Simple Fighting Game with Pygame

Final Project Report

Rafie Mustika Ramasna

2802522815

L1BC

Algorithm and Programming

BINUS University International

Jakarta

Table of Contents

Abstract

This is the final report that is serving as documentation and supplement for the final project of Algorithms and Programming that is scoping a students basic aptitude for the Python programming language.

This report is written for a simple fighting game that was created solely with pygame and basic python modules.

Project Specification

Introduction

Python as a programming language is extremely flexible and is able to encompass a broad plethora of problems in many different genres, topics, and fields.

For this final project of the Algorithm and Programming, in order to achieve the express goal of going further and beyond what is taught in the class itself, a choice of creating a fighting game with several game states and classes were made in order to fulfill these requirements. A fighting game can be used to help create a fun, social environment due to its inherently competitive nature and its involvement between several individuals when playing any particular game. Moreover the game systems in a fighting game, even a simple one like the one I have created here, are quite complicated and cluttered in pygame due to its inherently “low-level” engine which only comes with the most basic features.

Idea Inspiration

From the very moment that this project was given to me, I knew that I was going to make a game due to my extremely long history of being intrigued with the machinations to create one. Originally this game was meant to be more of a puzzle game with more narrative elements but that was quickly changed with this fighting game to give me more of a challenge in the programming department.

The reason I only chose pygame here as a sole module is mainly to test my creativity despite the limitations that it imposes. While using a more high end game engine would surely make a much better product, ultimately my choice of pygame was mainly to learn the more nitty gritty details which encompass the area of game development and to serve as the first stepping stone to further understand the ins and outs of game development.

As for why I chose to create a fighting game, I chose it due to its rather mechanically complex states when playing, being one where people need to truly focus on the screen to genuinely win. This feeling is something that I hope to achieve to anyone playing the game.

Solution Design

To create this fighting game, a proper framework for characters needed to be created so that players are able to fight one another in a loop which consists of several rounds as per the specifications of any other typical fighting game. For this particular game that I have created the main things that a player can do include: moving, attacking, parrying, jumping, and sprinting. Not to mention that players should be displayed on the screen and updated to show their positions at any time as well as their current state (i.e attacking, parrying, etc.)

At the very start pygame and its mixer needs to be initialized and the screen is given a WIN_WIDTH and WIN_HEIGHT which is configurable in the config module. The game is given a FPS cap of 60 and all fonts are loaded. At the same time the *Fighter* objects which are the objects that the players control are also loaded, this will be further explained in the *Essential Algorithms* section of this documentation.

Before entering the fighting game, players must navigate through a very simple main menu screen which allows some downtime for players to get ready and maybe read the documentation on github to learn controls before clicking play. This was created through blitting on the screen surface an image of a button and this button is a class in of itself which uses pygame to check if the mouse has hovered over the image and then does a function (which in this case is enter the game) once MOUSEDOWN is clicked on the image. This home

screen is kept in a *while* loop and will not terminate the program unless QUIT is pressed, waiting until the button is pressed before entering the game scene.

To begin with the game scene, the players are suspended on the air, the background is *blit* on to the screen, a health bar which is the ratio of the *fighter's* health / 100 is shown at the top of their side of the screen with their score below it, and a timer is counted down using *pygame.time.get_ticks()* minus a lastCountUpdate variable which is given a *pygame.time.get_ticks()* every second, lowering the intro count variable as well each second which is displayed on screen to display a round timer that goes down until 0 from 3. In this intro round sequence players have zero control of their character to once again prepare the players on the controls.

Once the time has reached 0 the players will have control of their characters due to the *move* function from both *fighter* objects, this function allows for the players to use either WAS or the ARROW KEYS to move their character and jump, R and T or . and / to attack, as well as Q or ‘ to parry/roll, with LSHIFT or RSHIFT acting as the sprint button that must be held. The control scheme of each player is dictated by the *player* argument set at the creation of the game object. The player's position will be updated and constantly drawn with *blit* each frame using the *update* function which flips through images from left to right on the spritesheet to play an animation of the action

being done currently. (Once again this will all be explained in much greater detail in *Essential Algorithms)*.

