NEA

Name: Jez Snelson
Candidate Number: 1209
Centre Number: 62337

Contents

1 Analysis

1.1 Dungeon Crawlers

A dungeon crawl is a scenario in role playing games in which the main character navigates a dungeon environment often solving traps or fighting monsters to progress through the level. A video game or board game made up of predominantly dungeon crawls is considered to be a dungeon crawler.

Most dungeon crawlers have a fixed map that is the same every time which can lead to little replay value as it can be boring to replay the same map over and over.

1.2 The Problem

Dungeon Crawler style games can be boring and repetitive, this means they can have little to none replay value. Additionaly alot of Dungeon crawlers have a steep learning curve that makes it hard for new or casual players to fully enjoy them. These games are also very complex often demanding lots of time for a simple playthrough. In addition, Non-Computational Methods are inconvenient as they can take up alot of space, take a long time to set up and you cannot save your game state to pick it up later easily.

1.3 Stakeholders

1.3.1 Survey

I chose a set of questions in order to survey my stakeholders and help me find success criteria for the project to fulfill their needs.
1. How often would you say you play video games on a scale of 1-10 (1 being every other week 10 being every day)
2. Do you have any specific or requirements for this computer game?
3. How would you use this game?
4. Would you say you have the time to commit to learning a complex or unintuitive game?(yes,probably not,no)
5. How long would you say is your average gaming session?(1-5 hours)
6. Which different ways do you play video games?(multiple choice: controller, wasd, arrows)
7. Have you played any Dungeon Crawler games(e.g. Legend of Zelda, Binding of Isaac, Dead Cells, Hades)?
8. If not would you want to try a Dungeon Crawler Game?
9. Rank the features of classic dungeon crawlers you dislike the most(Lack of Replayability, Long Unskippable Cinematics, High length of time required for a playing session, The Learning Curve, The Difficulty)
10. Rank the features you think are most essential for the game to be enjoyable for you(Procedurally Generated Dungeons, Loot to Collect and utilise, Some Sort of skill tree, Co-Op mode, Puzzles, Hidden Areas)

1.3.2 Survey Results

Time available:
On average my stakeholders session length is around 2 hours for a single game. On average they play videogames almost every day however there is one that plays infrequently. Because of this I will have to try and make it easy to pick up without much you have to remember about previous sessions.
Most of my stakeholders do not have time to commit to learning a complex or unintuitive game and so I will have to make the game easy to pick up but still have complexities for those who want a challenge.
All controlling mechanisms where popular but WASD was the most so I will prioritise that.
50% of my stakeholders have played dungeon crawlers and so may be experienced with it but 50% have not so I should aim to make it a good introduction to the dungeon crawler genre with the potential of adding optional difficulty for those more experienced.

Disliked Features (Ranked most to least disliked):
1. Lack of replayability.
2. High length of time required for a playing session.
3. The Learning Curve.
4. Long Unskippable Cinematics.
5. The Difficulty.

Due to this I will focus on replayability through the use of procedural generation whilst still aiming to exclude the more disliked features.

Liked Features (Ranked from most to least liked):
1. Some sort of skill tree.
2. Hidden Areas
3. Procedurally Generated Dungeons.
4. Loot to collect and utilise (e.g. weapons).
5. Puzzles.
6. Co-Op Mode.

Because of this I will prioritise getting the more liked features done and exclude some of the less liked features from my success criteria.

1.3.3 About Stakeholders

Name	Description	How they will use my product
Samuel Vanderstelt-Hook	18 year old Male Sixth Form Computer Science Student, Casual Gamer who enjoys a wide range of games.	Sam will use my solution for casual gaming for fun as a break from his studies. He has stated needs for a game that is replayable and gives him a reason to come back to it.
Daniel Olde Scheper	18 year old Male A Level Computer Science Student	Daniel will use my solution as a way to relax from his A-Level Studies. He has stated needs for a fun, replayable and easy to pick up game.
Peter Dunn	17 year old Male College Student and aspiring hobbyist game developer.	Peter will use my solution as a form of entertainment after studies and as he loves Dungeon Crawl Style games. He needs a replayable game with an intuitive combat system.
Sadiya Shorkar	17 year old Female Student and Casual Video Game Enjoyer	Sadiya will use my solution as a form of casual entertainment for short sessions. Sadiya has seizures and so needs accessibility options like volume control and options for less vibrancy.
Penelope Castiau	18 year old Female Sixth Form Student, Avid Computer Gaming Enjoyer and Hobbyist Streamer.	Penny will use my product for entertainment purposes and to play on stream. Because of this Penny needs subtitles to make the game easy to follow for viewers.
Steff Stylianos	17 year old Female College Student and Game Developer	Steff will use my product to relax from studies. Steff needs a replayable game but also want it to be engaging.

1.4 Research

1.4.1 Existing Solutions

Edmund McMillen's The Binding of Isaac
Edmund McMillen created the popular dungeon crawler roguelike The Binding of Isaac and released it on Steam$_{(1)}$. This game was relatively unique as it had procedurally generated dungeons
using a system of rooms that tesalate with each other.

The procedurally generated dungeons consist of different shaped square based rooms that tesalate and are generated next to each other in a psuedo random fashion whilst obeying a set of rules. The mobs that spawn in each room can vary but there is usually only one or two enemy types per room and as you go up levels the amount of enemies and difficulty the pose increases. This system allows for every playthrough of the game to be different to the next with the same reccuring theme/difficulty which allows for lots of replay oppurtunity. This would be an appropiate way for me to fix the replayabilty issue.

I like the games simple UI design as it clearly indicates all the necessary parts. The Map also shows the basic stucture of the level without revealing too much. I want to take inspiration from the simplicity of ui in order to help my game be intuitive.

However, the game has a couple issues that mean that it does not completely solve our problem. First is the steep learning curve that the game presents which, although to some is a welcome challenge, can put off new or less experienced players especially due to its roguelike nature meaning when you die you start from scratch. The game also has an unintuitive movement and fighting system as there is only really quad directional projectiles and a simple walking design which when combined contributes to the steep learning curve. These are some of the features I will not include in my product opting to instead try for an easier approach by allowing scaleable difficulty and oct-directional movement and attacks.

Figure 1: A screenshot of The Binding of Isaac UI and Map

Motion Twin's Dead Cells

Motion Twin created the roguelike dungeon crawler and metroidvania Dead Cells which is released on steam(2). This game is known for its permadeath system and its procedurally generated dungeons.

The way Dead Cells uses procedural generation interests me as it allows for there to be some fixed attributes to the level whilst still allowing elements of randomness. The developers talk about how they do this in a video devlog(3), here the dev talks about his system of having a fixed structure for each level almost like a skeleton. This skeleton will include stuff like important rooms along the way and how much distance of rooms has to be between them. It then fills in all the spaces for rooms with one of the many handmade rooms made by the developers. After one room has been chosen for a spot this leaves less choice for the other spots as the rooms need to join and flow into each other properly and so as it chooses more of them the structure of the level is determined similar to the wave function collapse algorithm(4).

This style of generation allows for a unique experience each time whilst keeping a hand crafted and natural feel to the levels that is often lost in other techniques. Because of these advantages I will take heavy inspiration from this style of procedural generation for my level generation in order to make them more unique.

However due to the game being aimed at more hardcore gamers with it being part of the rouguelike genre it can often appear complex and offputting to newer players who dont like the idea of taking multiple runs just to have very little to show for it and not much forward progress in the game. Although the game is a side on game I think that I will use the idea of its procedural generation as inspiration in my product aswell as aiming to forgo some of the games more complex or challenging mechanics such as permadeath in order to create a more accessible game.

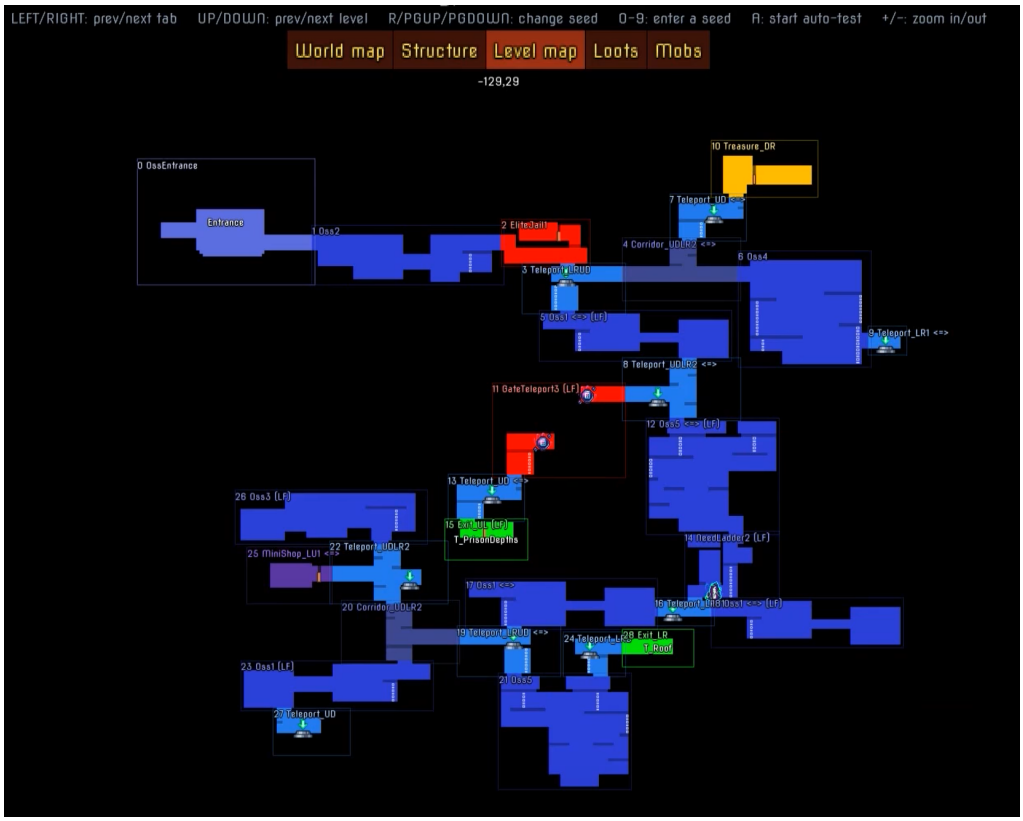

Figure 2: Dead Cells Level Generation Example

Nintendo's Legend Of Zelda Breath of the Wild
Nintendo created the open-world dungeon crawler which is released on the Nintendo Wii U and the Nintendo Switch[(5)]. This game is known for its open world approach to dungeon crawlers as well as its easy to pick up nature for first time players.

The game starts with a tutorial that teaches players the mechanics of the game (combat, exploration, and resource gathering). This tutorial helps players into the world without overwhelming them, offering opportunities to learn at their own pace which helps reduce the steep learning curve of other games in the genre. The open-world nature of the game also adds to its replayability, allowing the player to take many different routes to complete the game. However, while the games size and allows for alot of replayabilty, the volume of content and time required to explore everything can reduce its effectiveness as a game that can be picked up easily for shorter sessions. Its 3D world and complex systems are features that would be too tricky to implement within the scope of an A-level computer science project. It also does not fully fit the dungeon-crawler genre, particularly as it is less dungeon-focused.

I want to take inspiration from the open-world nature of the game to allow different routes through my game to increase replayability aswell as its approach to tutorials in order to make the learning curve steeper. Ontop of this another feature I would like to take inspiration from is the intuitiveness of the combat system which is easy to learn but hard to master in particular its feature of being able to lock onto enemies.

Some features I will not be including are the 3D nature and the overall content heaviness aswell as the focus less on dungeon crawling as I believe these would be unnecesary features which would drive up the complexity of the solution both to make and run.

Figure 3: BOTW tutorial map

This map shows how the BOTW tutorial has the different "shrines" placed to help the player learn the basic mechanics of the game.

1.5 Limitations and Requirements

Requirement	Description	Justification
Hardware	PC or laptop with a Keyboard or Game Controller, minimum of 4GB RAM. For Windows/Linux: x86_32 CPU with SSE2 instructions, any x86_64 CPU, ARMv8 CPU. For Macos: x86_64 or ARM CPU. Integrated graphics with full OpenGL 3.3 support	These are the requirements for running an executable from Godot. The keyboard(WASD) or controller is needed as the input for the game.
Software	I will be using the Godot Game Engine and GDScript to program my game.	I will be using Godot as it is a good 2D game designer that is Free and Open-Source it changes less often than alternatives such as Unity. Ontop of this I have prior experience in Godot and GDScript.
OS Limitations	For Native Exports: Windows 7 or newer, macOS 10.13 or newer, Linux distribution released after 2016 For Web: Firefox 79, Chrome 68, Edge 79, Safari 15.2, Opera 64	Godot can export easily to any of these platforms and more accessibility is good and I can also export a HTML5 version to be hosted in a website such as https://www.itch.io.
General System Limitations	A visually or auditory excellent experience	I do not have the experience with shaders or music and sound effects to add these features to the game in this time and it would make the game requirements higher.

1.6 Features

1.6.1 Essential Features

Feature#	Feature	Description	Justification
1	Player Movement and Controls	The player will control movement using the WASD keys for up, left, down and right respectively Q will trigger a dash. Alternatively they will use the left control stick of a controller.	This will be used to navigate around the Dungeon environment and WASD was the most popular control mechanism for the stakeholders with controller close behind. The controls of my game aim to follow the stakeholder feedback aswell as the general controls that seemed to be favoured in my research
2	A Basic Combat System	The combat system will consist of a primary weapon (melee, magic or ranged) on mouse-1/1 key/X button and a sheild or secondary weapon on mouse-2/2 key/Y button. I will have to implement projectiles and hitboxes for both the player and enemies.	A basic combat system is essential as it will provide the main difficulty and entertainment within the game. The existing solutions all have at least a basic combat system as one of the driving forces for progress through the game.
3	Dungeon Environment	The Dungeon Environment will consist of different shaped rooms with different purposes(e.g. boss room, chest room and shop room.) with hallways connecting inbetween them and a starting room.	A Dungeon Environment is essential as it is the environment the player will play in. All the existing solutions had dungeon environments as this is an essential part of a 'dungeon crawler'.
4	Different Enemies	The Enemies will consist of a variety of enemies that attack the player with different patterns and have different looks and animations.	This is essential as it will add variety to the gameplay and each enemy will provide a challenge to the player as I saw it used during my research.
5	Appearance and Animations of the Player	The Player will have a recognisable appearance aswell as animations for all its actions such as walking and fighting	This is essential as it tells you about where your character is aswell as what they are doing even if the animations are basic like in Binding of Isaac.
6	Login System	Users will be able to login in order to save and reload their progress. The login system will use a username and password with the details being encrypted and stored in an external database. Their will be options for signing in or creating a new account aswell as resetting your password.	This is an essential feature as saving progress is essential for making the game replayable. All the Exisiting solutions I looked at either had login systems or used an existing login system (e.g. steam) in order to manage seperate user saves.
7	User Interface	A Simple UI that shows status indicators like health, weapons being used, enemy health and magic points.	This would allow the player to be aware of the characters health and communicate necessary information for playing the game as I found through the Binding of Isaac UI.

