**Documented Design:**

**Contents:**

2-7 - [Object Design](#_Object_Design:)

8-19 - [Data/Program Flow Diagrams](#_Data_Flow_Diagrams:)

8-12 - [Main Program Overview](#_Main_Program_Overview:)

13-14 - [Floor Generation](#_Floor_Generation:)

15-19 - [Gameplay Loop](#_Gameplay_Loop:)

# **Object Design:**

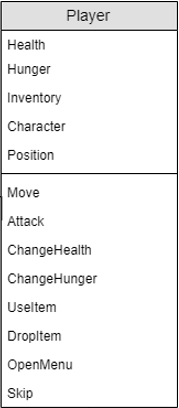
A close up of text on a white background

Description automatically generated

The entity relationship diagram above demonstrates the idea that in my program, there will be a total of six main object types: Enemies, Items, Rooms, Floors, the Inventory, and the Player. The diagram shows how the objects will be able to interact, in that there will be a single Player that accesses many Floors. On each Floor, there will be many Rooms. For each Room, there will be many Items and Enemies. As well as this, the Player will have access to the single Inventory, which will contain many Items.

The relationship between Floors, Rooms, Items, and Enemies demonstrates the way in which the Floors are going to be generated, as well as giving an idea as to how the program will act in general.

The number of Enemies and Objects that will be generated in each Room will be likely to be random, meaning there is a possibility that there will not be any of either in a single Room. As the user progresses through the Floors, it is likely that the number of each that can be generated will increase.

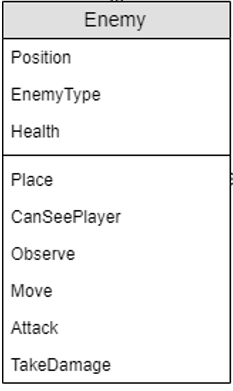
Player:

The player object represents the player character, with attributes that represent the player’s health, hunger, inventory, character, and position.

* Health is the value that dictates when the player loses, as when the health reaches 0, the player is considered dead and the run ends.
* Hunger reduces every turn after the player has performed an action. While hunger is above 10, the player would regenerate health at the end of every turn. This stops when the hunger is equal to or below 10. When hunger equals 0, the player would lose health at the end of every turn.
* The Inventory would also be an object of its own, where there is a set number of items that can be help. There could also be a potential system where the player can only hold one of each type of item. The inventory is an object so to neaten the “player” object.
* The Character attribute is set by the player selecting which character they want to use. This would dictate the base health, attack, etc. of the player. For example, a “tank” kind of character with low attack but high health or a “glass cannon” with high attack and low health.
  + Base values for:
    - Health
    - Attack
    - Hunger
    - Range
    - Sprite
* Position indicates where the player is on the floor grid.

The player will be able to perform various actions, represented as methods for the player object.

* The Move method will be used to change the player’s position. It will check that the space to be moved into is a valid tile and do so if it is.
* The Attack method will be used in order to deal damage to enemies within the player’s range.
* ChangeHealth will be used to change the player’s Health attribute. This will be used when the player is damaged, uses a healing item, or in accordance with the value of the Hunger attribute.
* ChangeHunger will be used to change the player’s Hunger attribute. This will be used when the player uses an item to increase their Hunger and at the end of their turn to decrease Hunger slightly.
* UseItem will be used to use an item in the inventory. Items that can be used will be one-time use items that have effects like increasing the health or hunger stat, dealing damage to all enemies in the room, teleporting the player to another area of the floor, etc.
* DropItem will drop an item to the player’s feet, if the tile is empty.
* OpenMenu will open the menu to the player, allowing them to save and quit, change options, see their inventory, etc.
* Skip will simply skip the player’s turn without performing an action.

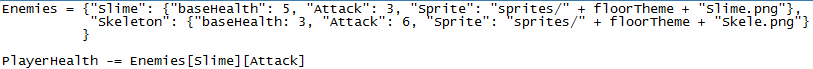
Enemy:

Enemies function similarly to the player with attributes, but do not have the inventory or the hunger attributes that the player does. Potentially the enemies may be able to hold an item such as a weapon. They will make use of a dictionary/library in a similar way to the Player Character attribute in order to assign sprites, statistics, and abilities. The enemies on the floor will be stored as an array so that the program will go through the array to perform each enemy’s turn.

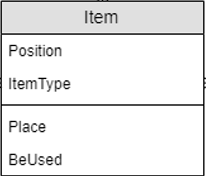
* Position is used to show where the enemy is and to detect if it can attack the player.
* EnemyType is used similarly to how Character is for the Player in order to assign the enemy’s statistics, such as its health, attack, etc. This will be done using a library/database, and will make use of the current floor number so to increase the statistics of the enemies to make later floors harder.
* Health represents the health of that specific enemy and is unique to each enemy, therefore it cannot be simply found from the library/database and much be assigned to each enemy on the floor.

As with the attributes, enemies share a few methods with the Player object. However, the enemies also can make use of state machine AI in order to make gameplay more complex and interesting for the player.

