they play and think “It’s cool” that they “can also use space to jump” because they “don’t normally use W for it.” One stakeholder disliked the default movement button options, and wished “there was a way I could change the keys”. This issue was addressed in the success of the project: M1 section.

Development for player movement was a more complex process than I initially expected, because the player had to stay on the center of the canvas at all times, and the objects had to move around the player based on the x position of the player.

Weapon Usage

The goal of weapon usage was for the player to be able to use the weapons as easily as possible, whilst being able to select from a variety of different weapon types. To simplify the controls for player shooting, instead of the player having to re-click the mouse every time they want to fire the weapon, they can hold the mouse down and the weapon will fire at its assigned weapon type whilst there is still ammo remaining in the magazine.

To achieve this method of firing the weapons, I used the p5 millis() function to calculate the time that had passed since the player had last fired the weapon, and if the assigned fire rate time had passed, then the weapon would trigger the shooting validation code.

In addition, the player is able to switch between their weapon slots to change weapons using the first 5 number keys on the keyboard. To make the controls intuitive, each weapon slot on the weapon HUD has its appropriate key number stated on it. The feedback from the stakeholders towards the weapon switching was positive, and a stakeholder commented that they “like how the slot I have equipped is highlighted.”

The weapon slot HUD display was achieved by writing a for loop which iterated through the list of weapons that the player had equipped whilst drawing a numbered box on the canvas each time an iteration has passed. When a weapon is detected in the weapon list, its sprite is drawn within the box.

Limitations of the Project

The core limitation of the project in its current game state is that there is no interaction between different players of the game. Users have no way to compare their statistics, friend each other or play with each other. The most simple way for players to interact with each other in a way that provides some additional entertainment value would be for players to be able to view their statistics on a global leaderboard.

This would be a relatively simple feature to implement into the game once persistent storage is implemented because the database containing the different player statistics could simply be accessed (and even compared) by an SQL query and then the data could be displayed on a menu screen using p5 text manipulation.

Further Development

There are many different features that I would like to implement into the game, and most of them are Could objectives (although some are extensive Should objectives) that were not attempted from the MOSCOW objective plan. The largest scale feature that I could add to the game would be multiplayer compatibility (S4) where the player has the option to search for a game and enter a level with another player. Once the player is able to play with another user, I could add various features such as the ability for the players to add each other as friends and invite one another to a game in the future without having to search randomly.

Multiplayer compatibility would definitely be a challenge to develop in the game's current state because of how the level is generated around the location of one player. I predict that if this feature was to be attempted, there would be many testing issues with objects in the levels being displayed at different locations for different clients.

Another feature that I hadn't previously considered implementing is the idea of a 'survival' gamemode, where the player enters an infinite level and has to fight off increasingly difficult waves of zombies until they die. This would work excellently with the implementation of a leaderboard system, because the players would be able to compete for the 'highest wave' score on the leaderboard.

Appendix - Code

Index.html

```
<!DOCTYPE html>
<html>
 <head>
   <meta charset="utf-8">
   <meta name="viewport" content="width=device-width">
   <title>Reclamation</title>
   <link href="style.css" rel="stylesheet" type="text/css" />
   <script
src="https://cdnjs.cloudflare.com/ajax/libs/p5.js/0.10.2/p5.js"
type="text/javascript"></script>
 </head>
 <body>
   <script src="spritesheets.js"></script>
   <script src="main.js"></script>
   <script src="button.js"></script>
   <script src="menus/main-menu.js"></script>
   <script src="menus/singleplayer-menu.js"></script>
   <script src="singleplayer-game.js"></script>
   <script src="menus/level-select-menu.js"></script>
   <script src="menus/pause-menu.js"></script>
   <script src="menus/weapon-select-menu.js"></script>
   <script src="menus/shop-menu.js"></script>
   <script src="player.js"></script>
   <script src="bullet.js"></script>
   <script src="enemy.js"></script>

 </body>
</html>
```