1.6.2 Desireable Features

Feature#	Feature	Description	Justification
8	Weapons and a more Advanced Combat System.	A system of weapons where you can get them from boss drops and potentially shops and a combat system with normal, charged (based on how long you hold down) and special attacks (using a special key).	Different weapons will allow each player to have a playstyle more customized to them and will allow for the player getting stronger as they progress more. An advanced combat system will allow for a more smooth and enjoyable fighting experience as I saw through my research into Dead Cells and BOTW.
9	Skill Tree	A skill tree to unlock unique skills/abilities and get better at using existing skills/weapons. You would gain points from playing the game and can then put them into different areas in order to create a customized character build	This would further allow the player to choose their own play style and add an element of replayability where you can try going for a different build each time you play. This was also requested by the stakeholders and can be seen in Dead Cells.
10	Procedurally Generated Dungeons	The Dungeons would be procedurally generated whilst keeping some amount of structure (e.g. the same amount of distance between posses and key rooms). This would happen through many similar small room sections that can be slotted together in order to make a full dungeon.	This would create a more engaging game which is different each time you play it and therefore increase replayability exponentially as the different combinations of room increases. This was also requested by the stakeholders and was used in Dead Cells to allow for greater replayability.
11	Hidden Areas	Secret areas that can be unlocked through wasy such as progressing further in the game and coming back or through puzzles/fake walls. Could have secret loot or bosses.	This feature was highly requested by the stakeholders and can be seen in alot of existing solutions and would allow for more time spent having fun in the game through finding these areas.
12	Inventory System	An Inventory to be opened with the E key or the + button through which you will manage equipped weapons, key items, skills and more.	An Inventory System is an essential feature if we want to add more weapons/weapon types and a skill tree. It can be seen in BOTW and less complex in Dead Cells.
13	Settings and Volume Control	A settings page to control the volume of noises aswell as the vibrancy of colours.	One of the Stakeholders has requested this as a feature to help the game be more accessible to them.
14	Difficulty Levels and Hardcore Mode	A Difficulty level selector which allows the user to up the difficulty(damage the enemies do etc) and a Hardcore Mode which switches the game to a roguelike format with seperate save state to the normal game.	50% of the stakeholders are experienced with Dungeon Crawlers so in order to help the game still be reasonably challening for them I will add a difficulty slider.

1.7 Success Criteria

Criteria #	Abstraction	Success Criteria	Success Indicators
1	Players to be able to control and move the player using both the WASD keys and a controller.	1.1 W key - Forward 1.2 A key - Left 1.3 S key - Backward 1.4 D key - Right 1.5 Q key - Dash 1.6 Left Control Stick directional movement corresponds to player movement.	WASD/Left Stick direction - Move in that direction Q - Faster movement in direction player is facing
2	Players to be able to have different weapons and attack with them.	2.1 mouse-1/1 key/X button - Primary Attack 2.2 mouse-2/2 key/Y button - Secondary Attack 2.3 Add a basic melee sword 2.4 Add a basic ranged bow and projectiles 2.5 Add a basic magic staff and projectiles 2.6 Add a basic magic staff with area of effect attacks 2.7 Add a hitbox for the player 2.8 Add a health bar for the player 2.9 Make sure all attacks go in the direction the player is facing	Attacks are triggered when their corresponding controls are pressed. Melee attacks affect all enemies within range in the direction the player is facing causing them to lose health. Projectiles launch on a ranged attack and travel in the direction the player is facing Area of effect attacks spawn an area around the player that slowly damages enemies that come into it. Enemy attacks cause player health to go down. Player health accurately displayed on a health bar in the UI
3	A Dungeon environment for the character to walk around and different rooms	3.1 Walls that you cannot walk through 3.2 Floor of the Dungeon 3.3 Interactive chests for loot 3.4 Seperate Boss, Chest and Monster Rooms 3.5 A room Door that only opens on a certain condition 3.6 A Dungeon Environment built out of the rooms and corridors	Ability to walk around the dungeon environment and remain contained by it. Ability to open chests and recieve a specific quantity of random loot from a pool. A level built out of specific purpose built rooms and corridors.
4	Different Enemies for the player to face including bosses	4.1 Enemy Sprites 4.2 Enemy Pathfinding Abilities 4.3 Enemy sight range 4.4 Enemy hitbox 4.5 Enemy health tracking 4.6 Melee Enemies 4.7 Projectile Enemies 4.8 Boss Enemies with different attack combinations	Enemies have distinct and visually recognisable sprites with smooth animations. Enemies navigate around walls and obstacles and follow the player. Enemies detect the player within a certain range and react. Player attacks are registered and decrease enemy health. When an enemies health runs out it will die. Melee enemies attack the player within close range.

5	Appearance and Animations of the Player	5.1 Player Sprite 5.2 Walking Animation 5.3 Player sprite turns to face the direction of movement 5.4 Melee Animation 5.5 Magic Animation 5.6 Dash Animation	The Player has a distinct and visually recognisable sprite with smooth animations for walking, melee attacks and others. The direction of the player changes based on last direction moved.
6	Login System	6.1 Password Hashing Algorithm 6.2 SQL Table to store username and hashed password pairs 6.3 Ability to create a new account with unique username 6.4 Validation of Usernames ($1 \leq chars < 15$) 6.5 Input Sanitisation (Removing any escape chars for SQL before sending the command) 6.6 Ability to log in with an exisiting account and correct password 6.7 Ability to reset password (With challenge question) 6.8 A general login form which links the other forms. 6.9 Ability to delete an account.	Uses a strong hashing algorithm with salting. Username password pairs are stored in an SQL table. Users can only create an account if the username is unique and between 1 and 14 characters. Preventation of SQL injection attacks. User's can login with credentials. User's can reset their password. A general login form links to registration, password reset and logging in.
7	User Interface	7.1 Health Bar 7.2 Magic Points Bar 7.3 Display of the weapon being used 7.4 Popup display with enemy health over their head when they get damaged 7.5 ability to switch between weapons	Displays Player health and MP accurately and updates dynamically. Clearly indicates which weapon is being used. Clearly displays enemies health when they get damaged. Allows switching between weapons.
8	Weapons And a More Advanced Combat System	8.1 Different Styles of melee, magic and ranged weapons 8.2 Boss Drops 8.3 Shop System that appears throughout levels 8.4 Charged Attacks (based on how long you hold down) 8.5 Special attacks	Distinct different weapon styles, levels and dynamics. Defeated Bosses drop unique or rare items. Shops can appear throughout levels. Holding down attack button increases power of attacks. More powerful secondary special attacks.
9	Skill Tree	9.1 UI Menu for the skill tree (Some skills required before others unlocked). 9.2 Different Branches (Melee, Ranged, Magic, Defense) 9.3 Experience system. 9.3.1 Experience gained after killing enemies/bosses 9.3.2 Different experience amounts required for different skills 9.4 Ability to unlock skills 9.5 Ability to reset your skill tree	A user-friendy menu displaying skills and prerequesites. Seperate branches for Melee, Ranged, Magic and Defense skills. Players gain xp from defeating enemies and bosses. Different skills requiring different amounts of XP to unlock. Players can spend XP to unlock skills. Players can reset and redistribute points in the skill tree.

10	Procedurally Generated Dungeons	10.1 Creating requirements for each level to satisfy 10.2 Creating different room sections/rooms to peice together 10.3 Creating the algorithm to generate which room sections are slotted together where. 10.4 Create an algorithm to peice the sections together to create a fully playable level. 10.4.1 Level's generated satisfy length requirements 10.4.2 Level's generated contain all the special rooms needed (chest room, secret rooms, etc.)	Each level generated meets specific preconditions. Different Room sections are designed to be peiced together dynamically. An algorithm places room sections together to form a level layout. Generated levels are fully playable and contain all rooms needed.
11	Hidden Areas	11.1 Add mechanics to get into the secret rooms (breakable walls, climbing vines, keys, etc.) 11.1.1 Add a hammer to break walls with 11.1.2 Add climbing gloves which you need in order to climb vines 11.2 Add secret Boss and Treasure rooms for behind these obstacles.	Players can access secret rooms using specific methods. A hammer item allows players to break walls. Players need climbing gloves to scale vines. Secret areas contain unique bosses or loot.
12	Inventory System	12.1 UI for Inventory 12.2 Storage of Extra weapons and key items (keys, armour, charms, etc) 12.3 E key to open up the inventory 12.4 Ability to switch out what Weapons, Armour and charms are equipped. 12.5 Ability to add or remove items from the inventory. 12.6 SQL table to store inventory contents	A clear and intuitive menu for managing items. Players can store extra weapons and items to be saved in their inventory. E Key - Open Inventory UI. Players can swap weapons/armour. Players can remove items from their inventory. Inventory state persists even when game closes.
13	Settings and Volume Control	13.1 Settings UI with buttons for each setting 13.2 Ability to control the volume 13.3 Ability to control the vibrancy of colours in the game.	A clear and accessible menu with buttons for different settings. Players can adjust the volume of each source of noise in the game. Allow users to adjust colour intensity along with accessibility needs.
14	Difficulty Levels and Hardcore Mode	14.1 A slider for difficulty in create save 14.2 Increasing difficulty based on the slider 14.2.1 Increasing enemy health 14.2.2 Decreasing player health 14.2.3 Increasing number of enemies 14.3 A Hardcore mode at maximum difficulty with a seperate save state to the normal game. 14.3.1 roguelike features (permadeath, resource management, etc) 14.4 SQL table to store different saves	A difficulty slider in the save menu. The game adjusts difficulty by increasing enemy health and damage and increasing the number of enemies. A hardcore mode with permadeath that can be toggled in the save creation menu. Saves persist even when game closes.

1.8 Computational Methods

2 Design & Plan

2.1 Overview

2.1.1 Global Variables

I have a couple of main global variable scripts Global, Inventory, Database etc.

Source	Identifier	Data Type	Justification

2.1.2 Folder Structure

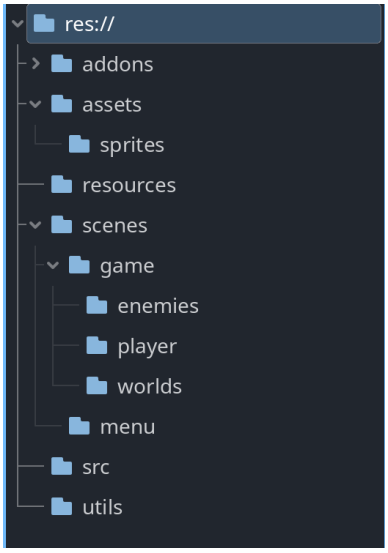

Figure 4: Folder Structure

I chose this folder structure as it will allow me to clearly define where all the different parts of the game are aswell as easily being able to access the closely related parts.
The assets folder will contain all of the external assets, sprites, spritesheets and audio.
The resources folder will contain all of the items (weapons, armour, keys and charms) that I will make to be included in the game.
The scenes folder will contain all the scenes for the menu and the game sorted into their respective folders.
The src folder will contain all of the preloaded scripts for the game.
the utils folder will contain any testing or debugging scripts/scenes to help with the development process.

2.1.3 Naming Convention

For naming I conventions I will adopt the naming conventions already used in godot for ease of integration, readability and consistency with documentation.
The naming conventions are as follows.

Type	Convention	Info
File Names	snake_case	yaml_parsed.gd
Class Names	PascalCase	YAMLParser
Node Names	PascalCase	
Functions	snake_case	
Variables	snake_case	
Signals	snake_case	Past tense ”door_opened”
Constants	CONSTANT_CASE	

2.2 Database Design

I will be using an SQL Database in order to store the data about my users.

2.2.1 ERD

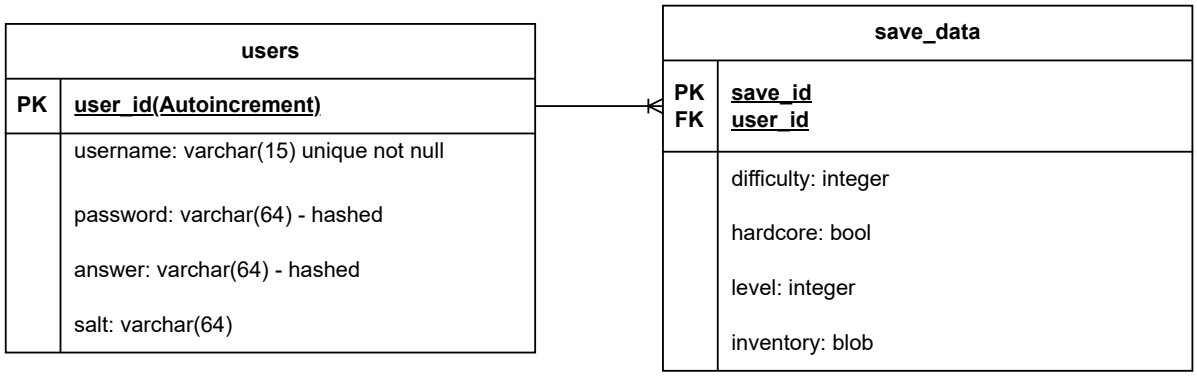

Figure 5: Database Design

Figure 2 shows the Database Design:
The users table will be the main table containing all the login details.
Each user will be able to have multiple save instances which will be stored in save data.
Upon designing the inventory I have decided that I will split the inventory into the stored items which I will use a seperate table to store with the item_id (the path to the item resources location in the game files) as a primary composite key with the save _id aswell as storing the equipped items in the save_data table. I have also decided to split the composite key in the save_data table and just have the save_id as the primary key autoincrementing. This will help keep the inventory more accessible and prevent the need for a BLOB decoder.

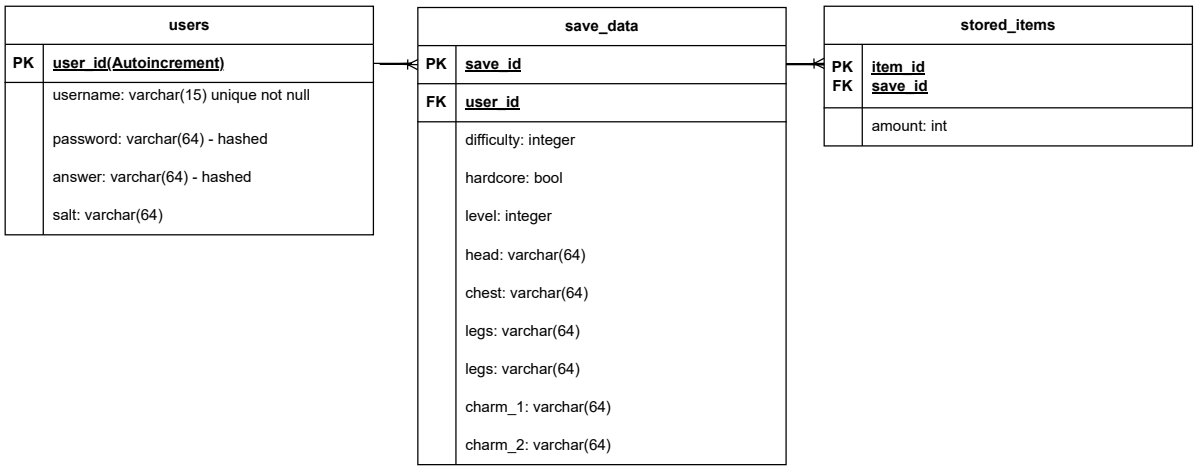

Figure 6: Database Design

2.2.2 Database Naming Conventions

The naming conventions I will adopt for the database is as follows.