* The Place method is used in order to place the enemy in the floor after it has been generated, generating what type of enemy it is, assigning its starting health, and giving it a position on the grid.
* CanSeePlayer is a method that will return a Boolean value. A check will be made before any other actions are performed to see if the enemy is in the player’s line of sight. If not, False is returned and the enemy will do nothing. Otherwise, if True is returned, the enemy will perform an action based on that enemy type’s actions and attributes.
* If the enemy is a more complex one, every turn that the enemy can see the player, it will “observe” the player. This will inform decisions based on the enemy’s type and statistics. For example, if it sees the player has just killed an enemy of the same type as it, it may run away rather than attack the player. This will be performed using state machine AI.
* The Move method will allow the enemy to move to an available space.
* If within range on the player, the enemy can perform the Attack method so to inflict damage on the player.
* The TakeDamage method will be called when the player attacks the enemy, or if they take damage in any other way. Once the enemy’s health has reached 0, the enemy will die. This will be achieved by removing the enemy from the enemy array. When the enemy dies, it will drop a certain amount of money based on the enemy’s base money drop and the current floor number.



Rough example of how the enemy library will work.

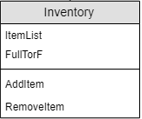
Item:

The Item object encompasses the items that the player is able to pick up, and will make use of libraries or databases in order to assign the item’s purpose. Items will be randomly generated on the grid of the floor after rooms have been generated and connected by corridors. The number and quality of items generated will increase as the floor number increases.

* Position will dictate the tile on the floor that the item takes up. When the item is in the inventory, Position will have a value of Null.
* ItemType will make use of a dictionary/database in order to assign the item’s effect, as well as its sprite. The floor number will be used in order to scale the items to have stronger effects as the player progresses. For example, a healing potion found on the first floor may heal 20 health while one found on the twentieth floor may heal 60. As well as the variety of item effect, the ItemType will be used to assign the sprite to be used.

Due to the fact that each item will have its effect assigned by a dictionary, there is a single “usage” method.

* Place is used when generating the floor, as well as when the player drops an item.
* BeUsed performs the action of the item, based on the effect found from the library/database. For example, increase the player’s health by 20 points.

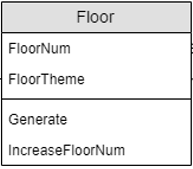
Inventory:

The inventory object will be used to store the items that the player has collected, and will be used in order to simplify the Player object. It will take the form of various lists based on item type. This will cause the process of checking for certain item types faster, based on their ItemType attribute, as the items themselves will be objects as well.

* ItemList will be the attribute wherein the items in the inventory are stored. This will be done through either the use of a two-dimensional array or through multiple arrays based on the item’s type.
  + Item Types include:
    - Weapons
    - Armours
    - Healing Items
    - Food
    - Single Use Items
    - Passive Items
* FullTorF is the attribute that will specify if the inventory is full or not based on a maximum inventory size.

As the inventory will be formed of arrays, the methods are based around this.

* AddItem will add an item acquired by the player to the inventory based on its type.
* RemoveItem will remove an item from the inventory. This will be used when an item is used or dropped.

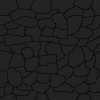
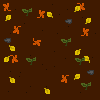
Floor:

The Floor object represents the grid-based floor of the game. A new floor is generated every time the player advances and each one is randomly generated so to make the game feel less repetitive. This means that no two runs will be the same. However, there may be specified structures so to give the player something familiar to find occasionally. A “staircase” tile will be generated, which is the tile the player must find and stand on in order to advance.

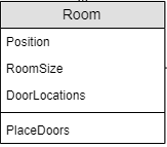
* FloorNum is the number of floors currently seen by the player, meaning that the first floor generated will be “Floor 1”. This attribute will be used in ways when generating each floor so to allow the difficulty of the game to scale as the player advances.
  + FloorNum will dictate:
    - The size of the grid used to generate the floor.
    - The number of rooms the floor has.
    - The number and strength of enemies and items on that floor.
* FloorTheme is simply an attribute used to decide which sprites are used for that floor to give in a theme. This is to further increase variety for the player.
  + Floor themes could include:
    - Forest.
    - Icy cave.
    - Cave.
    - Wasteland.
  + Sprites that would be changed include:
    - Floor tiles.
    - Wall tiles.
    - Enemies.

As the player advances to a new floor, the first thing that will be done is the generation of a new floor.

* The Generate method is the method that will be used in order to procedurally generate the new floor. This will be performed by generating the rooms on the floor, as well as various “corridor nodes”. These nodes will then be connected with the doors of the rooms in order to create a fully linked floor with no areas that are inaccessible. Once this has been generated, enemies, items, and the player itself will be generated according to the Floor Num.
* The IncreaseFloorNum method simply increases the FloorNum attribute, and will be called once the player has advanced from the previous floor.