Style.css

```
/* removes the cursor from the webpage */
* { cursor: none; }
```

Main.js

```
//preloads all resources before the game is run.
function preload(){

 //loading all images that will be used in the game
 defaultCrosshair = loadImage('images/default-crosshair.png');
 cursorImg = loadImage('images/cursor-image.png');
 enemySpriteSheetImg = loadImage('images/zombie-sprite-sheet.png');
 lvl1bg = loadImage('images/lvl1-bg.jpg');
 treeSprite = loadImage('images/tree-sprite.png');
 weaponSpriteSheetImg = loadImage('images/weapon-sprite-sheet.png');

};

//initialises all the different menu screens
var mainMenuButtons = [];
var singlePlayerMenuButtons = [];
var playerShopButtons = [];
var pauseButtons = [];
var levelSelectButtons = [];
var weaponSelectButtons = [];
var gameMode = 'mainMenu'; //stores the current gamemode scene
var buttons = []; //stores the current buttons that are being displayed

var equippedButtons = [];
var unequippedButtons = [];
var shopButtons = [];

//initialises singleplayer gamemode variables
var player; //stores the main player object
var floorHeight; //stores the floor height for the level
var currentLevel; //stores the current level the player has chosen
var currentBullets; //creates an array that will store all the bullet
objects currently in the game
var currentEnemies; //creates an array that will store all the enemy
objects currently in the game
var timeLastFired; //stores the program time in which the player last
fired a bullet
```

```
var reloadStartTime; //stores the program time in which the player
started to reload their weapon
var currentlyReloading; //stores if the player is in the process of
reloading their weapon or not


//stores the weapons that the player has currently chosen to equip
before going into a level
var equippedWeapons = [
 {name: null, ammoInMag: 0},
 {name: null, ammoInMag: 0},
 {name: null, ammoInMag: 0},
 {name: null, ammoInMag: 0},
 {name: null, ammoInMag: 0}
]

//stores the current weapon slot the player has equipped
var currentWeaponSlot = 0;

//stores a list of all the weapons owned by the player
var ownedWeapons = [ "Glock"];

//stores the amount of ammo that the player owns for each ammo type
var ownedAmmo = [
 { type: "9mm", amount: 'unlimited'},
 { type: "44 Magnum", amount: 0},
 { type: "7.62mm", amount: 100},
 { type: "5.56mm", amount: 0},
 { type: "8mm", amount: 0},
 { type: ".45 ACP", amount: 0},
 { type: "M18 Grenade", amount: 0},
 { type: "7mm", amount: 0},
 { type: "50 BMG", amount: 0}
];

//initialises array that stores all weapons available to unlock in the
game
var weaponTypes = [
 { type: "Glock", bullet: "9mm", magSize: 7, fireRate: 1, reloadSpeed:
1, price: 0},
```

```
 { type: "Magnum", bullet: "44 Magnum", magSize: 10, fireRate: 1.5,
reloadSpeed: 3, price: 500},
 { type: "AK-47", bullet: "7.62mm", magSize: 30, fireRate: 0.25,
reloadSpeed: 3, price: 3000},
 { type: "Scar", bullet: "5.56mm", magSize: 25, fireRate: 0.6,
reloadSpeed: 5, price: 5000},
 { type: "Minigun", bullet: "8mm", magSize: 75, fireRate: 0.1,
reloadSpeed: 15, price: 99999},
 { type: "Uzi", bullet: ".45 ACP", magSize: 30, fireRate: 0.25,
reloadSpeed: 6, price: 1000},
 { type: "RG-6 Grenade Launcher", bullet: "M18 Grenade", magsize: 3,
fireRate: 4, reloadSpeed: 10, price: 50000},
 { type: "Hunting Rifle", bullet: "7mm", magSize: 1, fireRate: 0,
reloadSpeed: 3, price: 8000},
 { type: "M24 Sniper", bullet: "50 BMG", magSize: 1, fireRate: 0,
reloadSpeed: 3, price: 35000},
 { type: "Machine Pistol", bullet: "9mm", magSize: 15, fireRate: 0.3,
reloadSpeed: 5, price: 1000},
 { type: "M16A1", bullet: "7.62mm", magSize: 30, fireRate: 0.5,
reloadSpeed: 5, price: 5500},
 { type: "Thompson", bullet: ".45 ACP", magSize: 25, fireRate: 0.3,
reloadSpeed: 6, price: 2350},
 { type: "Vector", bullet: ".45 ACP", magSize: 35, fireRate: 0.35,
reloadSpeed: 4, price: 3400}
];


//initialises array that stores each bullet object with its values
var bulletTypes = [
 { type: "9mm", speed: 20, size: 5, colour: "white", damage: 10, price:
0 },
 { type: "44 Magnum", speed: 25, size: 10, colour: "white", damage: 20,
price: 10},
 { type: "7.62mm", speed: 15, size: 7.5, colour: "white", damage: 4,
price: 25},
 { type: "5.56mm", speed: 13, size: 7, colour: "white", damage: 5,
price: 30},
 { type: "8mm", speed: 30, size: 8, colour: "white", damage: 5, price:
50},
```

```
 { type: ".45 ACP", speed: 30, size: 5, colour: "white", damage: 3,
price: 15},
 { type: "M18 Grenade", speed: 10, size: 20, colour: "red", damage: 30,
price: 100},
 { type: "7mm", speed: 40, size: 15, colour: "white", damage: 100,
price: 30},
 { type: "50 BMG", speed: 50, size: 20, colour: "white", damage: 150,
price: 400},
 { type: "4.6mm", speed: 20, size: 5, colour: "white", damage: 10,
price: 10}
];

//initialises array that stores each enemy object type with its values
var enemyTypes = [
 { type: "basic", speed: 1, health: 30, spriteSheet: null,
attackDistance: 100, attackType: 'projectile', attackSpeed: 2, reward:
10},
 { type: "speedy", speed: 3, health: 20, spriteSheet: null,
attackDistance: 300, attackType: 'projectile', attackSpeed: 1, reward:
25}
]

var currentMoney;

//sets up different properties before the draw loop is run.
function setup(){

 currentMoney = 10000;

 //creates the game canvas that sets the boundaries for every object
drawn on the screen
 canvas = createCanvas(1905, 917);

 //creates the buttons and adds them to the different menu screens
 mainMenuButtons = [
   new Button("Single Player", 952.5, 300, 600, 100, () => { gameMode =
'singlePlayerMenu'; buttons = singlePlayerMenuButtons }),
   new Button("Shop", 952.5, 450, 600, 100, () => { gameMode =
'shopMenu'; buttons = playerShopButtons })
 ]
```

```
 singlePlayerMenuButtons = [
   new Button("Play", 952.5, 300, 600, 100, () => { gameMode =
'levelSelectMenu'; buttons = levelSelectButtons;
resetEquippedWeapons()}),
   new Button("Main Menu", 952.5, 450, 600, 100, () => { gameMode =
'mainMenu'; buttons = mainMenuButtons })
 ]

 inGameButtons = [
   new Button("Pause", 1830, 50, 100, 50, () => { gameMode =
'pauseMenu'; buttons = pauseButtons })
 ]
  pauseButtons = [
   new Button("Resume", 952.5, 300, 600, 100, () => { gameMode =
'singlePlayerGame'; buttons = inGameButtons }),
   new Button("Quit Game", 952.5, 450, 600, 100, () => { gameMode =
'mainMenu'; buttons = mainMenuButtons })
 ]

 settingsButtons = [
   new Button("Sound", 500, 300, 300, 100, () => {/* This will mute /
unmute the game */ })
 ]

 levelSelectButtons = [
   new Button("Level 1", 300, 200, 150, 150, () => { gameMode =
'weaponSelectMenu'; currentLevel = 1; buttons = weaponSelectButtons}),
   new Button("Main Menu", 952.5, 800, 300, 50, () => { gameMode =
'mainMenu'; buttons = mainMenuButtons})
 ]

 weaponSelectButtons = [
   new Button("Main Menu", 952.5, 800, 300, 50, () => { gameMode =
'mainMenu'; buttons = mainMenuButtons})
 ]

 playerShopButtons = [
   new Button("Main Menu", 200, 50, 300, 50, () => { gameMode =
'mainMenu'; buttons = mainMenuButtons})
```

```
 ]
 //puts the player at the main menu when they initially start the game
 buttons = mainMenuButtons;

 //settings for the alignment of anything drawn on the screen using p5
 rectMode(CENTER);
 textAlign(CENTER);
 imageMode(CENTER);
 ellipseMode(CENTER);

}

//executed every time the player presses their mouse
function mousePressed(){

 //loops through each button on screen and checks if the player has
clicked each button
 for(b of buttons){
   b.clicked();
 }

 //if in the weapon select menu, button check the weapon slot buttons
 if(gameMode == "weaponSelectMenu"){
   //check for click event on each button
   for(b of equippedButtons){
     b.clickCheck();
   }
   for(b of unequippedButtons){
     b.clickCheck();
   }
 }

 if(gameMode == "shopMenu"){
   //check for click event on each button
   for(b of shopButtons){
     b.weaponBuyClickCheck();
     b.ammoBuyClickCheck();
   }

 }
```

```
}

//main draw loop for the game that executes every frame.
function draw(){

 //clears the canvas and sets background
 background("black");
  //calls a scene script depending on the current gameMode the player
is on
 switch(gameMode){

   case 'mainMenu': mainMenu();
     break;

   case 'singlePlayerMenu': singlePlayerMenu();
     break;

   case 'singlePlayerGame': singlePlayerUpdate(); //updates the
singleplayer environment
     break;

   case 'levelSelectMenu': levelSelectMenu();
     break;

   case 'pauseMenu': pauseMenu();
     break;

   case 'settingsMenu': settingsMenu();
     break;

   case 'playerShop': playerShop();
     break;

   case 'weaponSelectMenu': weaponSelectMenu();
     break;

   case 'shopMenu': shopMenu();
     break;
```

```
 }
}

//function that resets the weapons the player has equipped
function resetEquippedWeapons(){
 equippedWeapons = [
 {name: null, ammoInMag: 0},
 {name: null, ammoInMag: 0},
 {name: null, ammoInMag: 0},
 {name: null, ammoInMag: 0},
 {name: null, ammoInMag: 0}
 ]
}
```