Abstract	Convention	Examples	Justification
Tables	Plural snake_case	users,save_data	SQL is case insensitive so with CamelCase it cann't tell the difference between undervalue and underValue
Fields	Singular snake_case	inventory_content, username	
Keys	singular snake_case_table_id	user_id, save_data_id	

2.2.3 SQL Queries

I have to write Queries for each of the actions I want to do.

Name	Description/Justification	SQL

create_table_users	Create's a table for users if it does not exist.	`CREATE TABLE IF NOT EXISTS users (`
		`  user_id INTEGER PRIMARY KEY AUTOINCREMENT,`
		`  username VARCHAR(15) NOT NULL,`
		`  password VARCHAR(64) UNIQUE NOT NULL,`
		`  salt VARCHAR(64) NOT NULL,`
		`  answer VARCHAR(64) NOT NULL`
		`);`

get_user_data	Returns the user data assuming it exists. If it doesnt it will return null.	`SELECT * FROM users` `WHERE username = ?;`

add_new_user	Inserts a new user into users with username, password, challenge question answer and salt	`--Assume hashed password and answer`
		`INSERT INTO`
		`users(username,password,answer,salt)`
		`VALUES (?,?,?,?);`

reset_password	Changes a users password	`--Assume hashed password and answer`
		`UPDATE TABLE users`
		`SET password = ?`
		`WHERE username = ?`

create_table_save_data	Create's a table for save_data if it does not exist.	`CREATE TABLE IF NOT EXISTS save_data (`
		`  save_id INTEGER AUTOINCREMENT,`
		`  FOREIGN KEY (user_id) REFERENCES users(user_id),`
		`  difficulty INTEGER,`
		`  hardcore INTEGER,`
		`  level INTEGER,`
		`  head VARCHAR(32),`
		`  chest VARCHAR(32),`
		`  legs VARCHAR(32),`
		`  weapon VARCHAR(32),`
		`  charm_1 VARCHAR(32),`
		`  charm_2 VARCHAR(32)`
		`);`

add_new_save_data	Adds new save data for a user.	`INSERT INTO`
		`save_data(user_id,difficulty,hardcore,level)`
		`VALUES (?,?,?,?);`

get_save_data	Get's the save data with a specific user_id and save_id	`SELECT * FROM save_data`
		`WHERE user_id = ?`
		`AND save_id = ?;`

get_user_save_data	Get's the save data for all entries with a specific user_id	`SELECT level, hardcore FROM save_data` `WHERE user_id = ?;`

update_save_data	updateSave	`UPDATE save_data`
		`SET`
		`  head = ?,`
		`  chest = ?,`
		`  legs = ?,`
		`  weapon = ?,`
		`  charm_1 = ?,`
		`  charm_2 = ?,`
		`  level = ?`
		`WHERE`
		`  user_id = ?`
		`AND save_id = ?;`

Name	Description/Justification	SQL
create_table_stored_items	Create's a table for stored_items if it does not exist.	`CREATE TABLE IF NOT EXISTS stored_items ( item_id INTEGER NOT NULL, save_id INTEGER NOT NULL, PRIMARY KEY(item_id,save_id), FOREIGN KEY(save_id) REFERENCES save_data(save_id) );`
update_stored_item_amount	Update's a specific person's stored items to increase the amount of something stored (assumes it is stored)	`UPDATE stored_items SET amount = amount + ? WHERE item_id = ? AND save_id = ?;`
get_stored_item_amount	Get's the amount of an item being stored	`SELECT amount FROM stored_items WHERE save_id = ? AND item_id = ?;`
add_stored_item	Adds an item to the stored_items	`INSERT INTO stored_items(save_id,item_id,amount) VALUES (?,?,?);`
count_stored_items	Count's the number of items stored for a save_id	`SELECT COUNT(*) FROM stored_items WHERE save_id = ?;`
remove_stored_item	Removes an item from stored_items	`DELETE * FROM stored_items WHERE save_id = ? AND item_id = ?;`
get_slot_value	Gets the file path of the item equipped in the slot	`SELECT ? FROM save_data WHERE save_id = ?;`
set_slot_value	Sets the value of the slot to the file path	`UPDATE save_data SET ? = ? WHERE save_id = ?;`

I will use godot's *query_with_bindings*() function in order to substitute in the bindings for the ?s in the queries. This is useful as it automatically performs input sanitisation so that the system isn't vulnerable to SQL injection.

2.2.4 Algorithms

login():
The login function will be used to find if the user exists and then check the hashed password if it does. This will help fulfill criteria 6.6.

```
def login(username,password):
    query_result = get_user_data(username) #Getting user data
    if len(query_result) == 0: #If user doesnt exist
        return "InvalidUsernameError"
    if hash(password) == query_result["password"]: #Checking password hash against stored hash
        return True
    return "IncorrectPasswordError" #If password doesnt match
```

add_user():
The add_user function will be used to generate salt for the user check if the username is unique and add the user. This will help fulfill criteria 6.3

```
def add_user(username, password, answer):
    salt = gen_salt() #Generating new salt
    hashed_password = hash(password,salt)
    hashed_answer = hash(answer,salt)
    if not add_new_user(username, hashed_password, hashed_answer, salt): #Tries to add user
                                                                          with hashed password and answer
        return "InvalidUsernameError" #If user cannot be added then the username must be
                                                                            invalid
    return True
```

reset_password():
The reset_password function will be used to check if the username is valid, fetch the user data and then check if the hashed answer is the same as the stored answer before updating the stored password. This will help fulfill criteria 6.7.

```
def reset_password(username, answer, password):
    query_result = get_user_data(username) #Getting user data
    if len(query_result) == 0: #If user doesnt exist
        return "InvalidUsernameError"
    if hash(answer) == query_result["answer"]: #Checking the answer hash against the stored
                                                                          hash
        reset_password(password,username)
        return True
    return "IncorrectAnswerError" #If answer doesnt match
```

2.2.5 Testing Plan

Test #	Function	Parameters	Expected Outcome
6.3.1	add_user()	”Hyrule”, ”Password”, ”Answer”	True
6.3.2	add_user()	”Hyrule”, ”Password”, ”Answer”	”InvalidUsernameError” as a user already exists with that username
6.6.1	login()	”Hyru1e”, ”Password”	”InvalidUsernameError”
6.6.2	login()	”Hyrule”, ”Password”	True
6.7.1	reset_password()	”Hyru1e”, ”Answer”, ”password	”InvalidUsernameError”
6.7.2	reset_password()	”Hyrule”, ”answer”, ”password”	”IncorrectAnswerError”
6.7.3	reset_password()	”Hyrule”, ”Answer”, ”password”	True
6.6.3	login()	”Hyrule”, ”Password”	”IncorrectPasswordError”

These tests will be used to evaluate to what extent I have met the criteria and allow me to identify and fix any issues in my code.

2.3 Login System

2.3.1 Activity Diagram

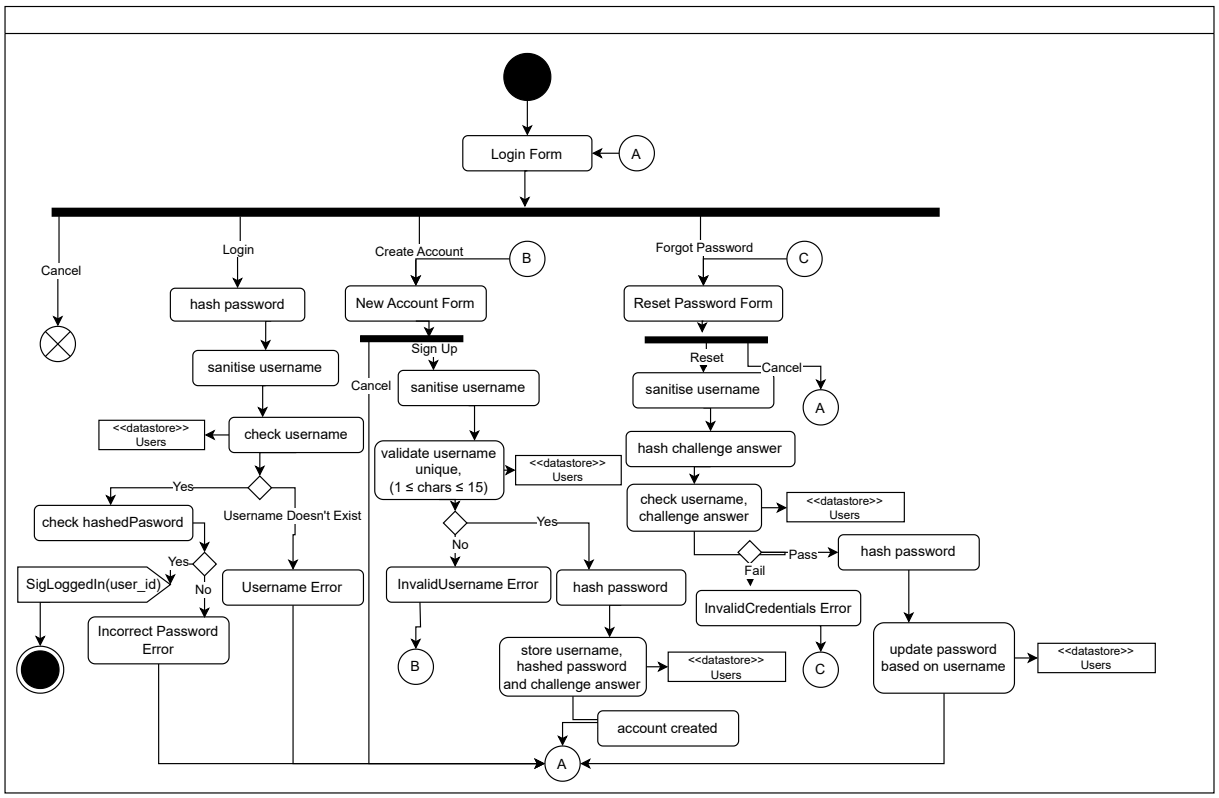

Figure 7: Activity Diagram for login forms

The login form will allow users to create accounts aswell as login with an existing account and reset a password. Upon successful login the user will be redirected to the GAME system. This login form along with the database functions provides all the necessary UI and algorithmic elements to fulfill criteria 6.

2.3.2 Algorithms

hash():
This function alternates between two different hashing functions a consistent number of times while making sure the final hash is a consistent length. It also adds in a unique salt for every user which is necessary to prevent rainbow table lookups and keep the passwords secure even if it is generic. This and the salt function fulfills criteria 6.1.

```
def hash(password: str, salt: str):
    hashedPassword = password
    #Repeating a consistent but unpredicatble amount of times
    #On even rounds the password is sandwidged on odd rounds the salt is sandwidged
    #Alternating the use of sha256 and md5 but making sure to end on sha256 so the hash is a
                                                        predictable length.
    for x in range(1,6*len(password)+1):
        if x\%2 == 0:
            hashedPassword = sha256(md5(salt[x:]+hashedPassword+salt[:x]))
        else:
            hashedPassword = md5(sha256(hashedPassword[:x]+salt+hashedPassword[x:]))
          return hashedPassword
```

genSalt():
The genSalt function uses random processes to generate a salt string to be used in the hashing of the password and challenge question. This needs to be random for every user to prevent the potential to create a table of

common password hashes to loop up user's passwords in.

```
def gen_salt():
    salt = "string"
    x = randint(5,10)
    for i in range(2**x):
        salt = hash(salt,sha256(str(i)))
    return salt
```

2.3.3 Testing Plan

Test #	Function	Parameters	Expected Outcome
6.1.1	gen_salt()		random 256 bit hex string
6.1.2	hash()	"password", "salt"	random 256 bit hex string
6.1.3	hash()	"password", "salt"	the same random 256 bit hex string
6.1.2	hash()	"Password", "salt"	random 256 bit hex string different from before

2.3.4 Mockup Forms

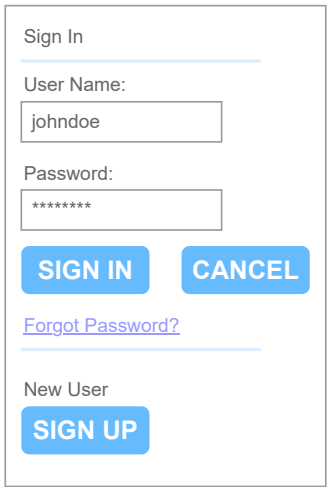

Figure 8: Login Form

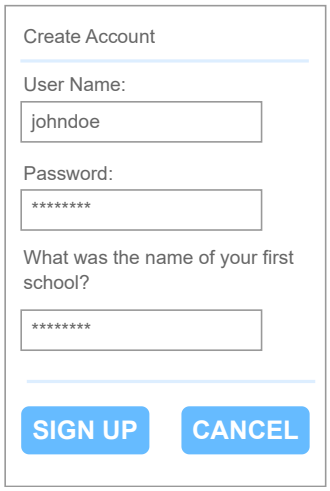

Figure 9: New Account Form

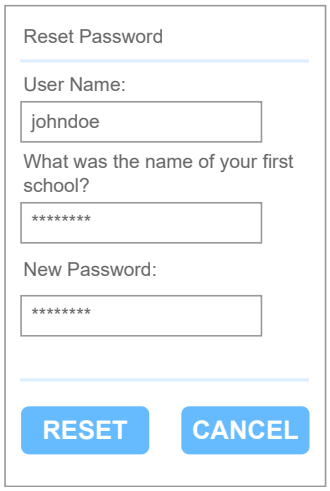

Figure 10: Reset Password Form

These forms would be used in order to create an account, reset your password and login, Helping fulfill criteria 6.3, 6.6, 6.7 and 6.8.
The Password and Challenge question entries would be hidden/starred for privacy.

2.4 Save Select System

I will implement the save select system as an additional menu for the login system in order to improve replayability to help using a ScrollContainer with a VBoxContainer inside to create a list of scrollable items. I will get the list of saves for a given user_id and then display them and when the button for that save is pressed the current_save_id will be set.
In order to add new save's I will add a button for hardcore and a slider for difficulty as well as an add save button which will add the save and update the list.

2.4.1 Algorithms

_ready():

```
def _ready():
    save_data_list = get_user_save_data(current_user_id)
    for save_data in save_data_list:
        button = Button.new
        button.text = f"Level: {save_data['level']} \t Difficulty: {save_data['difficulty']}\n
                                                    Hardcore: {save_data['hardcore']}"
        button.connect("pressed", self, "_on_save_selected", [save_data])
        add_child(button)
```

_on_save_selected():

```
def _on_save_selected(save_data):
    current_save_id = save_data["save_id"]
    get_tree().change_scene_to_file(world)
```

2.5 Item Design

I will implement the different item types using Godot's resource system. This will allow me to define properties that all items of the same type will share and I can use inheritance to allow classes to derive from a parent class.
The resource system is useful as it is reusable throughout scenes and scripts and it can easily be saved and loaded from disk.
The types of items I will aim to implement will be different weapon types, charms/trinkets/amulets, armour and keys.