Rough examples of floor themes.  
Left: Forest  
Right: Cave

Room:

The Room object will be used to represent each rectangular section of open tiles. These will be the areas in which the player, enemies, and items will be placed, as well as the exit staircase. The rooms will have doors, which will be used as stated previously to connect the rooms together in order to create an interconnected system of rooms and corridors.

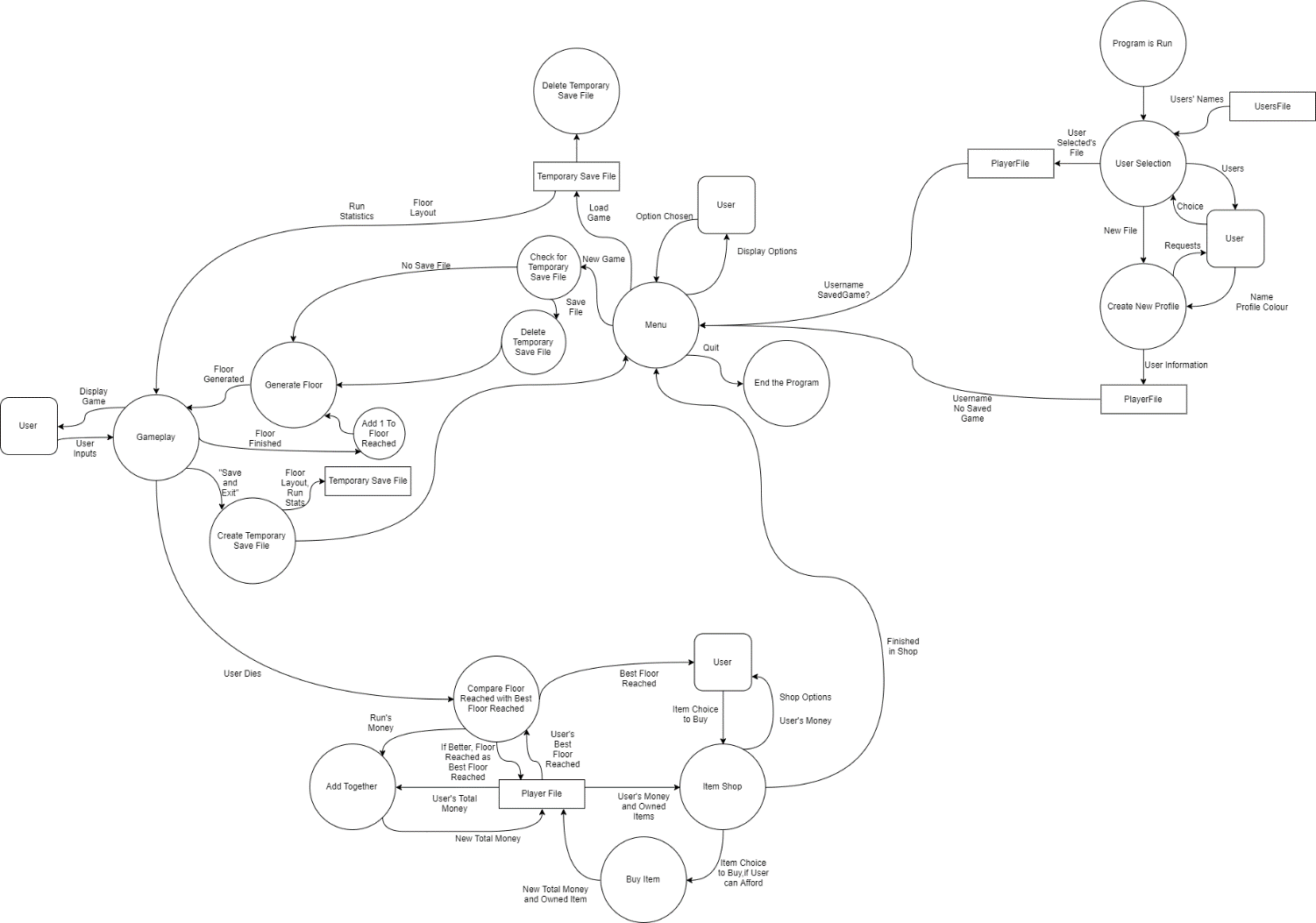
* Position indicates the top-left corner of the room and will be used when generating the floor to “carve” the room out.
* RoomSize is also used when generating the floor. It is the size of the room, both horizontally and vertically. This will be used when first generating the rooms’ locations in order to validate that they do not overlap over the edge of the floor’s grid.
* DoorLocations will be an array containing the coordinates for each door on the room’s edge. These doors will be used in order to generate corridors between this room and others.

Similarly to how the Inventory is used to simplify the Player object, the Room object exists mostly to have a simple way for storing information about the floor, without having to store it in the Floor object. The Floor object will contain an array of Room objects in order so that each may have its attributes stored in an easier to understand way, as well as to allow each to perform methods one at a time.