Singleplayer-game.js

```
//script for the singleplayer gameplay mode

//this function is executed once when the player chooses a level on
singleplayer, to initialise variables.
function singlePlayerInitialise(){

 floorHeight = 800;
 player = new Player(0, floorHeight-50);
 currentBullets = [];
 currentEnemies = [];
 timeLastFired = 0;
 reloadStartTime = 0;
 currentlyReloading = false;

}

//this function is executed every draw loop to update the level
function singlePlayerUpdate(){

 // //if health goes to 0 return to main menu
 // if(player.health <= 0){
 //   gameMode = "mainMenu";
 //   buttons = mainMenuButtons;
 // }

 //checks the current level that the player has selected and generates
the level from the corresponding level function
 switch(currentLevel){

   case 1:  generateLevel1();
     break;

 }

 //draws all the buttons for the current scene on the screen
 for(b of buttons){
   b.render();
 }
```

```
 //draws the floor of the level
 fill("lightgreen");
 rect(width/2, floorHeight+7.5, 1905, 15);

 //updates and draws the current bullets on the screen by looping
through each bullet in the array
 for(var i=0; i <= currentBullets.length-1 ; i++){

   currentBullets[i].collisionCheck();

   //removes bullets from the game if they are off the canvas / hit the
floor
   if(currentBullets[i].x > 1930 || currentBullets[i].x < -30  ||
currentBullets[i].y > floorHeight || currentBullets[i].y < -30){
     currentBullets.splice(i,1);
   }

   //removes bullets from the game if they have collided with their
target
   else if(currentBullets[i].destroyed){
     currentBullets.splice(i,1);
   }

   else{
     //if they are on the canvas, they are updated and drawn
     currentBullets[i].update();
     currentBullets[i].render();
   }
 }

 //updates and draws the current enemies on the screen by looping
through each bullet in the array
 for(var j=0; j <= currentEnemies.length-1; j++){

   //removes enemies from the game once they reach 0 health
   if(currentEnemies[j].health <= 0){
     //rewards player with cash for killing enemy
     currentMoney += currentEnemies[j].reward;
     currentEnemies.splice(j,1);
```

```
  }

  else{

    //updates the enemy
    currentEnemies[j].travelDistanceCheck();

    //renders the enemy
    currentEnemies[j].render();

  }
}

//checks for specific key presses
if(keyIsDown(65)){ //if 'a' key is pressed
  player.left(); //player moves left
}

if(keyIsDown(68)){ //if 'd' key is pressed
  player.right(); //player moves right
}

if(keyIsDown(87) || keyIsDown(32)){ //if 'w' or space is pressed
  if(player.y == floorHeight-50){ // and the player is on the floor
    player.jumping = true; // player jumps
  }
}

//weapon slot switching
if(keyIsDown(49)){
  changeWeaponSlot(0);
}
if(keyIsDown(50)){
  changeWeaponSlot(1);
}
if(keyIsDown(51)){
  changeWeaponSlot(2);
}
if(keyIsDown(52)){
  changeWeaponSlot(3);
```

```
 }
 if(keyIsDown(53)){
   changeWeaponSlot(4);
 }


 //Reload Mechanic
 if(keyIsDown(82) || currentlyReloading ||
equippedWeapons[currentWeaponSlot].ammoInMag == 0){ //if 'r' is pressed
or player is  currently reloading or auto reloads if a setting

   //finds the corresponding weapon type from the currently equipped
weapon
   let weapon = matchWeaponType(equippedWeapons[currentWeaponSlot])

       //finds the correct ammo type for the weapon the player has
equipped
       for(ammo of ownedAmmo){
         if(ammo.type == weapon.bullet) {

           //checks if the mag isn't full and is available to reload
           if( equippedWeapons[currentWeaponSlot].ammoInMag <
weapon.magSize ){

             //checks if the player has enough ammo left to fully
reload the magazine
             if( weapon.magSize -
equippedWeapons[currentWeaponSlot].ammoInMag <= ammo.amount){


               //sets the start time of the reload for reload delay
reference
               if(!currentlyReloading){
                 reloadStartTime = millis();
               }

               currentlyReloading = true;

               //waits the reload time of the weapon before performing
the reload
```

```
              if( millis() - reloadStartTime > weapon.reloadSpeed *
1000 || reloadStartTime == 0){

                //takes away the ammo from the players spares that is
going to be used
                ammo.amount -= weapon.magSize -
equippedWeapons[currentWeaponSlot].ammoInMag;

                //fully reloads the magazine
                equippedWeapons[currentWeaponSlot].ammoInMag =
weapon.magSize;
                console.log("Player reloaded " + weapon.type + "
magazine fully with " + (weapon.magSize -
equippedWeapons[currentWeaponSlot].ammoInMag) + " spare " + ammo.type +
" bullets.");

                currentlyReloading = false;

              }
            }

            //special case for weapons with unlimited ammo to fully
reload
            else if( ammo.amount == "unlimited" ){

              //sets the start time of the reload for reload delay
reference
              if(!currentlyReloading){
                reloadStartTime = millis();
              }

              currentlyReloading = true;

              //waits the reload time of the weapon before performing
the reload
              if( millis() - reloadStartTime > weapon.reloadSpeed *
1000 || reloadStartTime == 0){

                //fully reloads the magazine
```

```
                console.log("Player reloaded " + weapon.type + "
magazine fully with " + (weapon.magSize -
equippedWeapons[currentWeaponSlot].ammoInMag) + " spare " + ammo.type +
" bullets.");
                equippedWeapons[currentWeaponSlot].ammoInMag =
weapon.magSize;

                currentlyReloading = false;

              }

            }

            //if the player doesn't have enough to fully reload the
magazine but has a small amount spare, use the last bit of ammo to
partially reload the magazine
            else if( ammo.amount > 0 ){

              //sets the start time of the reload for reload delay
reference
              if(!currentlyReloading){
                reloadStartTime = millis();
              }

              currentlyReloading = true;

              //waits the reload time of the weapon before performing
the reload
              if( millis() - reloadStartTime > weapon.reloadSpeed *
1000 || reloadStartTime == 0){

                //fully reloads the magazine
                console.log("Player reloaded " + weapon.type + "
magazine fully with " + (weapon.magSize -
equippedWeapons[currentWeaponSlot].ammoInMag) + " spare " + ammo.type +
" bullets.");
                equippedWeapons[currentWeaponSlot].ammoInMag +=
ammo.amount;
```

```
                //takes away the final bit of ammo from the players
spares that was used
                ammo.amount = 0

                currentlyReloading = false;

              }

            }
          }
        }
 }

 //Shooting Mechanic
 if(mouseIsPressed && !currentlyReloading){ //if the mouse button is
being held down and the player isnt reloading

   //finds the corresponding weapon type from the currently equipped
weapon
   let weapon = matchWeaponType(equippedWeapons[currentWeaponSlot])

       //finds the correct ammo type for the weapon the player has
equipped
       for(ammo of ownedAmmo){
         if(ammo.type == weapon.bullet) {

           //checks the player has enough ammo to shoot weapon
           if(equippedWeapons[currentWeaponSlot].ammoInMag > 0 ||
equippedWeapons[currentWeaponSlot].ammoInMag == "unlimited") {

             //checks if enough time has passed since last bullet was
fired for weapon to shoot
             if(millis() - timeLastFired >= weapon.fireRate * 1000 ||
timeLastFired == 0){

               //adds a new bullet into the game that is travelling
towards the position the player clicked at
               let mouseVector = getMouseVector() //get the direction
that the bullet needs to travel in
```

```
                currentBullets.push(new Bullet(width/2 , player.y,
mouseVector.x, mouseVector.y, ammo.type, 'enemy')); //add bullet to
array with its type
                //initialises properties of the new bullet

currentBullets[currentBullets.length-1].calculateTypeStats();

                //updates time last bullet was fired
                timeLastFired = millis();

                //takes a bullet away from the bullets in the weapon
magazine
                equippedWeapons[currentWeaponSlot].ammoInMag -= 1
              }
            }
          }
        }
 }

 //draws the player HUD

   //Health Bar
   textSize(20);
   fill("white");
   text("Health ❤️", 180, 40); //text above bar
   fill("black");
   rect(180, 60, 296, 25); //background of bar
   fill("red");
   rect(180, 60, 290, 20); //behind green health to show health lost
   fill("green");
   rectMode(CORNER);
   //green health bar length is dependant on the player's current
health
   rect(35, 50, (290 * (player.health/1000)), 20);
   rectMode(CENTER);
   textSize(14);
   fill("white");
   text(player.health + ' / 1000', 180, 65);

   //Weapon Info
```

```
    //getting weapon info
    let weapon = matchWeaponType(equippedWeapons[currentWeaponSlot]);
    let weaponSprite =
matchWeaponSprite(equippedWeapons[currentWeaponSlot].name);

    //render image of weapon
    image(weaponSpriteSheetImg, 450, 70, weaponSprite.sWidth*1.5,
weaponSprite.sHeight*1.5, weaponSprite.sx, weaponSprite.sy,
weaponSprite.sWidth, weaponSprite.sHeight);

    //Display weapon name, ammo left in mag and spare ammo remaining
    textSize(30);
    textAlign(LEFT);
    text(equippedWeapons[currentWeaponSlot].name, 580, 45);

    //display infinite for weapon with unlimited ammo
    if(matchBulletType(equippedWeapons[currentWeaponSlot]).amount ==
"unlimited"){
      text("Spare: ∞" , 580, 75);
    }
    //otherwise display amount of spare ammo
    else{
      text("Spare: " +
matchBulletType(equippedWeapons[currentWeaponSlot]).amount , 580, 75);
    }

    textSize(40);
    textAlign(CENTER);
    text(equippedWeapons[currentWeaponSlot].ammoInMag + "/" +
weapon.magSize, 480, 105);

  //Weapon Slots
    //loops through players equipped weapons
    for(var i=0; i < equippedWeapons.length; i++){

      fill("lightblue");
      stroke("black");
      strokeWeight(1);

      //if weapon slot is equipped highlights box
```

```
        if(currentWeaponSlot == i){
          fill(37, 255, 255);
          strokeWeight(3);
        }

        //draw box for weapon sprite to be inside
        rect( 800 + (i*80) + (i*2), 55, 80, 80);

        //draw weapon sprite of slot inside box if weapon is inside
        if(equippedWeapons[i].name != null){
          let weaponSprite = matchWeaponSprite(equippedWeapons[i].name);
          image(weaponSpriteSheetImg, 800 + (i*80) + (i*2), 50,
weaponSprite.sWidth/1.8, weaponSprite.sHeight/1.8, weaponSprite.sx,
weaponSprite.sy, weaponSprite.sWidth, weaponSprite.sHeight);
        }

        //draw number of weapon slot below box
        strokeWeight(0);
        fill("black");
        textSize(25);
        text(i+1, 800 + (i*80), 90);

      }


 //alerts the player if they need to reload
 if(equippedWeapons[currentWeaponSlot].ammoInMag <= 0 &&
!currentlyReloading){
   fill("yellow");
   textSize(30);
   text("Press 'R' to Reload or Switch Weapon", 540, 150);
 }
 //shows player that they are in the process of reloading and how much
time is left till their reload is complete
 else if(currentlyReloading){
   fill("yellow");
   textSize(30);
   let weapon = matchWeaponType(equippedWeapons[currentWeaponSlot]);
   text("Reloading in " + Math.round( weapon.reloadSpeed - ((millis() -
reloadStartTime) / 1000)) + "...", 540, 150);
```

```
 }
 //updates and draws the player object
 player.update();
 player.render();

 //draws the player crosshair at the position their mouse is aiming
 image(defaultCrosshair, mouseX, mouseY, 30, 30);

}

//function that matches the player's currently equipped weapon to the
corresponding weapon type
function matchWeaponType(equippedWeapon){

 for( weapon of weaponTypes){
   if(weapon.type == equippedWeapon.name){
     return weapon;
   }
 }
}

//function that matches the player's currently equipped weapon's bullet
type to the corresponding owned bullet element
function matchBulletType(equippedWeapon){

 let weapon = matchWeaponType(equippedWeapons[currentWeaponSlot]);

 for(ammo of ownedAmmo){
   if(ammo.type == weapon.bullet){
     return ammo;
   }
 }
}

//function that matches the input weapon name to it's corresponding
weapon sprite
function matchWeaponSprite(name){

 for(sprite of weaponSprites){
   if(sprite.name == name){
```

```
     //sets default width and height of sprite frame
     sprite.sWidth = weaponSprites[0].sWidth;
     sprite.sHeight = weaponSprites[0].sHeight;
     return sprite;
   }
 }
}

//function that changes the player's equipped weapon slot
function changeWeaponSlot(slot){
 if(equippedWeapons[slot].name != null){
   currentWeaponSlot = slot;
   currentlyReloading = false;
 }
}

//function that can be used to spawn a wave of a specific enemy type
function spawnWave(amount,type){

 for(var i=0; i < amount; i++){
   currentEnemies.push(new Enemy(0, floorHeight-50, 50, 100, type));
 }
 }

//generation script for level 1
function generateLevel1(){

 imageMode(CORNER);
 background(lvl1bg);
 imageMode(CENTER);
 fill("blue");
  //rect(player.x-300, floorHeight-50, 100, 100);
 image(treeSprite, player.x+500, floorHeight-90, 180, 250);
 image(treeSprite, player.x+1600, floorHeight-90, 180, 250);

 while(currentEnemies.length < 5){

   var spawnPoints = [ Math.floor((width - player.x + random(500))) ,
Math.floor((0 - player.x - random(500))) ];
```

```
   currentEnemies.push(new Enemy(spawnPoints[Math.floor(random(1.9))],
floorHeight-50, 50, 100, "basic"));
   currentEnemies[currentEnemies.length-1].propertyInitialise();

   currentEnemies.push(new Enemy(spawnPoints[Math.floor(random(1.9))],
floorHeight-50, 50, 100, "speedy"));
   currentEnemies[currentEnemies.length-1].propertyInitialise();

   currentEnemies.push(new Enemy(spawnPoints[Math.floor(random(1.9))],
floorHeight-random(150,400), 50, 100, "speedy"));
   currentEnemies[currentEnemies.length-1].propertyInitialise();

 }
}
```