2.5.1 Class Diagram

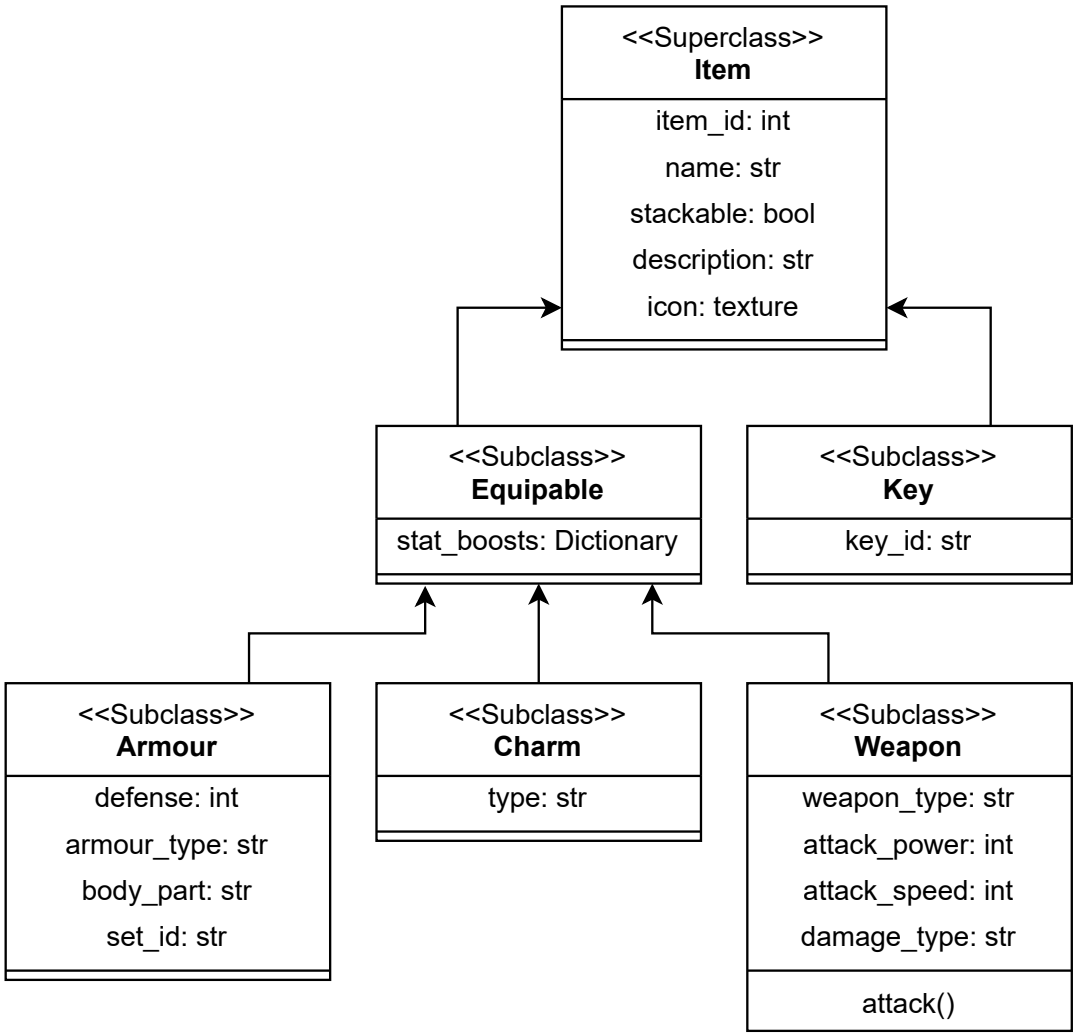

Figure 11: Player Class Diagram

2.5.2 Algorithms

attack():
I will have a number of different weapon types that will have different attacks.
I will implement the attack cooldown through the use of a CooldownTimer(timer node) attached to the player.
Melee:
I will implement the melee attack by creating a variable size hitbox(area2D) scene that can be instantiated as a child of the player in order to detect enemies that would be attacked in the range specified in the resource.

```
def attack(owner: node, direction):
    #Load the hitbox scene
    hitbox_scene = load(hitbox_scene_path)
    hitbox_instance = hitbox_scene.instantiate()
    hitbox_instance.range = range
    hitbox_instance.damage = damage
    hitbox_instance.rotation = direction.angle()

    #Add child
    owner.add_child(hitbox_instance)

    #Hitbox lasts for a tenth of a second
    sleep(0.1)

    hitbox_instance.queue_free()
```

Ranged:
My Bows and ranged magic weapons will shoot out projectiles.

```
def attack():
    #Load the projectile scene
    projectile_scene = load(projectile_scene_path)
    projectile_instance = projectile_scene.instantiate()
    projectile_instance.damage = damage

    #Add Child
    owner.add_child(projectile_instance)
```

display_string(): Upon designing the Inventory UI I have decided to add a function to output a string describing the item, this would make use of polymorphism to behave differently for each item and display the key stats like the name, stat_boosts and such.

2.6 Inventory Design

My inventory design will cover two main parts the equipped items and the item storage. I will have a maximum inventory size script variable so that we can still display all the items in the inventory and a max stack size for stackable items.

2.6.1 Stored Items

I will implement the stored items through a dictionary that stores the item resource and the quantity of it. I will implement add and remove item functions. I will also add a max inventory size.

2.6.2 Equipped Items

I will implement the equipped items through a dictionary where the keys are the slots and the values are what is equipped in that slot. I will also add a equip function to equip an item and an unequip function to unequip the item in a slot.

2.6.3 Clarification

Upon further thought I have decided it is best to use SQL tables instead of dictionaries and use SQL queries to manage the inventory.

2.6.4 Algorithms

add_item():

```
def add_item(item_id: file_path, amount):
    if get_stored_item_amount(save_id, item_id) and item.stackable: #Checks if you can stack
                                                                    the item
        update_stored_item_amount(amount, item_id, save_id)
    elif count_stored_items(save_id) >= max_inventory_size: #Full Inventory
        return "FullInventoryError"
    else:
        add_stored_item(save_id, item_id, amount)
    return True
```

remove_item():

```
def remove_item(item_id: Resource, amount: int = 1):
    if get_stored_item_amount(save_id, item_id): #If the item is in the database
        if get_stored_item_amount(save_id, item_id) < amount: #Not enough items
            return "ItemQuantityError"
        if get_stored_item_amount(save_id, item_id) == amount: #Exactly enough items
            remove_stored_item(save_id, item_id)
        else:
```

```
        update_stored_item_amount(-amount, item_id, save_id) #Removes the ammount of that
                                                              item
    return True #Indicates that it was successful
  return "ItemQuantityError" #Indicates that there is an item quantity error
```

unequip_item():

```
def unequip_item(slot: str):
  #Checks if there is an item to unequip
  if get_slot_value(slot, save_id) != null:
    item = get_slot_value(slot, save_id)
    if (add_item(item, 1) == "FullInventoryError"): #Adds the item back to the stored_items
                                                    and checks if the inventory is full
      return "FullInventoryError"
    set_slot_value(slot, null, save_id) #Sets the slot back to null
    return True
  return True #If not item in slot it is unequipped
```

equip_item():

```
def equip_item(item: Resource):
        #Checks if the item is Equipable
        if not(item.is_class(Equipable)):
                return False
        #Gets the slot to equip it into
        if item.is_class(Armour):
                slot = item.body_part
        elif item.is_class(Weapon):
                slot = "weapon"
        else:
                equipped = false
                #Tries both charm slots
                for slot in ["charm1","charm2"]:
                        if get_slot_value(slot, save_id) == null and not(equipped):
                                set_slot_value(slot, item, save_id) #Equips item
                                remove_item(item) #Removes from stored_items
                                equipped = True
                if not(equipped):
                        unequip_item(slot)
                        set_slot_value(slot, item, save_id) #Equips item
                        remove_item(item) #Removes from stored_items
                return True
        unequip_item(slot)
        set_slot_value(slot, item, save_id) #Equips item
        remove_item(item) #Removes from stored_items
        return True
```

2.6.5 Testing Plan

Test #	Function	Parameters	Expected Outcome
12.5.1	add_item()	"test_item.tres", 2	Adds test item to the stored_items table.
12.5.2	add_item()	"test_item.tres", 3	As the item already exists it should add 3 to the amount.
12.5.3	add_item()	"test_weapon.tres", 1	As in the testing environment the max inventory size will be 1 and this should return "FullInventoryError"
12.5.4	remove_item()	"test_item.tres", 2	As more than the amount of the item is in the inventory it should subtract 2.
12.5.5	remove_item()	"test_item.tres", 10	"ItemQuantityError" as there isnt enough of the item in the database
12.5.6	remove_item()	"test_item.tres", 3	As exactly the amount is in the database the item entry should get removed.
12.5.7	remove_item()	"test_item.tres", 2	"ItemQuantityError" as there isnt any of the item in the database
12.4.1	unequip_item()	"head"	True and the head slot should remain as NULL
12.4.2	equip_item()	"test_helmet.tres"	True and the head slot should become "test_helmet.tres"
12.4.3	equip_item()	"test_helmet_2.tres"	True and the head slot should become "test_helmet.tres"
12.4.4	unequip_item()	"head"	"FullInventoryError"
12.4.5	unequip_item()	"head"	True as I will empty the inventory and the head should be NULL and the inventory should contain the helmet.

2.7 Player Character

This is my design for the physical player character and sprite.

2.7.1 Composition

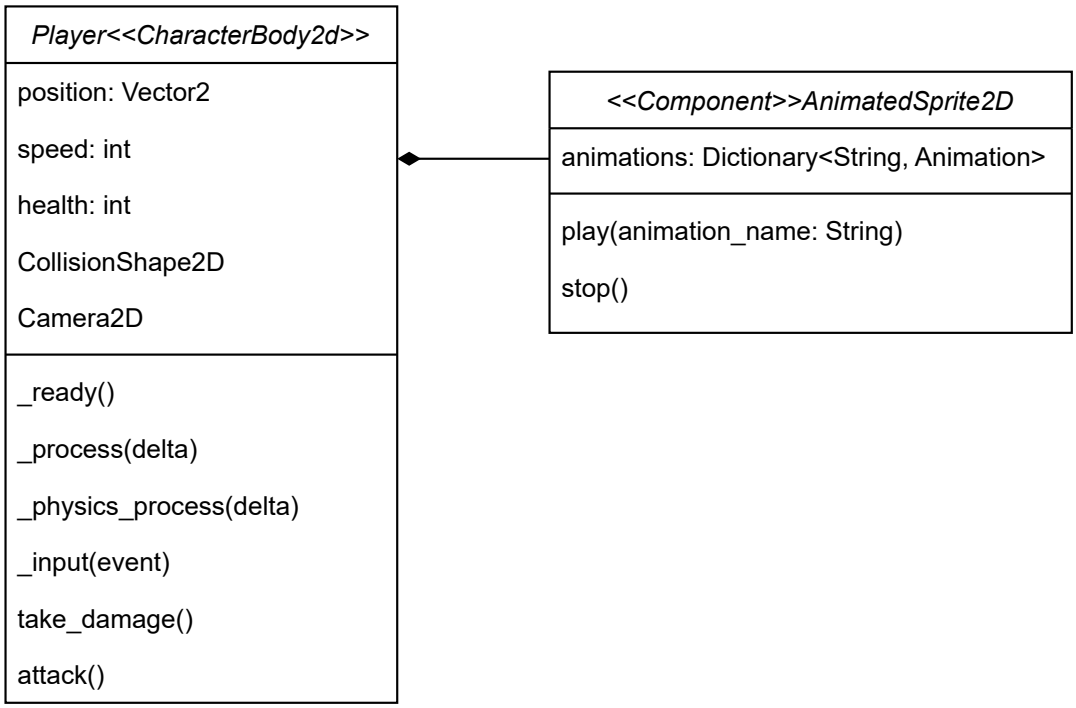

Figure 12: Player Class Diagram

The root node of the player which will contain all the child nodes will be Godot's CharacterBody2D as this will allow for a user controlled physics body. It will then have child nodes of CollisionShape2D(for collision detection), AnimatedSprite2D(for an animated character sprite) and Camera2D(for the player's view window to be centered on).
I have chosen to store the speed and health variables within the player class as they will reset/ be recalculated based of the equipment equipped.

2.7.2 Animations

I have an animation set for the player that includes 8 Directional top down animations for all player actions and so how I will decide which directional animation will be based of the last direction the player walked.
I will use a get_animation function in order to get the directional animation to play.

2.7.3 Help Screen

I have decided to add a simple Help Screen in order to allow the user to check the controls when they want a reminder. I will implement this by creating a seperate scene with a label and then when the help button is pressed I will pause the tree and instance the scene before waiting for the help button to be pressed again to unpause the scene tree and queue_free the label.

2.7.4 Algorithms

_ready():
The _ready() function gets called whenever the player is instantiated in a scene and so it will be used to setup

variables and the environment based on existing stuff.

```
def _ready():
        #Inventory calculates the speed based on any modifiers equipped.
        speed = Inventory.calc_speed()
        #Global Script calculates the health based on the player level and any modifiers
                                                    equipped.
        health = Global.calc_health()
```

_physics_process(delta):
The _physics_process(delta) function gets called every frame where delta is the time since the last fram and is usually used to deal with movement and physics processes.

```
def _physics_process(delta):
   direction = Input.get_vector("left", "right", "up", "down")
   velocity = direction * speed
   if direction:
      last_direction = direction
      anim.play(get_animation("run"))
   else:
      anim.play(get_animation("idle"))
   move_and_slide()
```

_input(event):

```
def _input(event):
   if event.is_action_pressed('attack'):
      attack()
   if event.is_action_pressed('help'):
      help_screen = preload("path/to/help/scene").instantiate()
      add_child(help_screen)
      get_tree().paused = True
      while not Input.is_action_just_pressed("help"):
         sleep(0.01)
      help_scene.queue_free()
      get_tree().paused = false
```

attack():

```
def attack():
   anim_player.play(get_animation("melee"))
   Inventory.get_weapon().attack(last_direction)
```

get_animation():

```
def get_animation(animation_type):
   anim = animation_type + "_"
   if direction.x:
      if direction.x == 1:
         anim += "r"
      else:
         anim += "l"
   if direction.y:
      if direction.y == 1:
         anim += "d"
      else:
         anim += "u"
   return anim
```

2.8 Weapon Hurtbox

I will use an Area2D node with a capsule shape CollisionShape2D node attached oriented in order to encompass the sword swing area and then I will rotate it around the player dependent on the direction of attack. It will have variables for the damage and damage_type

2.8.1 Algorithms

_ready(): This script is used to get the bodies overlapping and if they are enemies call their take_damage function using polymorphism to decide the effect this will have on the specific enemy.

```
def _ready():
    bodies = self.get_overlapping_bodies() #Get bodies in the area
    for body in bodies:
        if body.is_in_group("enemies"): #Damages bodies if they are an enemy.
            body.take_damage(damage, damage_type)
```

2.9 Dungeon Environment Design

2.9.1 Generic Tiles

I will use a TileMap node as that will allow me to import a spritesheet for a tilemap and define the properties of the tiles such as hitboxes and place them down easily. This will be used for criteria 3.1 and 3.2.