* The PlaceDoors method will be used in order to place doors along the edge of the room. This will include validating that no doors are adjacent to others, on the corner of the room, or right on the edge of the floor grid. Once the doors have been assigned, the tile one out from the door on the edge of the room will be allocated as a “corridor node” in order to connect them.

# **Data/Program Flow Diagrams:**

# **Main Program Overview:**

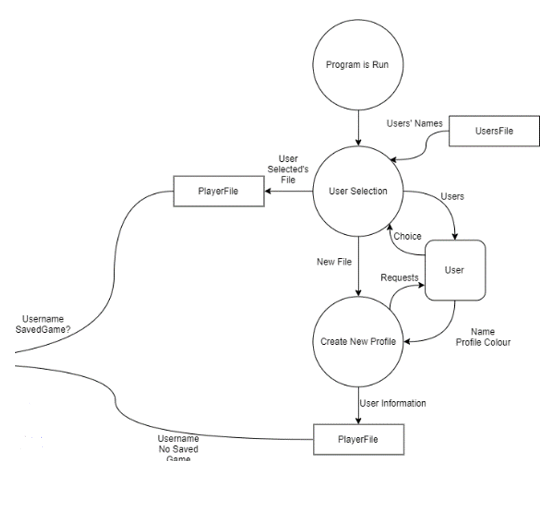
of

For the program, there are four main sections. These can be defined as the point when the program is first run, the main menu of the program, the main gameplay/floor generation loop, and the processes for when the user is defeated. These sections are mostly independent from each other, simply passing information on to the next to dictate what should be done in that section.

The program will make use of three different kinds of files:

* The User File, which will contain the names of every user registered with a profile in the program.
* The unique Player File, which will contain the unique information pertaining to a specific user, such as their current best run, currency, etc.
* The Temporary Save File, which will be used in the event that the user decides to save and quit the game during a run.

Booting the Program:



When the program is run, the first process that will be performed is a user selection screen. This will take information from a file containing the information on the users registered in order to display these users to the current user of the program. The current user will have the option to either select one of the displayed users or to create a new profile.

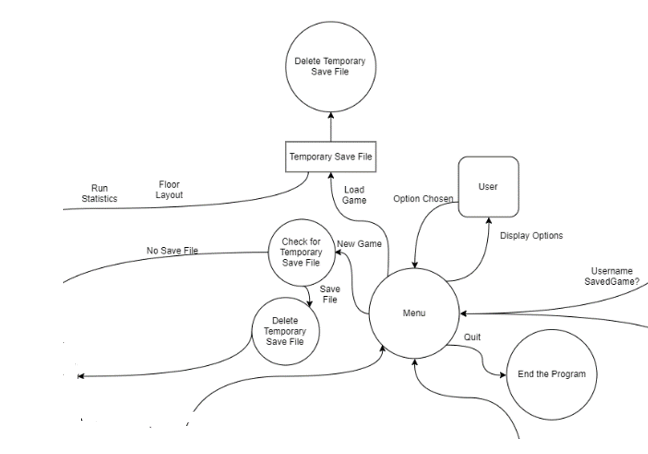
If the user selects to create a new profile, they will be asked what they want their profile name to be, as well as potentially what colour they want to be associated with it. This information will be stored in a separate file created for that user, known as their “PlayerFile”. This will be unique for each user, and will be where the information regarding their runs, current saved game, etc. will be stored. The fact that the current user has no saved game will be stored and used later on.

If the user selects a profile that already exists, the program will check that player’s file in order to find and store information saved to their profile. For example, their current currency, whether or not they have a saved game, etc.

This process will be that which allows the program to have multiple users with multiple levels of progress simultaneously, as well as allowing a player to stop using the program and come back to it later with no loss of progress.

The files could be replaced with a database containing fields pertaining to each piece of information that may be saved, such as the current floor’s grid, or the player’s total currency.

Main Menu:



After selecting/creating a profile, the user will have the main menu displayed to them. Here, they would have the option to perform various actions. The main options that the user would be able to perform would be to create a new game, load the current game (this would be unavailable if the player has no game saved), and to quit the game, which would end the program. These options would be displayed in a similar format as the user selection screen.

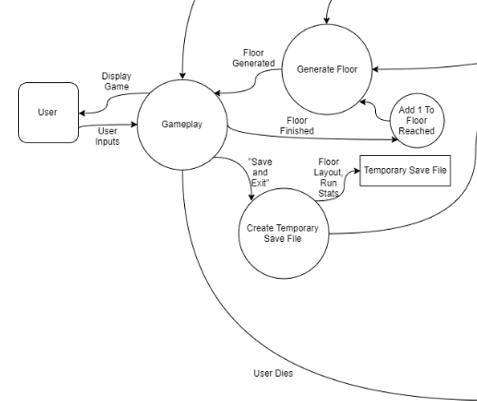
Should the user select to create a new game, the program would check if the user already has a saved game on their profile. If not, it will simply move on to begin the gameplay section of the game, but if the user does have one, the program will check with the user to see if they would like to delete the current saved game in order to start the new one. If the user agrees, the program will delete the current saved game from the user’s profile, and will begin to start a new one.