Player.js

```
//this class creates the player object
class Player {
  constructor(x,y){
   this.x = x //note: the x position for the player acts as a
placeholder to move all other objects in the game, the player's x
position never actually changes
   this.y = y
   this.speed = 3;
   this.jumpSpeed = 6;
   this.jumping = false
   this.health = 1000; //player max health is 1000
   this.currentDirection = "staticright"; //used for sprite animation,
shows direction of player travel
   this.currentFrame; //used for moving sprite frame switching
 }

 //draws the player to the screen
 render(){
   //draws correct player sprite frame at center of the screen
   switch(this.currentDirection){

     case "staticleft":
       image(enemySpriteSheetImg, width/2, this.y, 50, 100,
playerSprites[1].frame3.sx, playerSprites[1].frame3.sy,
playerSprites[0].sWidth, playerSprites[0].sHeight);
       break;

     case "staticright":
       image(enemySpriteSheetImg, width/2, this.y, 50, 100,
playerSprites[1].frame4.sx, playerSprites[1].frame4.sy,
playerSprites[0].sWidth, playerSprites[0].sHeight);

       break;

     case "left":

       //change frame every time width of sprite has passed
       if( Math.floor(this.x) % Math.floor(50/3) == 0){
```

```
          if(this.currentFrame - 1 == 0){
            this.currentFrame = 2;
          }
          else{
            this.currentFrame = 1;
          }
        };

        //display one of the two moving sprite frames
        if(this.currentFrame == 1){
          image(enemySpriteSheetImg, width/2, this.y, 50, 100,
playerSprites[1].frame1.sx, playerSprites[1].frame1.sy,
playerSprites[0].sWidth, playerSprites[0].sHeight);
         }
         else{
          image(enemySpriteSheetImg, width/2, this.y, 50, 100,
playerSprites[1].frame2.sx, playerSprites[1].frame2.sy,
playerSprites[0].sWidth, playerSprites[0].sHeight);
         }
         break;

      case "right":

        //change frame every time width of sprite has passed
        if( Math.floor(this.x) % Math.floor(50/3) == 0){
          if(this.currentFrame - 1 == 0){
            this.currentFrame = 2;
          }
          else{
            this.currentFrame = 1;
          }
        };

        //display one of the two moving sprite frames
        if(this.currentFrame == 1){
          image(enemySpriteSheetImg, width/2, this.y, 50, 100,
playerSprites[1].frame5.sx, playerSprites[1].frame5.sy,
playerSprites[0].sWidth, playerSprites[0].sHeight);
         }
         else{
```

```
         image(enemySpriteSheetImg, width/2, this.y, 50, 100,
playerSprites[1].frame6.sx, playerSprites[1].frame6.sy,
playerSprites[0].sWidth, playerSprites[0].sHeight);
        }
        break;

   }


 }

 //this function is called every draw loop and updates the player
position
 update(){

   //Jumping Mechanics
   //checks to make sure that the player jump is not exceeding the max
jump limit
   if(this.y <= floorHeight-200){
     this.jumping = false;
   }

   //decreases the player y value if the player is jumping
   else if(this.jumping == true){
     this.y -= this.jumpSpeed;
   }

   //creates the effect of gravity if the player is not still jumping
   if(this.jumping == false && this.y < floorHeight-50){
     this.y += this.jumpSpeed;
   }
  }

 //moves the player left if left key is pressed/held
 left(){
   this.x += this.speed;
   this.currentDirection = "left";
 }

 //moves the player right if right key is pressed/held
```

```
 right(){
   this.x -= this.speed;
   this.currentDirection = "right";
 }

}
```