2.9.2 Chests

I will implement Chests, the basic structure of the chest will be a StaticBody2D with a circular area2D node to tell if the player is within range to open the chest aswell as the chest sprite. Each chest will have a key path for the key needed to unlock it. The chests will have a loot pool given by a dictionary where the key is the item path and the value relates to the ratio of getting it. There will also be a variable for the amount of items given by the chest and upon opening the chest will give that many items.

2.9.3 Doors

I will implement doors, the basic structure of the door will be using a StaticBody2D with an Area2D Node to detect if the player is infront of the door. If the player is infront of the door and if the I key is pressed the door will either dissappear revealing another room or switch the scene to another room depending on the purpose. I will export the variables for the scene to go to and wether it just dissappears or changes scene. This would fulfill all of criteria 3.5

2.9.4 Algorithms

Chest Script:
This will consist of the exported variables aswell as an input function that checks if the requirements to unlock the chest are met before generating the random items based on their probabilities which will be set up using an array and after the chest is opened then it will dissappear.

```
@export item_pool: dict
@export item_number: int
@export key_path: string

#Constructing the item_list
def _ready():
    item_list = []
    for item in item_pool: #For all items
      item_list += [item for x in range(item_pool[item])] #Adding on the amount relevant to
                                                          the ratio of the item

def _input(event):
    if event.is_action_pressed("enter") and Area2D.overlaps_body("player"): #Check if player
                                                        within range and pressed button
        if Inventory.remove_item(key_path,1) is bool: #Check if player has the key to unlock it
            for x in range(item_number):
                Inventory.add_item(item_list[randint(0,len(item_list)-1)])
          queue_free()
```

Door Script:
This consists of the variables and input function that checks if they right key is pressed then it checks if the

player is in the Area2D Node and if so will check if the player has the correct key perform the right action

```
@export change_scene: bool
@export scene_path: string
@export key_path: string

def _input(event):
    if event.is_action_pressed("enter") and Area2D.overlaps_body("player"): #Check if player
                                                within range and pressed button
        if Inventory.remove_item(key_path,1) is bool: #Check if player has the key to unlock it
            if change_scene:
                get_tree().change_scene_to_file(scene_path) #Change Scene
            else:
                queue_free() #Dissappear
```

2.10 Projectile Design

2.10.1 Overview

I will create two types of projectiles, one ranged type which will deal damage on impact and dissappear and one area of effect which will deal damage on an enemy or the player entering it and spawn next to the player/enemy that casts it. These projectiles will allow me to fulfill criteria 2.4, 2.5, 2.6, 4.7 and be used for the creation of boss enemies in 4.8.

2.10.2 Ranged

This will have the damage and damage_type variables to be used to call other entities take_damage functions, a variable so it only impacts once called attacking and a speed to determine how fast it goes. It will be comprised of a CharacterBody2D node and a AnimatedSprite2D node which will play the default animation until collision.
Its physics process will handle movement in the direction of rotation (as projectiles can be fired in any direction) and handle collisions.

2.10.3 Area Of Effect

This will have the damage and damage_type variables to be used to call other entities take_damage functions and a time variable for how long it lasts. It will be comprised of an Area2D node and an AnimatedSprite2D node which will play the default animation.
The ready function will be used to await the timer timeout before queue_freeing the area.
The _on_body_entered function will be linked from the Area2D and will deal damage if the body is an enemy or the player.

2.10.4 Algorithms

_physics_process(delta):
For the ranged projectile to handle movement in rotation direction and collisions.

```
def _physics_process(delta):
    #Rotated Movement
    velocity.x = speed * cos(rotation)
    velocity.y = speed * sin(rotation)
    if not attacking:
        move_and_slide()
    #Collisions
    for i in range(get_slide_collision_count()): #Loops through collisions
        if not attacking:
            attacking = true
            collision = get_slide_collision(i)
            collider = collision.get_collider()
            if collider.is_in_group("player") or collider.is_in_group("enemies"): #Damages what
                                                        it hits if its an enemy or the player
                collider.take_damage(damage, damage_type)
```

```
        CollisionShape2D.disabled = true #Disable collision shape so cant block other
                                                projectiles
        AnimatedSprite2D.play("impact")
        await AnimatedSprite2D.animation_finished #Wait for animation to finish
        queue_free()
```

_ready():
Will set the area of effect to dissappear after a certain time

```
def _ready():
    await get_tree().create_timer(time).timeout
    queue_free()
```

_on_body_entered(body):
This is called by the Area2D node upon a body entering it with that body passed as a parameter so we can make it take damage if it needs to.

```
def _on_body_entered(body):
    if body.is_in_group("enemies") or body.is_in_group("player"): # If its an enemy or player
                                                    it will damage It
        body.take_damage(damage, damage_type)
```

2.11 Enemy Design

2.11.1 Overview

For my basic enemies I have decided to go with different types of slimes representing different elements. The slimes will have animations and collisions aswell as a radius that they will detect the player and navigate towards them dealing damage upon impact. Each unique slime will deal a different damage type and have a different look.

2.11.2 Composition

The root node of the slimes will be a CharacterBody2D node to allow for a physics body that can be easily moved via the script. I will then have a CollisionShape2D for collision detection aswell as an Area2D node for detecting the player. It will contain an AnimatedSprite2D for the animations.

Identifier	Data Type	Justification
speed	Exported Integer	This allows slimes to have variable speeds to allow for difficulty increase.
health	Exported Integer	This allows for slimes to have variable healths to allow for difficulty increase.
damage	Exported Integer	This allows for slimes to deal variable damage to allow for difficulty increase.
damage_type	Exported String	This allows for slimes to deal different damage types to pose a different challenge.
direction	Vector2	This is used for direction to move in and the type of animation to play.
weaknesses	Exported List	Shows which damage_types to take more damage from.

2.11.3 Navigation

Upon researching I found the godot documentation on 2D navigation(6) and decided to use the NavigationAgent2D node to utilise the A* algorithm, this means that I would need to add a navigation layer to my tileset to show the areas that the navigation agent can use. I decided to use a Timer node that autostarts and repeats and update the navigation path on the timeout so that the player can evade the slimes to a certain extent. Ontop of this I used a circular CollisionShape2D with and Area2D node to detect when the player is within a certain range and pathfind towards them.

2.11.4 Animation

I decided to add idle animations for all of the 4 cardinal directions aswell as idle and hurt animations. I used a get_animation() function to get the relevant animation based on the direction.

2.11.5 Projectile Enemies

There will be some enemies that will shoot a projectile instead of navigating in the direction given by the NavigationAgent2D so that the targetting would only update every couple seconds. These enemies will be non moving enemies.

2.11.6 Algorithms

get_animation(animation_type):
This gets the animation based on the type and direction.

```
def get_animation(animation_type: String):
  if abs(direction.x) > abs(direction.y):
    if direction.x > 0:
      return animation_type + "_r"
    else:
      return animation_type + "_l"
  else:
    if direction.y > 0:
      return animation_type + "_d"
    else:
      return animation_type + "_u"
```

_ready():
This is ran on addition to the scene tree to add the slime to the enemies group.

```
def _ready():
   add_to_group("enemies")
```

_physics_process(delta):
Melee Enemies:
This checks if the player is within the detection range and if so moves towards them aswell as handling animations and detecting if the slime collides with the player so that the player will be damaged.

```
def _physics_process(delta):
   move = false
   for body in Area2D.get_overlapping_bodies(): #Checking if player in the detection area
      if body.name == "Player":
         move = true
   if not animating: #If not playing a different animation
      if move: #If the slime can move
         direction = NavigationAgent2D.get_next_path_position().normalized() #Getting the
                                                          direction of the next point on the
                                                          path
         velocity = direction * speed
         AnimatedSprite2D.play(get_animation("walk"))
      else:
         AnimatedSprite2D.play(get_animation("idle"))
   move_and_slide() #moving

   if can_attack: #If the attack cooldown is done
      for i in range(get_slide_collision_count()): #Loops through collisions
         collision = get_slide_collision(i)
         collider = collision.get_collider()
            if collider.is_in_group("player"): #Checks if collision is with the player
               collider.take_damage(damage, damage_type) #Deals damage
               can_attack=false
               AttackTimer.start() #Starts attack cooldown
```

Ranged Enemies:
This checks if the player is within detection range and if so will fire a projectile towards them.

```
def _physics_process(delta):
   var player_detected = false
   for body in Area2D.get_overlapping_bodies: #Checking if the player is in the detection
                                                Area
      if body.name == "Player":
```

```
        player_detected = true
    if not animating: #If not playing a different animation
        AnimatedSprite2D.play(get_animation("idle"))
        if player_detected and can_attack: # Will instantiate a projectile scene aimed at the
                                                player if it can attack and detects the
                                                player
            direction = NavigationAgent2D.get_next_path_position.normalized()
            var projectile = load(projectile_scene_path).instantiate()
            projectile.rotation_degrees = direction.angle()
            projectile.position = position + 20*direction
            projectile.damage = damage
            get_parent.add_child(projectile)
            can_attack = false
            AttackTimer.start() #Starts the attack cooldown
```

take_damage(damage, damage_type):
The take damage function will allow the enemies to take more or less damage based on weaknesses and update their health aswell as applying knockback and checking to see if the enemy should die.

```
def take_damage(damage, damage_type):
    player = get_tree().get_first_node_in_group("player")
    animating = true
    if damage_type in weaknesses: # If the enemy is weak to a specific damage type then they
                                            will take more damage
        health -= 2*damage
    else:
        health -= damage
    velocity = - 25 * to_local(player.global_position).normalized() # Move away from the
                                            player for knockback
    if health <= 0:
        queue_free()
    else:
        AnimatedSprite2D.play(get_animation("hurt"))
        await AnimatedSprite2D.animation_finished
        animating = false
```

_on_navigation_timer_timeout():
This will run when the navigation timer runs out and only update the navigation then to allow the enemies to not have perfect tracking so the player can avoid them easier.

```
def _on_navigation_timer_timeout() -> void:
    player = get_tree().get_first_node_in_group("player")
    NavigationAgent2D.set_target_position(player.global_position)
```

_on_attack_timer_timeout():
Thus function will run when the attack timer ends (1 second after an attack) to allow the enemy to attack again.

```
def _on_attack_timer_timeout():
    can_attack = True
```

2.12 Procedural Generation Design

2.12.1 Overview

3 Development & Testing

3.1 Database Development

I used a global autoloaded script database.gd in order to implement all of my functions for handling the database. Upon testing the functions I realised that the reset_password query was incorrect as it says UPDATE TABLE instead of just update.
I added all the prepared queries as private variables with strings in order to use db.query_with_bindings to sanitise and substitute inputs aswell as run the queries. This function would output wether the query succeeded or failed.
I then could use db.query_result in order to get the results of the query.

3.1.1 _ready()

```
func _ready() -> void:
    db.path = "res://game_data.db"
    db.open_db()
    if not db.query(_create_table_users):
        print("Error: users table unable to be created")
        return

    if not db.query(_create_table_save_data):
        print("Error: save_data table unable to be created")
        return

    if not db.query(_create_table_stored_items):
        print("Error: stored_items table unable to be created")
        return
    print("DONE")
```

Figure 13: _ready

In the database script db is declared using $SQLite.new()$ which is a wrapper class. I use the script to load the database and make sure all the necessary tables are present.
I also made it so that the database is closed when the script exits the tree so as to make sure all the data is saved properly.

3.1.2 Hashing

```
#Function for generating salt
func gen_salt() -> String:
    var salt = "string"
    var x = randi_range(5,10)
    for i in range(2**x):
        salt = j_hash(salt,str(i*randi_range(1,10)))
    return salt
```

Figure 14: gen_salt

```
#Function for hashing a password or challenge answer
func j_hash(string, salt):
    var hashedString = string
    #Repeating a consistent but unpredictable amount of times
    #On even rounds the password is sandwidged on odd rounds the salt is sandwidged
    #Alternating the use of sha256 and md5 but making sure to end on sha256 so the hash is a predictable length.
    for x in range(1,6*len(string)+1):
        if x % 2 == 0:
            hashedString = (salt.substr(x,hashedString.length()-x)+hashedString+salt.substr(0,x)).md5_text().sha256_text()
        else:
            hashedString = (hashedString.substr(0,x)+salt+hashedString.substr(x,hashedString.length()-x)).sha256_text().md5_text()
    return hashedString
```

Figure 15: hash

The hash and gen_salt functions implementation followed the pseudocode pretty faithfully apart from the fact I decided to not hash the number turned into a string as the salt doesnt have to be a certain length for the

code to work. I also decided to times the number by a random integer to increase randomness and the number of possible salts.

3.1.3 Login Functions

```
func login(username,password):
    db.query_with_bindings(_get_user_data,[username]) # Getting user data
    if len(db.query_result) == 0: # If user doesnt exist
        return "InvalidUsernameError"
    var user_data = db.query_result[0]
    var hashed_password = j_hash(password,user_data["salt"])
    if hashed_password == user_data["password"]: # Checking password hash against stored hash
        current_user_id = user_data["user_id"]
        return true
    return "IncorrectPasswordError" # If password doesnt match
```

Figure 16: login

This algorithm is a copy of the design algorithm just using godot's relevant functions instead. I further saved the current_user_id for ease of future queries.

```
#Function for creating a user
func add_user(username, password, answer):
    var salt = gen_salt() #Generating new salt
    var hashedPassword = j_hash(password, salt)
    var hashedAnswer = j_hash(answer, salt)
    if not db.query_with_bindings(_add_new_user,[username,hashedPassword,hashedAnswer,salt]): #Tries to add user
        return "InvalidUsernameError" #If user cannot be added then the username must be invalid
    return true
```

Figure 17: add_user

This algorithm is a copy of the design algorithm just using godot's relevant functions instead.

```
func reset_password(username, answer, password):
    db.query_with_bindings(_get_user_data,[username]) # Getting user data
    if len(db.query_result) == 0: # If user doesnt exist
        return "InvalidUsernameError"
    var user_data = db.query_result[0]
    var hashed_answer = j_hash(password,user_data["salt"])
    if hashed_answer == user_data["answer"]: # Checking the answer hash against the stored hash
        db.query_with_bindings(_reset_password,[password,username])
        return true
    return "IncorrectAnswerError" # If answer doesnt match
```

Figure 18: reset_password

This algorithm is a copy of the design algorithm just using godot's relevant functions instead.