Should the user select to load a saved game, the program will access the user’s current saved progress for their current run. The information will then be stored in the program in order to render the user’s current floor, inventory, etc.

This will have been stored in a “temporary save file”, which will be deleted once the information is loaded into the program. This is due to the fact that the game is a roguelike, meaning that the user is not able to save and load a game indefinitely to correct mistakes or retry areas. The use of the temporary save files will mean that the user is only able to save the game as is when they exit to the main menu, preserving one of the main features of a roguelike game.

Features that could be included in the main menu are the ability to change options, such as the volume of the game, graphical options like brightness, and being able to set a specific theme to always use with each floor. As well as this, the user may be able to view their statistics, such as the furthest floor they have reached, the number of enemies they have defeated, etc. Also, they may be able to delete their current saved game without having to start a new one.

Main Gameplay Loop:



If the player selected on the main menu to start a new game, the program will first procedurally generate the floor for them to start on. Due to the ways in which the game scales as the user progresses, with the floor counter being the main contributor for the numbers generated, this floor will have the fewest rooms, the weakest enemies, and the most basic items. Once the floor has been generated, the user will be able to play the game. Both the generation process and the loop of gameplay are detailed separately.

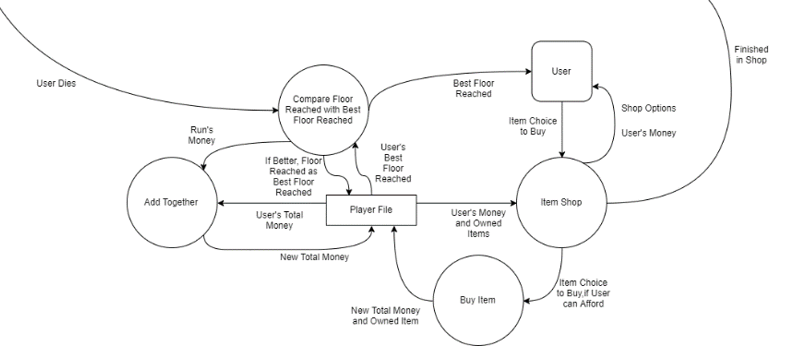
Once the floor has been completed, the floor counter will be increased, which will be how the game scales to be more difficult. After the counter is increased, the next floor will be generated for the user and this process will repeat indefinitely should the use never die or choose to save and exit.

If the user does decide to save and exit, a temporary save file will be created to save the current run’s statistics and the current floor layout for that user, so that they may pick up where they have left off later on. They will then be brought back to the main menu.

As detailed above, the save and exit feature is included in order to prevent the user from simply saving and loading whenever they please. However, this also means that should the program crash or the user closes the game externally through Task Manager or otherwise, they would lose all of their progress on that particular run. This could be remedied by occasionally performing an “auto-save”, perhaps at the start of every floor. This would mean that if the game closes unexpectedly, the user would only lose the progress they have made on that floor, retaining their items and currency that they had at the beginning. This would also include the floor map itself, meaning the user would not lose out on any items that may have been generated before the game ended. Ideally, these auto-saves would be frequent enough to prevent any incentive of abusing the feature, but not too frequent so to waste processing power.

As well as this, the save file, whether manually created or simply made through auto-saving, would be deleted as soon as the relevant information has been stored once the user dies.

User is Defeated:



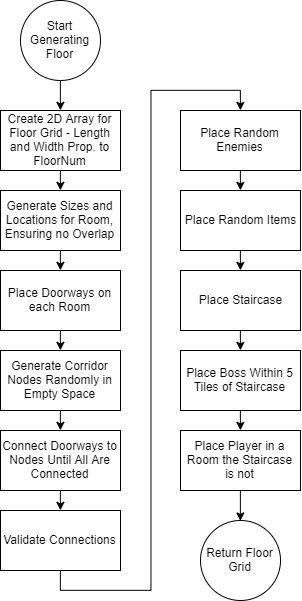
In the event that the user dies, the program will end gameplay immediately and will begin comparing and adding together statistics. This includes comparing the floor reached during this run to the user’s best floor reached ever and updating this in the user’s file, adding statistics like the number of enemies defeated to the user’s total enemies defeated, and updating their total currency.

This currency is then used in an item shop that can only be accessed after having died. Here the user will be able to purchase items and permanent upgrades. The items will be allowed to be generated on floors, and the permanent upgrades will increase the user’s base statistics and potentially allow them to start on later floors. These will both be stored in the user’s file, and this information will be checked and applied at the start of the user’s next run.

When the user is finished in the item shop, they will be brought back to the main menu, where they will be able to either start another run or quit the game.

The shop gives the user an incentive to continue to play the game, as they will be able to unlock stronger items and increase the base statistics of their character more and more, making it easier for them to progress further in the game, thus earning more money, allowing the cycle to repeat.

# **Floor Generation:**



In the process of floor generation, almost every aspect is derived from the current number of floors that the user has made their way through. This is in order to allow the game to scale in difficulty and complexity as the user progresses.