Bullet.js

```
//this class creates the player object
class Player {
  constructor(x,y){
   this.x = x //note: the x position for the player acts as a
placeholder to move all other objects in the game, the player's x
position never actually changes
   this.y = y
   this.speed = 3;
   this.jumpSpeed = 6;
   this.jumping = false
   this.health = 1000; //player max health is 1000
   this.currentDirection = "staticright"; //used for sprite animation,
shows direction of player travel
   this.currentFrame; //used for moving sprite frame switching
 }

 //draws the player to the screen
 render(){
   //draws correct player sprite frame at center of the screen
   switch(this.currentDirection){

     case "staticleft":
       image(enemySpriteSheetImg, width/2, this.y, 50, 100,
playerSprites[1].frame3.sx, playerSprites[1].frame3.sy,
playerSprites[0].sWidth, playerSprites[0].sHeight);
       break;

     case "staticright":
       image(enemySpriteSheetImg, width/2, this.y, 50, 100,
playerSprites[1].frame4.sx, playerSprites[1].frame4.sy,
playerSprites[0].sWidth, playerSprites[0].sHeight);

       break;

     case "left":

       //change frame every time width of sprite has passed
       if( Math.floor(this.x) % Math.floor(50/3) == 0){
```

```
          if(this.currentFrame - 1 == 0){
            this.currentFrame = 2;
          }
          else{
            this.currentFrame = 1;
          }
        };

        //display one of the two moving sprite frames
        if(this.currentFrame == 1){
          image(enemySpriteSheetImg, width/2, this.y, 50, 100,
playerSprites[1].frame1.sx, playerSprites[1].frame1.sy,
playerSprites[0].sWidth, playerSprites[0].sHeight);
         }
         else{
          image(enemySpriteSheetImg, width/2, this.y, 50, 100,
playerSprites[1].frame2.sx, playerSprites[1].frame2.sy,
playerSprites[0].sWidth, playerSprites[0].sHeight);
         }
         break;

      case "right":

        //change frame every time width of sprite has passed
        if( Math.floor(this.x) % Math.floor(50/3) == 0){
          if(this.currentFrame - 1 == 0){
            this.currentFrame = 2;
          }
          else{
            this.currentFrame = 1;
          }
        };

        //display one of the two moving sprite frames
        if(this.currentFrame == 1){
          image(enemySpriteSheetImg, width/2, this.y, 50, 100,
playerSprites[1].frame5.sx, playerSprites[1].frame5.sy,
playerSprites[0].sWidth, playerSprites[0].sHeight);
         }
         else{
```

```
         image(enemySpriteSheetImg, width/2, this.y, 50, 100,
playerSprites[1].frame6.sx, playerSprites[1].frame6.sy,
playerSprites[0].sWidth, playerSprites[0].sHeight);
        }
        break;

   }


 }

 //this function is called every draw loop and updates the player
position
 update(){

   //Jumping Mechanics
   //checks to make sure that the player jump is not exceeding the max
jump limit
   if(this.y <= floorHeight-200){
     this.jumping = false;
   }

   //decreases the player y value if the player is jumping
   else if(this.jumping == true){
     this.y -= this.jumpSpeed;
   }

   //creates the effect of gravity if the player is not still jumping
   if(this.jumping == false && this.y < floorHeight-50){
     this.y += this.jumpSpeed;
   }
  }

 //moves the player left if left key is pressed/held
 left(){
   this.x += this.speed;
   this.currentDirection = "left";
 }

 //moves the player right if right key is pressed/held
```

```
right(){
  this.x -= this.speed;
  this.currentDirection = "right";}}
```