3.1.4 Testing

```gdscript
func test_database():
    #Criteria 6.
    print("6.1:")
    print((len(db.gen_salt()) == 64) and (db.gen_salt() != db.gen_salt())) # Test 6.1.1
    print(len(db.j_hash("password", "salt")) == 64) # Test 6.1.2
    print(db.j_hash("password", "salt") == db.j_hash("password", "salt")) # Test 6.1.3
    print(db.j_hash("password", "salt") != db.j_hash("Password", "salt")) # Test 6.1.4

    print("6.3")
    print(db.add_user("Hyrule", "Password", "Answer")) # Test 6.3.1
    print(db.add_user("Hyrule", "Password", "Answer") == "InvalidUsernameError") # Test 6.3

    print("6.7,6.8,6.10")
    print(db.login("Hyru1e", "Password") == "InvalidUsernameError") # Test 6.7.1
    print(db.login("Hyrule", "Password")) # Test 6.7.2
    print(db.reset_password("Hyru1e", "Answer", "password") == "InvalidUsernameError") # Te
    print(db.reset_password("Hyrule", "answer", "password") == "IncorrectAnswerError") # Te
    print(db.reset_password("Hyrule", "Answer", "password")) # Test 6.8.3
    print(db.login("Hyrule", "Password") == "IncorrectPaswordError") # Test 6.7.3
```

Figure 19: test_database.gd

Test #	Function	Parameters	Expected Outcome	Actual Outcome
6.1.1	gen_salt()		random 256 bit hex string	Success
6.1.2	hash()	”password”, ”salt”	random 256 bit hex string	Success
6.1.3	hash()	”password”, ”salt”	the same random 256 bit hex string	Success
6.1.2	hash()	”Password”, ”salt”	random 256 bit hex string different from before	Success
6.3.1	add_user()	”Hyrule”, ”Password”, ”Answer”	True	Success
6.3.2	add_user()	”Hyrule”, ”Password”, ”Answer”	”InvalidUsernameError” as a user already exists with that username	Success
6.6.1	login()	”Hyru1e”, ”Password”	”InvalidUsernameError”	Success
6.6.2	login()	”Hyrule”, ”Password”	True	Success
6.7.1	reset_password()	”Hyru1e”, ”Answer”, ”password	”InvalidUsernameError”	Success
6.7.2	reset_password()	”Hyrule”, ”answer”, ”password”	”IncorrectAnswerError”	Success
6.7.3	reset_password()	”Hyrule”, ”Answer”, ”password”	True	Success
6.6.3	login()	”Hyrule”, ”Password”	”IncorrectPasswordError”	Success

Upon Testing I realised I needed a delete user function so that the user can be deleted.

I designed a simple SQL query and created a function to delete users.

```
var _delete_user = """
DELETE FROM users
WHERE username = ?;
"""
```

Figure 20: _delete_user

```
func delete_user(username, password):
    db.query_with_bindings(_get_user_data,[username])
    if len(db.query_result) == 0: # If user doesnt exist
        return "InvalidUsernameError"
    var user_data = db.query_result[0]
    var hashed_password = j_hash(password,user_data["salt"])
    if hashed_password == user_data["password"]: # Checking password hash against stored hash
        db.query_with_bindings(_delete_user,[username]) # Deleting User
        return true
    return "IncorrectPasswordError" # If password doesnt match
```

Figure 21: delete_user

Test #	Function	Parameters	Expected Outcome	Actual Outcome
6.9.1	delete_user()	”Hyrule”, ”Password”	IncorrectPasswordError	Success
6.9.2	delete_user()	”Hyrule”, ”password”	True	Success
6.6.4	login()	”Hyrule”, ”password”	InvalidUsernameError	Success

```
print(db.delete_user("Hyrule", "Password") == "IncorrectPaswordError") # Test 6.10.1
print(db.delete_user("Hyrule", "password")) # Test 6.10.2
print(db.reset_password("Hyrule", "Answer", "password") == "InvalidUsernameError") # Test 6.7.4
```

Figure 22: Test Remove User

Upon Testing the remove functions in the database I have updated the table queries to add ON DELETE CASCADE so that if the user gets deleted all their saves get deleted.

3.2 Login System Development

I used godot’s inbuilt label, button and line edit node’s in order to construct my forms. To each form I added an extra label in order to display Errors to the user.
I linked the buttons pressed signals to a script in order to determine what happens when the button is pressed and used variables to fetch and store the data from the line edit nodes.
I used node2ds in order to create groups of the nodes for more organisation and I kept the form layout mostly the same without some of the fancier unnecesary design elements from the mockup forms.

3.2.1 Login Form

Figure 23: Layout

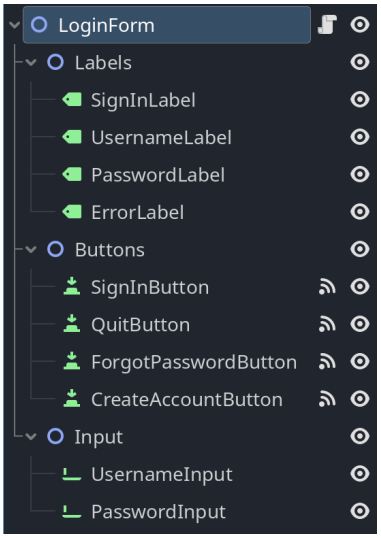

Figure 24: Structure

```gdscript
func _on_forgot_password_button_pressed() -> void:
    get_tree().change_scene_to_file("res://scenes/reset_password_form.tscn")


func _on_create_account_button_pressed() -> void:
    get_tree().change_scene_to_file("res://scenes/menu/create_account_form.tscn")


func _on_quit_button_pressed() -> void:
    global.quit()
```

Figure 25: button_pressed functions

These button functions are pretty simple as I only need to change scene or quit the game.

```gdscript
func _on_sign_in_button_pressed() -> void:
    var username = $Input/UsernameInput.text
    var password = $Input/PasswordInput.text
    var success = database.login(username, password)
    if not (typeof(success) == TYPE_BOOL and success == true):
        if success == "InvalidUsernameError":
            $Labels/ErrorLabel.text = "Invalid Username"
        elif success == "IncorrectPasswordError":
            $Labels/ErrorLabel.text = "Incorrect Password"
    else:
        get_tree().change_scene_to_file("res://scenes/menu/save_menu.tscn")
```

Figure 26: _on_sign_in_button_pressed

This is the function for when the sign in button is pressed it fetches the data and tries to login, displaying any errors it gets. If the login is successful theen it switches the scene to the save_menu scene.

3.2.2 Reset Password Form

Figure 27: Layout

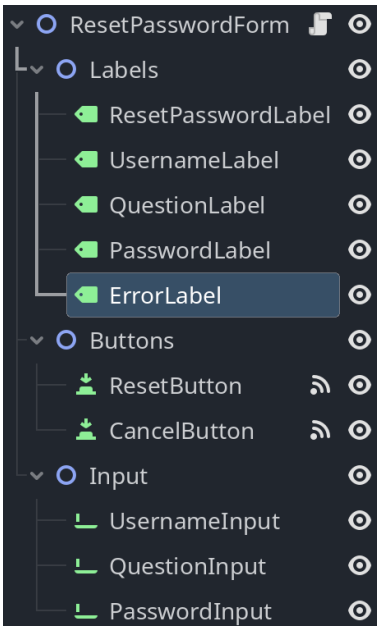

Figure 28: Structure

```gdscript
func _on_cancel_button_pressed() -> void:
    get_tree().change_scene_to_file("res://scenes/menu/login_form.tscn")
```

Figure 29: _on_cancel_button_pressed

This button function is pretty simple as I only need to change scene back to the login form.

```gdscript
func _on_reset_button_pressed() -> void:
    var username = $Input/UsernameInput.text
    var answer = $Input/QuestionInput.text
    var password = $Input/PasswordInput.text
    var success = database.reset_password(username, answer, password)
    if not (typeof(success) == TYPE_BOOL and success == true):
        if success == "InvalidUsernameError":
            $Labels/ErrorLabel.text = "Invalid Username"
        elif success == "IncorrectAnswerError":
            $Labels/ErrorLabel.text = "Incorrect Answer"
    else:
        get_tree().change_scene_to_file("res://scenes/menu/login_form.tscn")
```

Figure 30: _on_reset_password_button_pressed

This is the function for when the reset password button is pressed it fetches the data and tries to reset the password, displaying any errors it gets. If the reset is successful then it switches the scene to the login_form scene.

3.2.3 Create Account Form

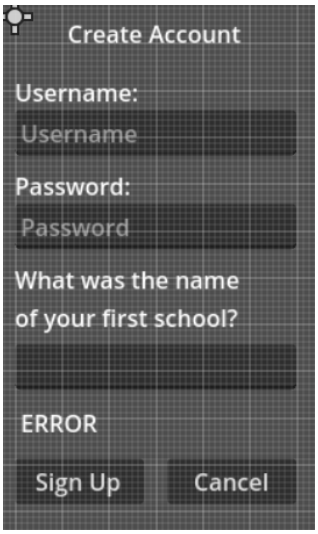

Figure 31: Layout

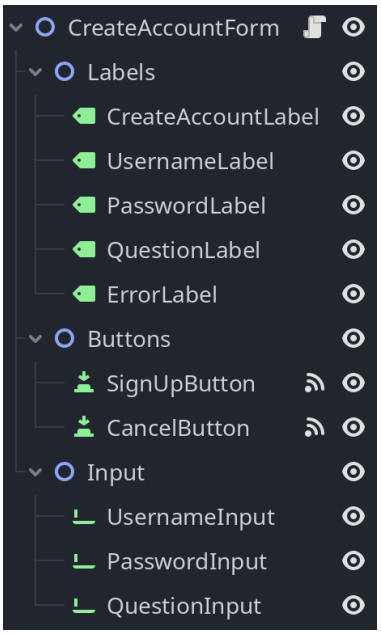

Figure 32: Structure

```
func _on_cancel_button_pressed() -> void:
    get_tree().change_scene_to_file("res://scenes/menu/login_form.tscn")
```

Figure 33: _on_cancel_button_pressed

This button function is pretty simple as I only need to change scene back to the login form.

```
func _on_sign_up_button_pressed() -> void:
    var username = $Input/UsernameInput.text
    var password =  $Input/PasswordInput.text
    var answer = $Input/QuestionInput.text
    var success = database.add_user(username, password, answer)
    if not (typeof(success) == TYPE_BOOL and success == true):
        if success == "InvalidUsernameError":
            $Labels/ErrorLabel.text = "Invalid Username"
    else:
        get_tree().change_scene_to_file("res://scenes/menu/login_form.tscn")
```

Figure 34: _on_reset_password_button_pressed

This is the function for when the create account button is pressed it fetches the data and tries to create the account, displaying any errors it gets. If the reset is successful then it switches the scene to the login_form scene.

3.3 Item Development

I will use resource scripts in order to implement the item classes and I will export the variables so that when I create new resources I can set the values.
In order to export the armour_type, body_part, charm_type, weapon_type and damage_type I will use an enum as it can only take one of the values in the list. This means the variables will take the form of an integer instead of a string.

3.3.1 Folder Structure

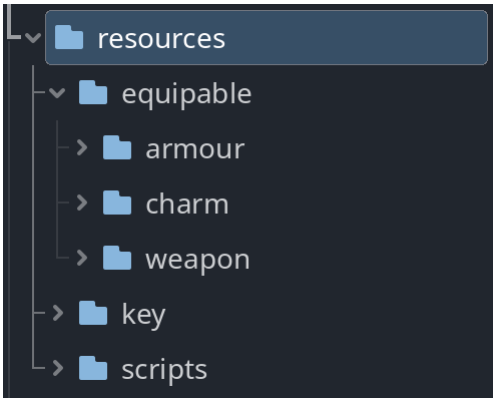

Figure 35: Folder Structure

I added more folders in order to organise the items into their groups aswell as keeping the resource scripts in a scripts folder.

3.3.2 Item

```
1  extends Resource
2
3  class_name Item
4
5  @export var stackable: bool
6  @export var description: String
7  @export var icon: Texture
```

Figure 36: Item Script

Figure 37: Item Exports

This shows the Item script and exported variables which I can set for each instance of that class including instances of classes that inherit from item.
I chose to remove the item_id as it seemed complicated to autoincrement it and enforce uniqueness and so I will store the file path in the item_id column in the database instead of an integer and so I updated the create_table_stored_items query in order to allow that.

3.3.3 Equipable

```gdscript
extends Item

class_name Equipable

@export var stat_boosts: Dictionary = {}

func _init():
    stackable = false
```

Figure 38: Equipable Script

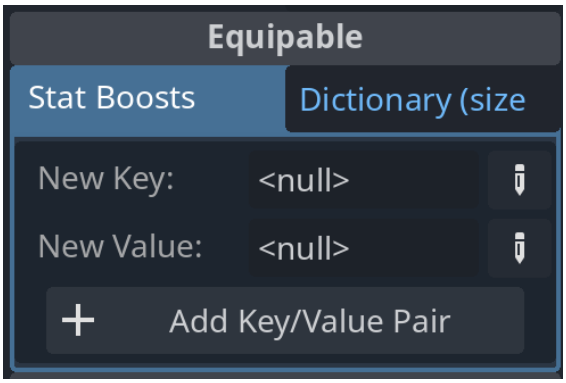

Figure 39: Equipable Exports

This shows the Equipable script and exported variables which I can set in any instance of this class or classes that inherit from it. I am using a dictionary to store stat boosts where the key is the stat and the value is the boost and these pairs can be added through the inspector. I set stackable to false by default as Equipable items will not be stackable.

3.3.4 Armour

```gdscript
extends Equipable

class_name Armour

@export var defense: int
@export_enum("Light", "Heavy") var armour_type: int
@export_enum("Head", "Chest", "Legs") var body_part: int
@export var set_id: int
```

Figure 40: Armour Script

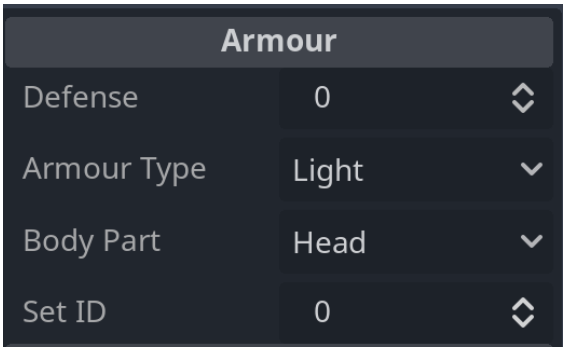

Figure 41: Armour Exports

This shows the Armour script and exported variables which I can set in any instance. I used an enum to represent the types of armour and body parts which it can be equipped on.

3.3.5 Charm

```gdscript
extends Equipable

class_name Charm

@export_enum("Ice", "Fire", "Cursed", "Divine","Poison") var charm_type: int
```

Figure 42: Charm Script

Figure 43: Charm Exports

This shows the Charm script and exported variables which I can set in any instance. I used an enum to represent the different charm types as you can only have one of them.

3.3.6 Weapon

```gdscript
extends Equipable

class_name Weapon

@export_enum("Ranged", "Magic", "Melee") var weapon_type: int
@export var attack_power: int
@export var attack_range: int
@export_enum("Ice", "Fire", "Cursed", "Divine","Poison") var damage_type: int
```

Figure 44: Weapon Script

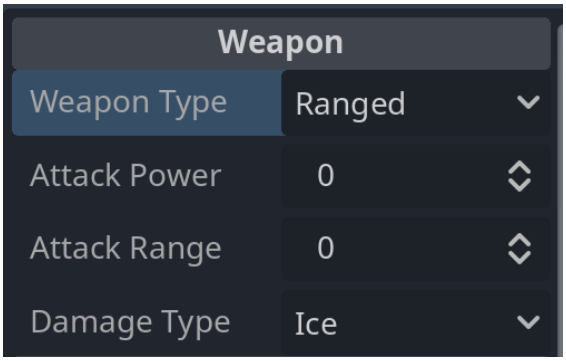

Figure 45: Weapon Exports

This shows the Weapon script and exported variables which I can set in any instance. I used an enum to represent the different weapon types and damage types so you can only select one

```gdscript
func attack(owner, direction: Vector2):
    #Load the hurtbox scene
    var hurtbox_scene = load(hurtbox_scene_path)
    var hurtbox_instance = hurtbox_scene.instantiate()
    hurtbox_instance.range = attack_range
    hurtbox_instance.damage = attack_power
    hurtbox_instance.damage_type = damage_type
    hurtbox_instance.rotation = direction.angle()
    #Add child
    owner.add_child(hurtbox_instance)
    #Hitbox lasts for a tenth of a second
    await owner.get_tree().create_timer(0.1).timeout
    hurtbox_instance.queue_free()
```

Figure 46: Weapon attack() function

This shows the weapon attack script that I implemented I changed the name from hitbox to hurtbox as that is more accurate and I had to use a timer in order to sleep for an amount of time that the hitbox will last for.