Features that will be based on this include:

* Length and Width of the 2D array used to generate the Grid.
* Number of Rooms.
* Size of Rooms.
* Number of “Corridor Nodes”.
* Number of Enemies.
* Strength of Enemies.
* Number of Items.
* Power of Items.

The connection of doorways and corridor nodes will be achieved using a pathfinding algorithm, like breadth-first or A\*, using them in order to find the closest node of the other variety when travelling through only walls, and connect them together with corridors along the shortest path between them. Once all nodes and doorways are connected to at least one other, a “cascade” will be used in order to validate that every non-wall tile can be reached. If this is not the case, the rooms that are not connected will have their doorways connect to the next closest node, and validation will repeat. This will prevent any rooms from being completely cut off and inaccessible to the user.

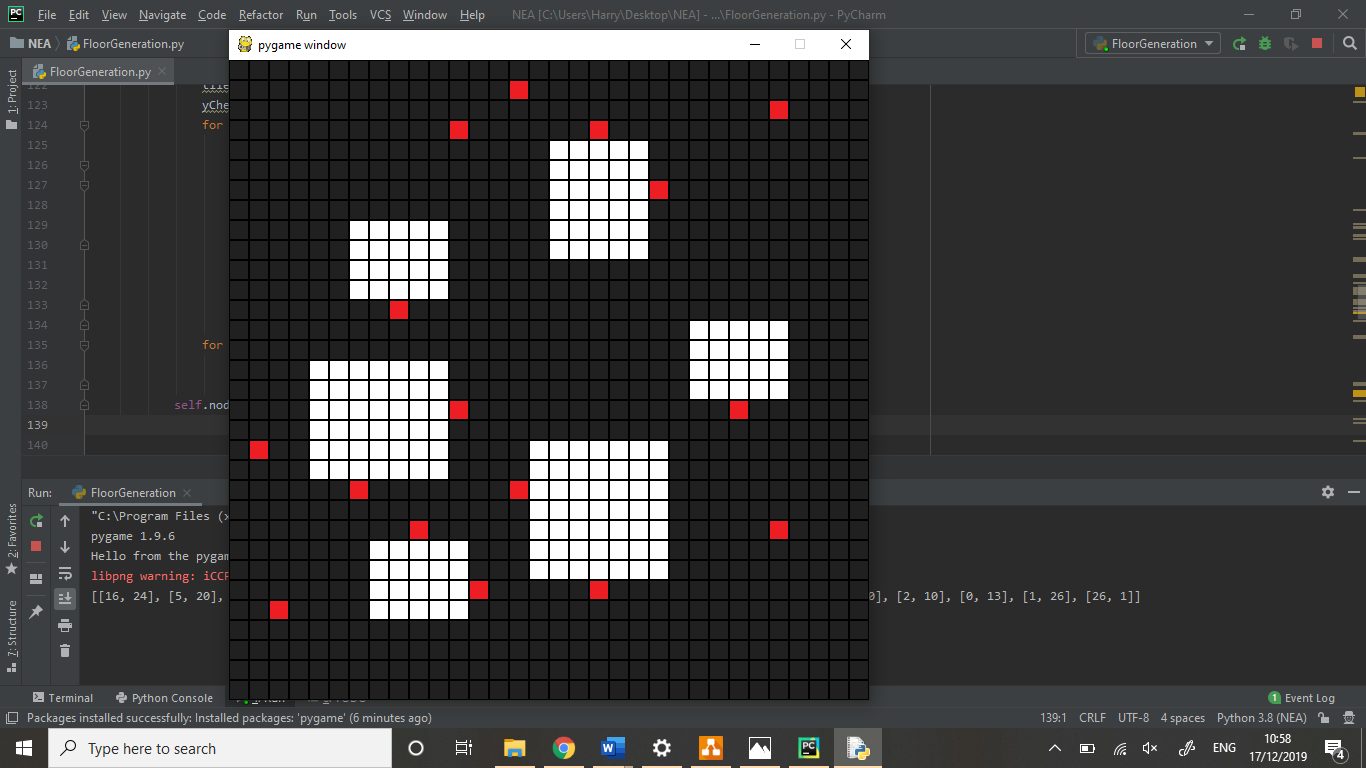
Once it has been validated that the floor is completely connected, the enemies and items will be placed. This will be done through an “enemies/items per room” quota with randomly generated positions. This means that there could be most of the enemies or items generated in a single room and none in another. This will potentially help to make the game feel less stale and repetitive, as there would be a different number of each in each room as opposed to there being the same amount in each and every one.

The ratio of enemies to items on each floor may change as the user progresses, so to increase the difficulty, since the items will become stronger as the game goes on.