Button.js

```
//this class creates the bullet object
class Bullet {

 constructor(x,y,xSpeed,ySpeed,type, target) {
   this.x = x; //initial position of the bullet is inside the player
   this.y = y;
   this.xSpeed = xSpeed;
   this.ySpeed = ySpeed;
   this.bulletType = type;
   this.target = target;
 }

 //calculates the bullet stats by locating the stats in the bullet
statistics array
 calculateTypeStats(){

   //loop through array until finds bullet with same type
   for(var bullet of bulletTypes){
     if(bullet.type == this.bulletType){
       this.speed = bullet.speed; //assigns speed to this object
depending on bullet speed
       this.colour = bullet.colour; //assings colour to this object
depending on bullet colour
       this.size = bullet.size; //assigns size to this object depending
on bullet size
       this.damage = bullet.damage; //assings damage value to this
object depending on bullet damage
     }
   }

   //buffs bullet damage for enemies the higher the current level
   if(this.target == 'player'){
     this.damage += Math.floor(this.damage * (currentLevel/4));
   }
 }

 update(){
   //moves the bullet towards its target direction
```

```
   this.x += this.xSpeed * this.speed;
   this.y += this.ySpeed * this.speed;
 }

 //checks if the bullet has collided with it's target and if so the
target's health is lowered by the damage of the bullet
 collisionCheck(){

   switch(this.target){

     case 'player': //checks if the bullet x and y is colliding with
the player x and y

       if( ((this.x+this.size/2) >= (width/2-25) &&
(this.x-this.size/2) <= (width/2+25) && (this.y+this.size/2) >=
player.y-50 && (this.y-this.size/2) <= player.y+50)){
         player.health -= this.damage;
         this.destroyed = true;  //this variable will tell the main
game that the bullet has been destroyed and will remove it from the
game after it has collided
       }
       break;

     case 'enemy': //checks if the bullet x and y is colliding with any
enemies currently in the game

       for(var i=0; i < currentEnemies.length; i++){

         if( (this.x+this.size/2) >= (currentEnemies[i].x + player.x) -
currentEnemies[i].width/2 && (this.x - this.size/2) <=
(currentEnemies[i].x + player.x) + currentEnemies[i].width/2 &&
(this.y+this.size/2) >= currentEnemies[i].y- currentEnemies[i].height/2
&& (this.y-this.size/2) <= currentEnemies[i].y +
currentEnemies[i].height/2){

           currentEnemies[i].health -= this.damage;
           this.destroyed = true;

         }
       }
```

```
      break;

    }
  }

  //draws the bullet on the canvas
  render(){
    stroke("black");
    stroke(1);
    fill(this.colour);
    ellipse(this.x, this.y, this.size);
    noStroke();
  }
}

//gets the normalised vector for the position of the mouse.
function getMouseVector(){
  let mouseXalt = mouseX - width/2;
  let mouseYalt = mouseY - player.y;
  let mouseDir = createVector(mouseXalt, mouseYalt);
  mouseDir.normalize();
  return mouseDir;
}
```