3.3.7 Key

```gdscript
extends "res://resources/scripts/item.gd"

class_name Key

@export var key_id: String
```

Figure 47: Key Script

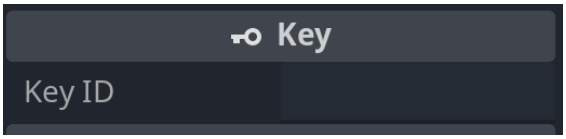

Figure 48: Key Exports

This shows the Key script and exported variables which I can set in any instance. The key ID will correspond to a door id and unlock that door.

3.3.8 Revision

Upon starting the inventory development I have decided to switch from enums to strings as the enums make it more complicated to access the string associated with the number.

3.4 Inventory Development

I used a seperate autoloaded script inventory.gd in order to implement the inventory functions.
I added functions to the database script in order to utilise the current_save_id script variable so that I dont have to input the variable every time I want to run a save_data or stored_items query. The functions execute the query passing in the current_save_id and return the query result.

3.4.1 Add Item

```
func add_item (item_id, amount):
    var item = load(item_id) # Loads the item
    if Database.count_stored_items() > max_inventory_size and not(item.stackable): # Full Inventory
        return "FullInventoryError"
    else :
        # Checks if you can stack the item and either adds a new entry or stacks it
        if len(Database.get_stored_item_amount(item_id)) != 0 and item.stackable: # If the item is in the database
            Database.update_stored_item_amount(amount, item_id)
        else :
            Database.add_stored_item(item_id, amount)
        return true
```

Figure 49: add_item()

The add_item function stayed mostly faithful to the pseudocode except I moved stuff arround for clarity.

3.4.2 Remove Item

```
func remove_item(item_id, amount):
    var amount_stored = Database.get_stored_item_amount(item_id)
    if len(amount_stored) != 0: # If the item is in the database
        amount_stored = amount_stored[0]["amount"]
        if amount_stored < amount: # Not enough items
            return "ItemQuantityError"
        elif amount_stored == amount: # Exactly enough items
            Database.remove_stored_item(item_id)
        else :
            Database.update_stored_item_amount(-amount, item_id) # Removes the ammount of that item
        return true # Indicates that it was successful
    return "ItemQuantityError"
```

Figure 50: remove_item()

The remove_item function is mostly the same as the pseudocode except I extracted some of the query data so I don't end up running repeat queries.

3.4.3 Unequip Item

```
func unequip_item(slot):
    # Gets the slot value
    var value = Database.get_slot_value(slot)
    # Checks if there is an item to unequip
    if value != null:
        if (add_item(value, 1) == "FullInventoryError"): # Adds the item back to the stored_items/
            return "FullInventoryError"
        Database.set_slot_value(slot, null) # Sets the slot back to null
        return true
    return true # If there is no item to unequip we have succeeded
```

Figure 51: unequip_item()

The unequip_item function is again close to the pseudocode except I extracted repeat queries to a variable.

3.4.4 Equip Item

```
func equip_item(item_id):
    var slot: String
    var item = load(item_id) # Loads the item
    # Checks if the item is Equipable
    if not(item is Equipable):
        return false
    # Gets the slot to equip it into
    if item is Armour:
        slot = item.body_part
    elif item is Weapon:
        slot = "weapon"
    else:
        if Database.get_slot_value("charm_1") == null:
            slot = "charm_1"
        else:
            slot = "charm_2"

    remove_item(item_id,1)
    var success = unequip_item(slot) # Checks the success of unequipping the item
    if not(success is bool) and success == "FullInventoryError":
        add_item(item_id,1)
        return "FullInventoryError"
    Database.set_slot_value(slot, item_id) # Equips the item
    return true
```

Figure 52: equip_item()

In the equip item function I decided to simplify the proccess of deciding which charm slot as it was unnecesarily complex and I also added a check for if the inventory is full making sure to still accept it if equipping the item leaves just enough room in the inventory.

3.4.5 Testing

```gdscript
func test_inventory():
    Database.add_user("test", "password", "answer")
    Database.login("test", "password")
    Database.add_new_save_data(1,true)
    Database.current_save_id = 1
    #Criteria 12
    print(Inventory.add_item("res://utils/test_item.tres", 2)) # Test 12.5.1
    print(Inventory.item_amount("res://utils/test_item.tres") == 2)
    print(Inventory.add_item("res://utils/test_item.tres", 3)) # Test 12.5.2
    Inventory.max_inventory_size = 1
    print(Inventory.add_item("res://utils/test_weapon.tres", 1) == "FullInventoryError") # Test 12.5.3
    print(Inventory.item_amount("res://utils/test_item.tres") == 5)
    print(Inventory.remove_item("res://utils/test_item.tres",2)) # Test 12.5.4
    print(Inventory.item_amount("res://utils/test_item.tres") == 3)
    print(Inventory.remove_item("res://utils/test_item.tres",10) == "ItemQuantityError") # Test 12.5.5
    print(Inventory.remove_item("res://utils/test_item.tres",3)) # Test 12.5.6
    print(Inventory.item_amount("res://utils/test_item.tres") == 0)
    print(Inventory.remove_item("res://utils/test_item.tres",2) == "ItemQuantityError") # Test 12.5.7
    print(Inventory.unequip_item("head")) # Test 12.4.1
    Inventory.add_item("res://utils/test_helmet.tres",1) #Test 12.4.2
    print(Inventory.equip_item("res://utils/test_helmet.tres"))
    print(Database.get_slot_value("head") == "res://utils/test_helmet.tres")
    print(Inventory.item_amount("res://utils/test_helmet.tres") == 0)
    Inventory.add_item("res://utils/test_helmet2.tres",1) # Test 12.4.3
    print(Inventory.equip_item("res://utils/test_helmet2.tres"))
    print(Inventory.item_amount("res://utils/test_helmet.tres") == 1)
    print(Inventory.unequip_item("head") == "FullInventoryError") # Test 12.4.4
    Inventory.remove_item("res://utils/test_helmet.tres",1) # Test 12.4.5
    print(Inventory.unequip_item("head"))
```

Figure 53: test_inventory()

Test #	Function	Parameters	Expected Outcome	Actual Outcome
12.5.1	add_item()	”test_item.tres”, 2	Adds test item to the stored_items table.	Success
12.5.2	add_item()	”test_item.tres”, 3	As the item already exists it should add 3 to the amount.	Success
12.5.3	add_item()	”test_weapon.tres”, 1	As in the testing environment the max inventory size will be 1 and this should return ”FullInventoryError”	Fail
12.5.4	remove_item()	”test_item.tres”, 2	As more than the amount of the item is in the inventory it should subtract 2.	Success
12.5.5	remove_item()	”test_item.tres”, 10	”ItemQuantityError” as there isnt enough of the item in the database	Success
12.5.6	remove_item()	”test_item.tres”, 3	As exactly the amount is in the database the item entry should get removed.	Success
12.5.7	remove_item()	”test_item.tres”, 2	”ItemQuantityError” as there isnt any of the item in the database	Success
12.4.1	unequip_item()	”head”	True and the head slot should remain as NULL	Fail
12.4.2	equip_item()	”test_helmet.tres”	True and the head slot should become ”test_helmet.tres”	Success
12.4.3	equip_item()	”test_helmet_2.tres”	True and the head slot should become ”test_helmet.tres”	Success
12.4.4	unequip_item()	”head”	”FullInventoryError”	Success
12.4.5	unequip_item()	”head”	True as I will empty the inventory and the head should be NULL and the inventory should contain the helmet.	Success

The add_item function failed to return ”FullInventoryError” as it only checked if the item wasnt stackable not if it wasnt stackabled and already stored so I updated the script.

```
func add_item (item_id, amount):
    var item = load(item_id) # Loads the item
    if Database.count_stored_items() >= max_inventory_size and not(item.stackable and item_amount(item_id) != 0):
        return "FullInventoryError"
    else :
        print("hi")
        # Checks if you can stack the item and either adds a new entry or stacks it
        if item_amount(item_id) != 0 and item.stackable: # If the item is in the database
            Database.update_stored_item_amount(amount, item_id)
        else :
            Database.add_stored_item(item_id, amount)
        return true
```

Figure 54: add_item() fixed

The remove_item failed due to a syntax error with the SQL statement so I edited the get_slot_value function to query for all values and then look it up I also updated the set_slot_value function to query the slot using a match case rather than bindings.

```
func get_slot_value(slot):
    db.query_with_bindings(_get_slot_values,[current_user_id, current_save_id])
    print(db.query_result)
    if len(db.query_result) == 0:
        return null
    return db.query_result[0][slot]
```

Figure 55: get_slot_value() fixed

```
func set_slot_value(slot, item_id):
    match slot:
        "head":
            db.query_with_bindings(_set_head_value, [item_id, current_save_id])
        "chest":
            db.query_with_bindings(_set_chest_value, [item_id, current_save_id])
        "legs":
            db.query_with_bindings(_set_legs_value, [item_id, current_save_id])
        "weapon":
            db.query_with_bindings(_set_weapon_value, [item_id, current_save_id])
        "charm_1":
            db.query_with_bindings(_set_charm_1_value, [item_id, current_save_id])
        "charm_2":
            db.query_with_bindings(_set_charm_2_value, [item_id, current_save_id])
    return db.query_result
```

Figure 56: set_slot_value() fixed

The unequip_item function failed due to not checking if success is a boolean before comparing it so I amended it.

```
func unequip_item(slot):
    # Gets the slot value
    var value = Database.get_slot_value(slot)
    # Checks if there is an item to unequip
    if value != null:
        var success = add_item(value, 1)
        if (not(success is bool) and success == "FullInventoryError"): #
            return "FullInventoryError"
        Database.set_slot_value(slot, null) # Sets the slot back to null
        return true
    return true # If there is no item to unequip we have succeeded
```

Figure 57: unequip_item() fixed

Retesting:

Test #	Function	Parameters	Expected Outcome	Actual Outcome
12.5.1	add_item()	"test_item.tres", 2	Adds test item to the stored_items table.	Success
12.5.2	add_item()	"test_item.tres", 3	As the item already exists it should add 3 to the amount.	Success
12.5.3	add_item()	"test_weapon.tres", 1	As in the testing environment the max inventory size will be 1 and this should return "FullInventoryError"	Success
12.5.4	remove_item()	"test_item.tres", 2	As more than the amount of the item is in the inventory it should subtract 2.	Success
12.5.5	remove_item()	"test_item.tres", 10	"ItemQuantityError" as there isnt enough of the item in the database	Success
12.5.6	remove_item()	"test_item.tres", 3	As exactly the amount is in the database the item entry should get removed.	Success
12.5.7	remove_item()	"test_item.tres", 2	"ItemQuantityError" as there isnt any of the item in the database	Success
12.4.1	unequip_item()	"head"	True and the head slot should remain as NULL	Success
12.4.2	equip_item()	"test_helmet.tres"	True and the head slot should become "test_helmet.tres"	Success
12.4.3	equip_item()	"test_helmet_2.tres"	True and the head slot should become "test_helmet.tres"	Success
12.4.4	unequip_item()	"head"	"FullInventoryError"	Success
12.4.5	unequip_item()	"head"	True as I will empty the inventory and the head should be NULL and the inventory should contain the helmet.	Success

Finally all my testing was a success.

3.5 Hurtbox Development

3.5.1 Layout

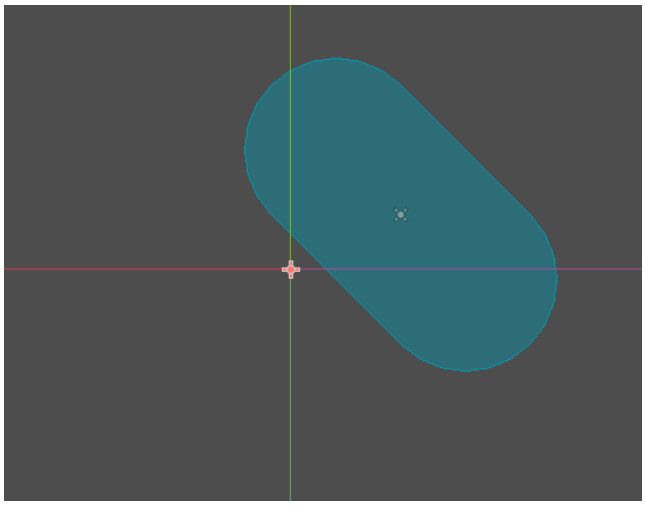

Figure 58: HurtBox Layout

I made the hurtbox like this as the center of the area2d node (the cross) can rotate allowing the actual collisionshape to rotate around the player depending on the direction of attack.

3.5.2 Script

```
extends Area2D

var damage: int
var damage_type: String
var range: int



func _on_body_entered(body: Node2D) -> void:
    if body.is_in_group("enemies"): # If its an enemy damages it
        body.take_damage(damage, damage_type)
```

Figure 59: Hurtbox Script

I changed the script from using the ready function to running every time a body enters the hurtbox as it allows the hurtbox to work for the duration of its existence rather than just at the start. Other than that the script is the same.

3.6 Player Development

3.6.1 Layout and Structure

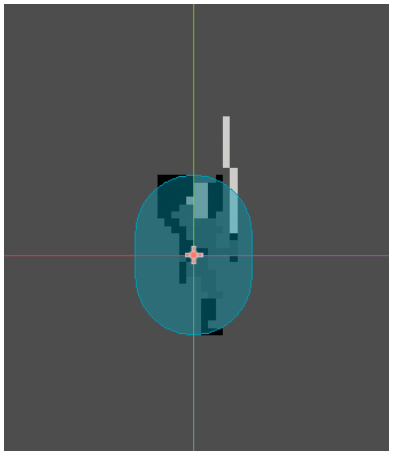

Figure 60: Layout

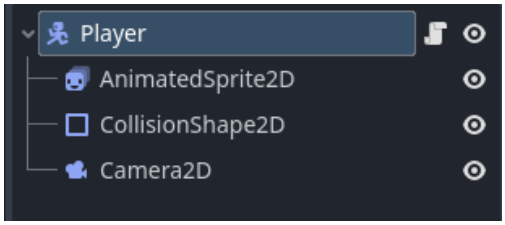

Figure 61: Structure

The structure is the same as was described in the class diagram and the layout is as such so that the player can interact with walls and enemy hits can register it.

3.6.2 _physics_proccess(delta):

```
func _physics_process(delta: float) -> void:
    # Get the direction
    var direction = Vector2(Input.get_axis("left", "right"),Input.get_axis("up","down"))
    # Velocity
    velocity = direction.normalized() * SPEED
    #Mocing/Idling if its not already animating
    if not animating:
        if direction:
            last_direction = direction
            $AnimatedSprite2D.play(get_animation("walk"))
        else:
            $AnimatedSprite2D.play(get_animation("idle"))
        move_and_slide()
```

Figure 62: _physics_process():

I decided to add an animating flag for use in this script that will be flagged when animations that cannnot be interrupted are playing (e.g. attacks) so that the player will not move or switch animations during that. Other than that the function is the same as the designed function with a slightly different method of getting direction.