The staircase is the way in which the user progresses to the next floor. This will be a specific tile that is generated within a room. The player progresses by standing on this tile after it becomes available. The staircase becomes available by the player defeating the “boss” of the floor. This is a strong enemy that has higher health and attack than usual enemies and will be chosen from a separate library as the standard enemies. The boss will spawn within a five-tile square of the staircase, so that the user does not have to search for the boss once they find the staircase, and so to make it obvious that the user must defeat the boss.

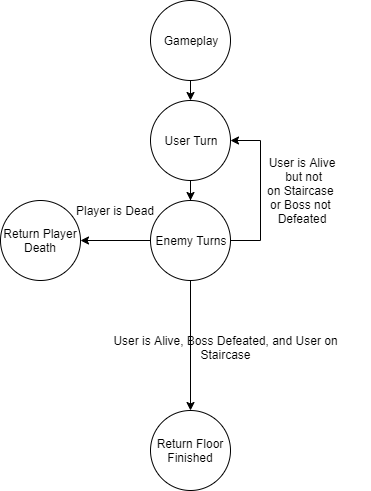
Once this has all been generated, the user will have their initial position generated. The position must be in a room that is not the room with the boss in. This helps to prevent the floor from just instantly being cleared, or the user from starting right next to the boss.

After this, the floor is displayed to the user and gameplay will be initiated.



Example of grid with nodes, without corridors, from an early build of the generation process.   
White Squares = Tiles that can be walked on.  
Black Squares = Tiles that cannot be walked on.  
Red Squares = Nodes

# **Gameplay Loop:**



The overall gameplay loop of the game is relatively simple, with the user taking their turn to perform an action, followed by the enemies performing their actions, and this cycle repeating until either the player is defeated, or the player has completed the condition for the floor to be finished. This provides a simple formula for how the game will play and will likely be achieved using a while loop with functions pertaining to the individual turns.

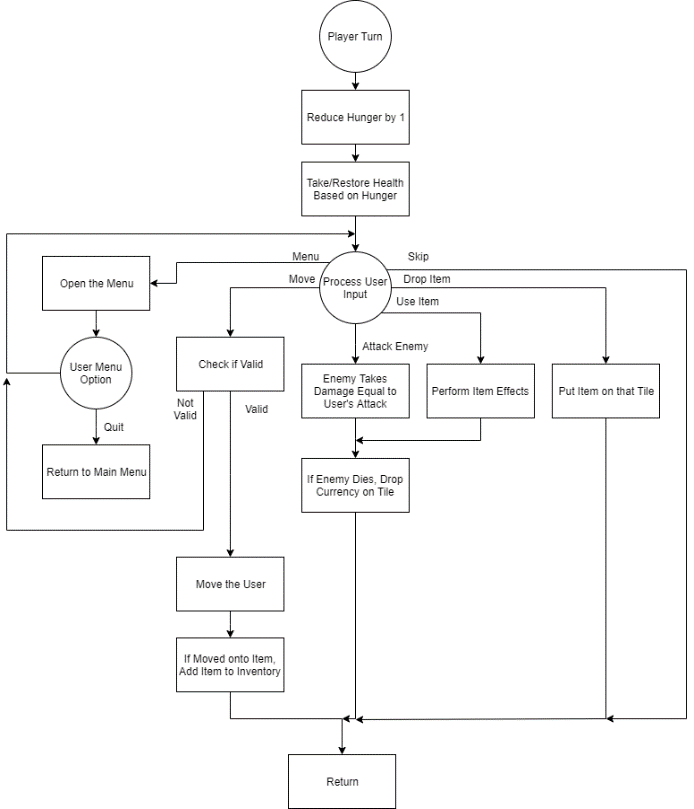
The conditions required for a floor to be considered are that the user is alive, the floor’s boss is defeated, and the user is on the floor’s staircase. In order to perform these checks, the program will verify a few factors:

* The user’s health is above 0.
* The boss enemy is not in the floor’s enemy list.
* The user’s position coordinates are equal to the coordinates of the staircase.

Once all of these factors are returned as true, the program will return that the user has finished the floor, will repeat the floor generation process, and return to the main gameplay loop. This will repeat until the user either decides to save and exit, or they die.

As soon as the user’s health drops to 0 or below, the program will consider them dead, and as soon as the enemies’ end their turns, the program will leave the main gameplay loop, and will move back to the main program flow, taking the user to the shops and statistics as previously stated.

User Turn:



Each turn, the user will have their hunger statistic reduced and have the effect caused by their current hunger act on them. This includes regenerating health if the hunger is above a certain point or reducing their health if the hunger is at 0. This gives a way for the user to recover health naturally in a way that still does have to be managed.