Spritesheet.js

```
//this script stores the arrays of all the sprite sheets for each
object used in reclamation
// each sprite sheet array is stored with each element being arrays of
objects containing the data for individual sprite frames for each
sprite type.

var weaponSprites = [
 {name: null, sWidth: 150, sHeight: 100},
 {name: "Glock", sx: 50, sy: 50},
 {name: "Scar", sx: 200, sy: 50},
 {name: "Magnum", sx: 375, sy: 50},
 {name: "AK-47", sx: 540, sy: 50},
 {name: "Uzi", sx: 50, sy: 200},
 {name: "Minigun", sx: 200, sy: 200},
 {name: "RG-6 Grenade Launcher", sx: 375, sy: 200},
 {name: "Hunting Rifle", sx: 540, sy: 200},
 {name: "M24 Sniper", sx: 50, sy: 350},
 {name: "Machine Pistol", sx: 200, sy: 350},
 {name: "Thompson", sx: 540, sy: 350},
 {name: "M16A1", sx: 700, sy: 350},
 {name: "Vector", sx: 550, sy: 525}
]

//enemy sprite frames are stored with the first frame being the left
facing frame then frames working their way to the right facing frames.
var enemySprites = [
 {name: null, sWidth: 45, sHeight: 72},
 {name: "basic", frame1: {sx: 434, sy: 668}, frame2: {sx: 504, sy:
665}, frame3: {sx: 516, sy: 489}, frame4: {sx: 442, sy: 492} },
 {name: "speedy", frame1: {sx: 42, sy: 667}, frame2: {sx: 109, sy:
664}, frame3: {sx: 118, sy: 489}, frame4: {sx: 42, sy: 489} }
]

var playerSprites = [
 {name: null, sWidth: 45, sHeight: 72},
 {name: "default", frame1: {sx: 241, sy: 308}, frame2: {sx: 371, sy:
308}, frame3: {sx: 305, sy: 306}, frame4: {sx: 301, sy: 132}, frame5:
{sx: 231, sy: 135}, frame6: {sx: 364, sy: 133}}]
```

Level-select-menu.js

```
//script for the level selection menu
function levelSelectMenu(){

 //draws all the buttons for the current scene on the screen
 for(b of buttons){
   b.render();
 }

 //draws the cursor on the screen
 image(cursorImg, mouseX, mouseY+30, 80, 80);

}
```

Main-menu.js

```
//script for the main menu scene of the game
function mainMenu(){
  //draws all the buttons for the current scene on the screen
 for(b of buttons){
   b.render();
 }

 //draws the menu heading on the screen
 textSize(70);
 fill("lightgreen");
 text("Reclamation", 952.5, 200);

 //draws the cursor on the screen
 image(cursorImg, mouseX, mouseY+30, 80, 80);
}
```

Pause-menu.js

```
//script for the main menu scene of the game
function mainMenu(){
  //draws all the buttons for the current scene on the screen
 for(b of buttons){
   b.render();
 }

 //draws the menu heading on the screen
 textSize(70);
 fill("lightgreen");
 text("Reclamation", 952.5, 200);

 //draws the cursor on the screen
 image(cursorImg, mouseX, mouseY+30, 80, 80);
}
```

Shop-menu.js

```
//used for shop button organisation
var yPos = 200;

function shopMenu(){

 //draws all the general buttons for the current scene on the screen
 for(b of buttons){
   b.render();
 }

 //draws player currency at top of screen
 fill("lightyellow")
 rect(width/2, 50, 400, 100);
 fill("blue");
 text("Your Balance:",width/2, 40);
 text("£"+currentMoney, width/2, 70);

 shopButtons = [];

 //creates shop buttons for weapons that the player doesn't own
 for(weapon of weaponTypes){

   if(ownedWeapons.includes(weapon.type)){
     //weapon already owned
     weapon.owned = true;
   }

   //create shop button
   shopButtons.push( new shopButton(weapon,500,200));

 }

 //organises shop buttons across the screen in rows
 for(var i=0; i < shopButtons.length; i++){
   shopButtons[i].x = (200 + (i)*100 + (i)*250);

   if(i >= 5){
     shopButtons[i].y = 450;
```

```
    shopButtons[i].x = (200 + (i-5)*100 + (i-5)*250);
  }
  if(i >= 10){
    shopButtons[i].y = 700;
    shopButtons[i].x = (200 + (i-10)*100 + (i-10)*250);
  }


}

for(button of shopButtons){
  button.renderWeaponButton();
  button.renderAmmoButton();
}

//draws the cursor on the screen
image(cursorImg, mouseX, mouseY+30, 80, 80);

}

//class that creates a shop button that the player can use to purchase
a weapon or ammo for the weapon
class shopButton{

constructor(weapon,x,y){

  this.x = x;
  this.y = y;

  this.weapon = weapon;

  //matches weapon name to its correct weapon sprite
  this.sprite = matchWeaponSprite(weapon.type);

  //matches correct ammo type to the weapon name
  this.ammo = matchAmmoType(weapon.bullet);
}

renderWeaponButton(){
```

```
   //draw weapon image in a box
   fill("lightgreen");
   rect( this.x, this.y, 100, 100);
   image(weaponSpriteSheetImg, this.x, this.y, this.sprite.sWidth/1.8,
this.sprite.sHeight/1.8, this.sprite.sx, this.sprite.sy,
this.sprite.sWidth, this.sprite.sHeight);

    //draw buy box option
   fill("lightblue")
   rect( this.x + 110, this.y, 120, 100);
   fill("blue");
   if(this.weapon.owned == undefined){
     text("BUY", this.x + 110, this.y-10);
     text("£" + this.weapon.price, this.x + 110, this.y+20);
   }
   else{
     fill("red");
     text("OWNED", this.x + 110, this.y);
   }
   fill("lightgreen");
 }

 weaponBuyClickCheck(){
   if (mouseX > this.x + 50 && mouseX < this.x + 170 && mouseY > this.y
- 50 && mouseY < this.y + 50) {
     //buy weapon if player has enough money and they don't already own
the weapon
     if(this.weapon.owned == undefined){
       if(currentMoney >= this.weapon.price){
         ownedWeapons.push(this.weapon.type);
         currentMoney -= this.weapon.price;
       }
     }
   }
 }

 renderAmmoButton(){

   //draw box for ammo buy option
   fill("lightyellow");
```

```
   rect(this.x+60, this.y + 100, 220, 90);
   fill("blue");
   text(this.ammo.type, this.x+60, this.y + 95);
   text("Buy x30 £" + this.ammo.price, this.x+60, this.y + 130);
 }

 ammoBuyClickCheck(){
   if (mouseX > this.x - 50 && mouseX < this.x + 170 && mouseY > this.y
+ 52 && mouseY < this.y + 140) {
     //buy ammo if player has enough money
     if(currentMoney >= this.ammo.price && this.ammo.type != "9mm"){

       for(var i=0; i < ownedAmmo.length; i++){
         if(ownedAmmo[i].type == this.ammo.type){
           ownedAmmo[i].amount += 30;
           currentMoney -= this.ammo.price;
         }
       }
     }
   }
 }

}

//function that matches ammo type to weapon name
function matchAmmoType(weaponBullet){
  for(ammo of bulletTypes){
   if(ammo.type == weaponBullet){
     return ammo;
   }
 }
}
```