3.6.3 Player Animation

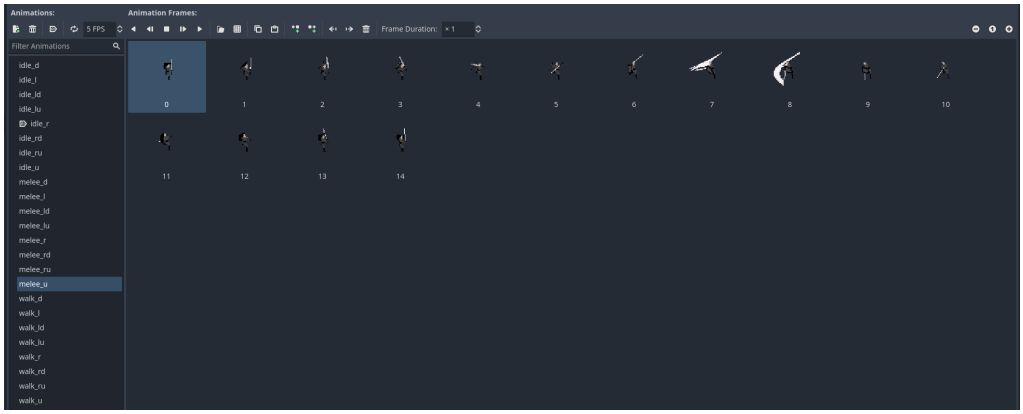

Figure 63: Animations

I added the animation frames into seperate animations in the Player's animationPlayer naming them such that the different directional animation names can be got using the get_animation() function.

```
func get_animation(animation_type: String):
    var anim = animation_type + "_"
    if last_direction.x:
        if last_direction.x == 1:
            anim += 'r'
        else:
            anim += 'l'
    if last_direction.y:
        if last_direction.y == 1:
            anim += 'd'
        else:
            anim += 'u'
    return anim
```

Figure 64: get_animation():

3.7 Dummy Development

As part of my developmental testing I deciced to create a simple dummy to test weapons on. The dummy consists of a StaticBody2D with a CollisionShape2D and a Sprite2D it also has a simple script for taking damage which I have set to print damage type and damage to the console.

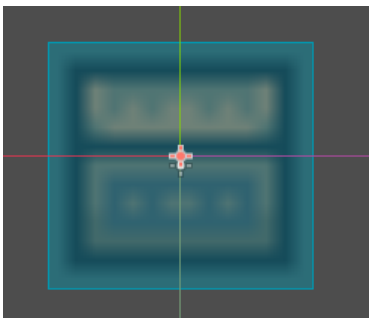

Figure 65: Layout

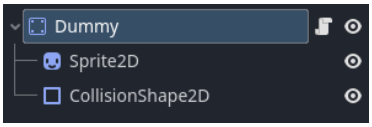

Figure 66: Structure

```
extends StaticBody2D

func _ready() -> void:
    add_to_group("enemies")

func take_damage(damage, damage_type):
    print(damage)
    print(damage_type)
```

Figure 67: Script

3.8 Dungeon Environment Development

3.8.1 Door

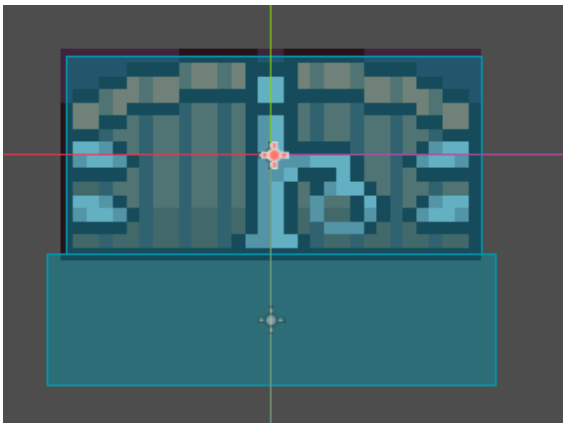

Figure 68: Layout

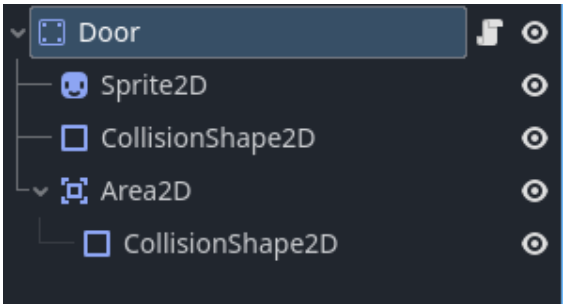

Figure 69: Structure

The layout and structure of the door is as shown above, mostly the same as was designed making sure to get the area2d infront of the door for opening.

```gdscript
extends StaticBody2D

@export var change_scene: bool
@export var scene_path: String
@export var key_path: String
@export var locked: bool

func _input(event: InputEvent) -> void:
    if event.is_action_pressed("interact"): # Check pressed button
        for x in $Area2D.get_overlapping_bodies():
            if x.name == "Player": # Check the player is within range
                if not(locked) or Inventory.remove_item(key_path,1) is bool: # Check if the player has the key and removes it if the door isnt locked
                    if change_scene:
                        get_tree().change_scene_to_file(scene_path) # Change Scene
                    else:
                        queue_free()
```

Figure 70: Script

I added a variable that detemines wether or not the door is locked aswell as looping through the overlapping bodies instead of checking if it overlaps a certain body as that was easier to implement.

3.8.2 Chest

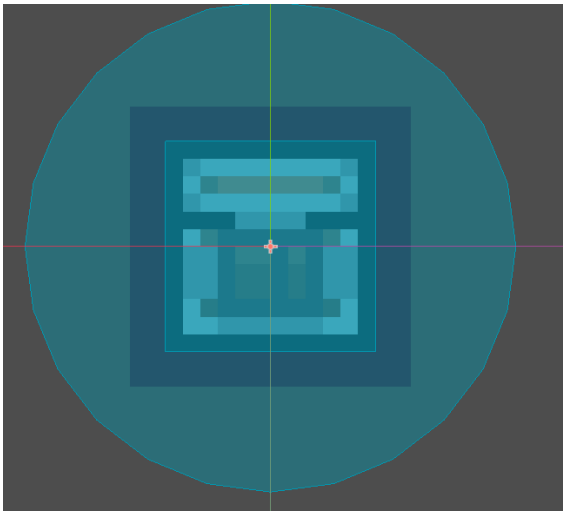

Figure 71: Layout

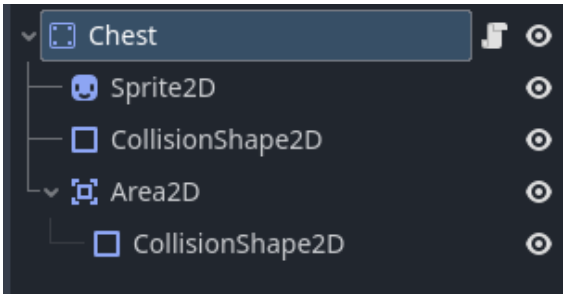

Figure 72: Structure

The layout and structure of the chest are mostly the same as was designed with the detection area being a circle around the chest.

```
extends StaticBody2D

@export var item_pool: Dictionary
@export var item_number: int
@export var key_path: String
@export var locked: bool
var item_list: Array



func _ready() -> void:
    for item in item_pool:
        var add_list = []
        for x in item_pool[item]:
            add_list.append(item)
        item_list.append_array(add_list)

func _input(event):
    if event.is_action_pressed("interact"):
        for i in $Area2D.get_overlapping_bodies():
            if i.name == "Player": # Check the player is within range
                if not(locked) or Inventory.remove_item(key_path,1) is bool: # Check if the player has the key and removes it if the door isnt locked
                    for j in range(item_number):
                        Inventory.add_item(item_list[randi_range(0,len(item_list)-1)],1)
                    print(item_list)
                    queue_free()
```

Figure 73: Script

The chest script is has a couple changes from the design, the addition of a locked flag aswell as making the item list a script variable so that it can be accessed from the input function and constructing the list without the use of pythonic list comprehension. I also switched from directly seeing if the player body overlaps to looping through overlapping bodies again as this wasy easier to implement and works the same.

3.9 Enemy Development

3.9.1 Layout and Structure

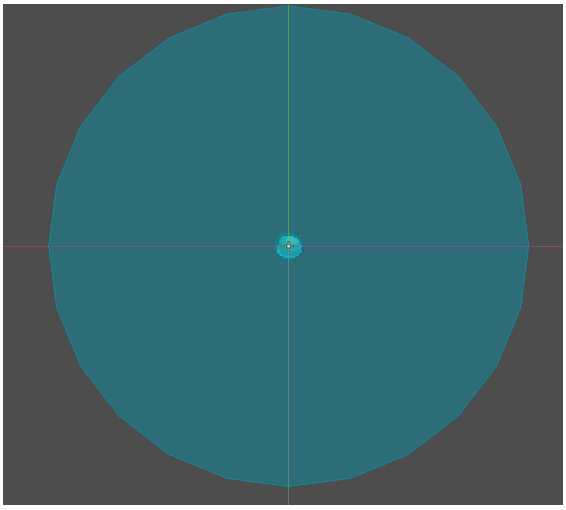

Figure 74: Layout

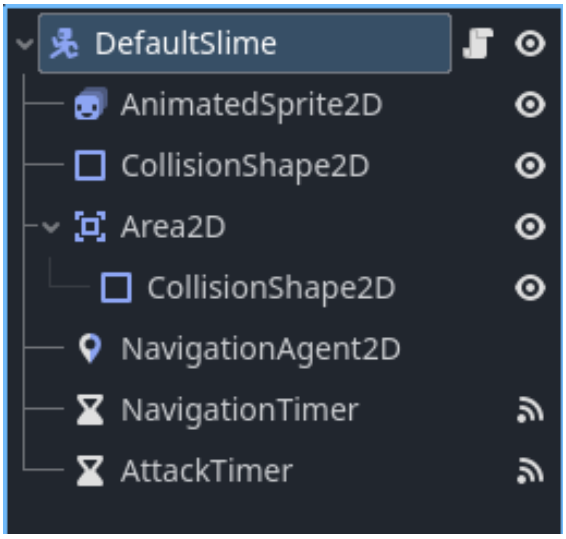

Figure 75: Structure

The layout and structure are as shown above with all the relevant timer nodes, the larger collision shape is the area2D for detecting the player whereas the smaller one handles physics collisons. The NavigationAgent2D handles the navigation and the AnimatedSprite2D handles animation.

3.9.2 General Script

```
extends CharacterBody2D



var direction = Vector2(0,1)
@export var speed:int = 20
@export var health: int = 10
@export var damage: int = 1
@export var damage_type: String = "normal"
@export var weaknesses: Array
var animating: bool
var can_attack = true

func get_animation(animation_type: String):

func _ready():

func _physics_process(delta: float) -> void:

func take_damage(damage, damage_type):

func _on_navigation_timer_timeout() -> void:

func _on_attack_timer_timeout() -> void:
```

Figure 76: General Script and Variables

```
func _ready():
    add_to_group("enemies")
```

Figure 77: _ready():

```
func _on_navigation_timer_timeout() -> void:
    var player = get_tree().get_first_node_in_group("player")
    $NavigationAgent2D.set_target_position(player.global_position)

func _on_attack_timer_timeout() -> void:
    can_attack = true
```

Figure 78: timer_timeouts

3.9.3 _physics_process(delta):

3.9.4 take_damage(damage, damage_type):

3.9.5 Animation

4 References

REF#	Date	Topic/Abstract	Type	URL or BOOK reference	How I used this
1	1/6/24	Research/ Existing Solutions	video games store, online	Steam (The Binding of Isaac)	One of the exisiting solutions I researched.
2	15/6/24	Research/ Existing Solutions	video games store, online	Steam (Dead Cells)	One of the existing solutions I researched
3	15/6/24	Research/ Existing Solutions	youtube video, online	Youtube (Motion Twin)	A dev log for an existing solution.
4	15/6/24	Research/ Existing Solutions	blog, online	robertheaton.com	An existing algorithm I researched.
5	25/11/24	Research/ Exisiting Solutions	video games store, online	Nintendo Store (Breath of The Wild)	One of the existing solutions I researched
6	04/03/25	Design/ Enemy Design	Game Engine Documentation, online	Godot Docs (2D navigation)	The documentation for godot's navigation nodes in 2D

5 Code Listings

```
extends Node

var max_inventory_size = 100


func item_amount(item_id: String):
	var query = Database.get_stored_item_amount(item_id)
	if len(query) == 0:
		return 0
	return query[0]["amount"]



func add_item (item_id, amount):
	if amount == 0:
		return true
	var item = load(item_id) # Loads the item
	if Database.count_stored_items() >= max_inventory_size and not(item.stackable and
	    item_amount(item_id) != 0): # Full Inventory
		return "FullInventoryError"
	else :
		# Checks if you can stack the item and either adds a new entry or stacks it
		if item_amount(item_id) != 0 and item.stackable: # If the item is in the database
			Database.update_stored_item_amount(amount, item_id)
		else:
			Database.add_stored_item(item_id, 1)
			add_item(item_id,amount-1)
		return true


func remove_item(item_id, amount):
	var amount_stored = item_amount(item_id)
	if amount_stored != 0: # If the item is in the database
		if amount_stored < amount: # Not enough items
			return "ItemQuantityError"
		elif amount_stored == amount: # Exactly enough items
			Database.remove_stored_item(item_id)
		else :
			Database.update_stored_item_amount(-amount, item_id) # Removes the
			    ammount of that item
		return true # Indicates that it was successful
	return "ItemQuantityError"


func unequip_item(slot):
	# Gets the slot value
	var value = Database.get_slot_value(slot)
	# Checks if there is an item to unequip
	if value != null:
		var success = add_item(value, 1)
		if (not(success is bool) and success == "FullInventoryError"): # Adds the item
		    back to the stored_items/checks if the inventory is full
			return "FullInventoryError"
		Database.set_slot_value(slot, null) # Sets the slot back to null
		return true
	return true # If there is no item to unequip we have succeeded


func equip_item(item_id):
	var slot: String
	var item = load(item_id) # Loads the item
	# Checks if the item is Equipable
	if not(item is Equipable):
		return false
	# Gets the slot to equip it into
	if item is Armour:
		slot = item.body_part
	elif item is Weapon:
		slot = "weapon"
	else:
		if Database.get_slot_value("charm_1") == null:
			slot = "charm_1"
		else:
			slot = "charm_2"
	remove_item(item_id,1)
```

```
73          var success = unequip_item(slot) # Checks the success of unequipping the item
74          if not(success is bool) and success == "FullInventoryError":
75                  add_item(item_id,1)
76                  return "FullInventoryError"
77          Database.set_slot_value(slot, item_id) # Equips the item
78          return true
79
80
81  func _ready() -> void:
82          equip_item("res://resources/equipable/weapon/test_weapon_magic_area.tres")
```