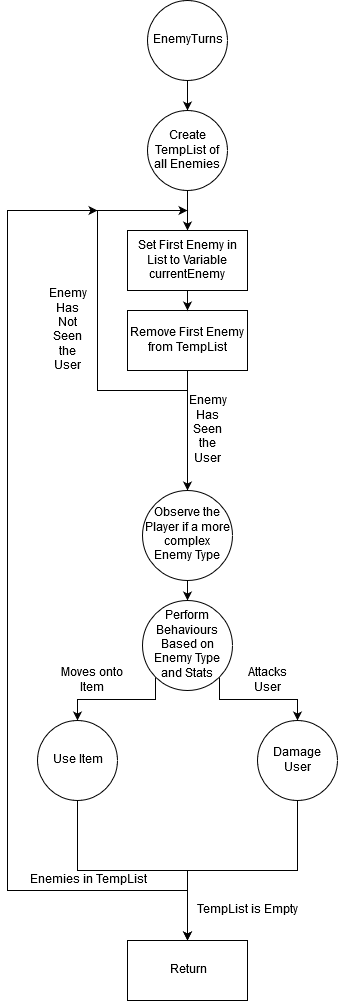
After the hunger effect has been taken into account, the user will be able to perform one of a few actions that will end the user’s turn. These include:

* Moving into an empty space.
  + In order to validate that the movement is valid, the program will check if the overall map grid there is taken up by a wall tile or if it has an enemy occupying it already. In the event that an enemy is taking up the space, the user will instead attack the enemy, remaining in place afterwards. If the user moves onto a tile containing an item, the item will be picked up, removing it from the floor’s item list and adding it to the inventory.
* Attacking an enemy.
  + When attacking an enemy, the enemy takes damage according to the user’s attack. This is reduced by the enemy’s defence statistic. If the enemy’s health drops below 0, they will be considered dead, and they will be removed from the floor’s enemy list. As well as this, currency based on the enemy’s currency value will be dropped. The multiplier to decide how much currency is dropped will scale with the number of floors passed by the user.
* Using an item.
  + If the user uses an item, that item’s effect based on its variety will activate. This can range from healing the user, restoring their hunger, “teleporting” them to a random tile on the floor, or performing an attack that affects the whole room. In the event that the item deals damage, the same process as when the player attacks an enemy is used to decide what happens if the enemy dies.
* Dropping an item.
  + Dropping an item simply checks if the tile beneath the user has an item on it already, and if it does not, the item is removed from the inventory and is placed into the floor’s item list with the user’s current position as its position attribute.
* Skipping their turn.
  + Skipping the turn simply performs no actions that turn. This will could be useful in that the user may be a number of tiles away from an enemy that means that should the user move towards the enemy, they would be hurt by the enemy, so instead they wait for the enemy to come to them.

After the user performs any of these actions, the program will return to the main gameplay loop.

The user is also able to open a menu, which will be where the user is able to save and quit to the main menu. This will cause the previously described “temporary save files” will come into effect. If the user closes the menu, their turn will not have been passed. The menu may also contain features such as settings and other typical features of a menu.

Enemy Turns:



In order for all enemies on the floor to be given the opportunity to act, all of the current enemies on the floor will be added to a temporary list. This will allow them all to perform an action via the use of having the first enemy in this temporary list to do so by setting it to a temporary variable and removing it from the list. This will repeat until the list is empty. This will ensure that every enemy currently alive on the floor will be able to act.

Once the enemy variable has been set, this enemy will begin to act. A check will be performed to see if the enemy has seen the user, either doing nothing that turn if it has not or performing an action if it has. This check will be done when the user enters a room, as all enemies in that room will be set to have “seen” the user, allowing them to perform actions. If the enemy has not been in the same room as the user for a few turns, it will be set to having not seen the player again.

When the enemy has seen the player, at the beginning of its turn, it will “observe” the user if it is a more complex enemy. This entails the enemy having unique statistics that change based on the actions that the enemy witnesses. This could include seeing if the user has defeated another of the same kind of enemy, which may cause the behaviour of the observing enemy to change from an offensive variety to one wherein the enemy keeps its distance and runs away if approached. This use of state machine AI would give the user more of a way to strategise upon encountering certain varieties of enemies.

The use of stateless AI could be introduced within the action phase of the enemies’ turn, wherein they could have a “morale” statistic that dictates how they interact with the user. For example, if a weak enemy sees a user with no armour and relatively low attack, the enemy may have a high morale and approach and attack the user. However, if this same enemy sees a user with a strong weapon, the morale of the enemy may drop below a threshold where the enemy would rather run away. This method of using stateless AI to dictate behaviours may also give the game some variety, as well as helping to avoid the user from defeating the same, weak enemies over and over again, making the game feel more fluid and giving the user a feeling of power.

If an enemy moves onto an item, whether it is approaching or fleeing the user, they will instantly activate it if it is an activatable item, such as a healing item or one-use item. This may also extend to enemies picking up items such as weapons in order to increase their attack, etc.

If the enemy is within range of the user, which may be dependent on the enemy’s range statistic, it will be able to attack the user. This would simply use the enemy’s attack and the user’s defence statistics in order to determine the health lost by the user. It may be the case that the user could select higher difficulties where the damage done by an enemy would be reduced less by the defence of the user, in order to make the game more challenging and less forgiving.

After the enemy has acted, the program will perform the check to see if the temporary enemies list is empty or not, either repeating the process with the next enemy or returning to the main gameplay loop.