Weapon-select-menu.js

```
//script for the weapon select menu where the player chooses which of
their weapons to equip before they enter a level
function weaponSelectMenu(){

 //draws all the buttons for the current scene on the screen
 for(b of buttons){
   b.render();
 }

 //draws heading on the screen
 textSize(70);
 fill("lightgreen");
 text("Select your Weapons", width/2, 105);

 //stores equipped weapon buttons (clears each loop)
 equippedButtons = [];
 //stores unequipped weapon buttons
 unequippedButtons = [];

 //creates equipped weapon slot buttons based off weapons equipped in
slots
 for(var i=0; i< equippedWeapons.length; i++){
   equippedButtons.push(new weaponButton(equippedWeapons[i].name,
"lightblue", (740 + i*100 + i*2), 250, "unequip", i))
 }

 //creates unequipped weapon buttons based off weapons unequipped and
owned by the player
 for(var i=0; i< ownedWeapons.length; i++){
    //if this owned weapon is not equipped
    if(!checkIfEquipped(ownedWeapons[i])){
     //create a button for the owned weapon
     unequippedButtons.push(new weaponButton(ownedWeapons[i],
"lightblue", (500 + unequippedButtons.length*100 +
unequippedButtons.length*20), 390, "equip", undefined));
    }
 }
```

```
 //draws equipped weapon slots
 for(var i=0; i < equippedButtons.length; i++){
   equippedButtons[i].render();
 }

 //draws unequipped weapon options
 for(b of unequippedButtons){
   b.render();
 }

 //checks if the player should be able to begin the level (if no more
spare weapons or if they have 5 equipped)
 if(unequippedButtons.length == 0 ||
!spareSlotCheck(equippedWeapons,"equip")[0]){
   //add a play button option to the screen
   buttons = [
   new Button("Main Menu", 952.5, 800, 300, 50, () => { gameMode =
'mainMenu'; buttons = mainMenuButtons}),
   new Button("Play", 952.5, 160, 300, 50, () => { gameMode =
'singlePlayerGame'; buttons = inGameButtons; singlePlayerInitialise()})
   ]
 }
 else{
   buttons = weaponSelectButtons;
 }


 //draws the cursor on the screen
 image(cursorImg, mouseX, mouseY+30, 80, 80);

}

//class that creates a weapon button that the player can press to
equip/unequip weapon
class weaponButton {

 constructor(weaponName, fill="lightgreen", x, y, type, slotIndex){
   this.x = x;
   this.y = y;
   this.fill = fill;
```

```
  //type stores if it is a slot with option of equip or unequip
  this.type = type;

  this.weaponName = weaponName;
  //matches correct weapon sprite to object
  this.sprite = matchWeaponSprite(weaponName);
  this.slotIndex = slotIndex;
 }

 //draws button object
 render(){
  fill(this.fill);
  rect( this.x, this.y, 100, 100);
  if(this.weaponName != null){
    image(weaponSpriteSheetImg, this.x, this.y,
this.sprite.sWidth/1.8, this.sprite.sHeight/1.8, this.sprite.sx,
this.sprite.sy, this.sprite.sWidth, this.sprite.sHeight);
  }
  if(this.type == "unequip"){
    fill("black");
    textSize(25);
    text(this.slotIndex+1,this.x,this.y+30);
  }
 }

 clickCheck(){

  //if the mouse is within the box when clicked
  if (mouseX > this.x - 50 && mouseX < this.x + 50 && mouseY > this.y
- 50 && mouseY < this.y
    + 50) {

      var slotData = spareSlotCheck(equippedWeapons,this.type);

      //if a slot is not empty, give slotdata the index of this button
slot
      if(slotData[1] == undefined){
        slotData[1] = this.slotIndex;
      }
```

```
      //if weapon is equippable
      if(this.type == "equip"){
        //if there is a spare weapon slot left
        if(slotData[0]){
          //add weapon to player's equipped weapons
          equippedWeapons[slotData[1]].name = this.weaponName;
        }
      }

      //if the weapon is already equipped
      else{

        //remove the weapon from the equippedWeapons
        equippedWeapons[slotData[1]].name = null;
      }
   }
 }

}

//function that returns if a weapon slot is empty and the location of
the empty slot
function spareSlotCheck(equippedWeapons,type){

 //if weapon is trying to be equipped
 if(type == "equip"){
   //check if there is a spare slot
   for(var i=0; i<equippedWeapons.length; i++){
     if(equippedWeapons[i].name == null){
       return [true,i];
     }
   }
   return [false];
 }
 //if weapon is being unequipped
 else{
   return [false];
 }
```

```
}

//function that checks if the input weapon type is equipped
function checkIfEquipped(weapon){

 for(w of equippedWeapons){
   //if equipped
   if(w.name == weapon){
     return true;
   }
 }
 return false;
}
```