Each *fighter* is given the base health of 100 and each attack removes 10 health from the opposing *fighter,* checked with a *collidерect* function of the attack to ensure the attacks reach the enemy. However if this *collidерect* hits a *parry rect* which is formed in front of a player in a short window when a player parries, the attacker will be placed in *stun* where they cannot do anything until they are attacked in which after 500 milliseconds they can move again.

If at any point a *fighter* is whittled down to 0 health through attacks, the round will end, showing that the round has ended and the *opposing fighters* score is tallied by 1, after an internal timer of 3 seconds the round begins again and both *fighters* position is reset. Once any score reaches 3 then the game will display who won and the game will reset back after 3 seconds.

It should be noted that all of this is done in a *while* loop that can only be broken through pressing the quit game otherwise the game will keep going again and again.

Discussion

.Essential Algorithms

A. *Fighter* Class

The fighter class is the main object that players control, this Class is located in the module *objects.py (*more details in class diagram) and takes in the arguments of *x, y, flip, data, sprite_sheet, animation_steps, player, sound_data.*

➔ *x* and *y* are initial positions for *self.rect* of the object.

➔ *flip* is a boolean variable which when true will point to the left instead of right, used to make sure that they are facing the right place when suspended.

➔ *data* is a list that is created in the module *sprites.py* which consists of: *FIGHTER_SIZE, FIGHTER_SCALE, FIGHTER_OFFSET.* This list is done to make initialization take less arguments.

➔ *animation_steps* is a list of integers that contains the length of animation for each action in the sprite sheet.

➔ *sprite_sheet* is the png which has all the animations in the form of images that are laid out in a horizontal fashion, each different action is laid vertically.

➔ *sound_data* is a list consisting of all the *Sound* objects from the pygame module. Its purpose is similar to *data.* Audio is played through the variable that pertains to each sound through pygame.mixer.find_channel().play() so that it can be played with the background song as only one channel can only play one sound.

It consists of variables for which state they are in through boolean variables. These variables include *self.jumping, self.running, self.rolling, self.hit, self.alive, self.attacking.* There are also integer variables *self.action* and *self.frameIndex* to take the *self.image* from the grid images in the animationList (explained in the animationCreate function). As well as floats for timers in *self.stunTime, self.attackTime, etc.*

These booleans, integers, and floats are altered through many different functions of this class.

i. *animationCreate()* function

This function takes the *sprite_sheet* and *animation_steps* as arguments, it creates a list variable *animation*_list and then enumerates the variable *y* in a for loop for each *animation* in argument list *animation_steps.*

This loop creates a temporary imgList where another for loop is nested this time with the range as the *animation* (or the number of steps in that animation.)

This for loop takes a subsurface of the *sprite_sheet* or a portion of that sheet with x = (*x* * width), y = (*y* * height) and the width, height is taken from *self.size* which is part of the *data* that is initialized. This method uses a grid-like system where each space is the width and height and the coordinates are given from the enumerate for the y as well as the range for the x. This subsurface is stored as a variable *tempImg*.

This image is then scaled up and appended to the *tempImgList* where this list will be appended into the *animation_list*.

The end result will be an animation list which consists of many different lists in them, numerically indexed by their y coordinate in the grid. Each list consists of the image subsurfaces in order horizontally.

This list is returned and this function is called at the *init* as variable *self.animationList*.

ii. *move*() function

The *move* function primarily uses *pygame.key.get_pressed()* to see which keys are being pressed at the current moment so that the game can execute the command that it needs to when it is pressed. This pygame function is stored in the variable *key*.

It also has the function of adding gravity, ensuring the character is on screen, as well as making sure the *fighter* is flipped or not to face the

opposing *fighter* at any given moment. Methods of how each is done can be seen below:

➔ **Movement Checking**: The checks will start happening once the starting variables of *self.hit, self.stun, and self.attacking* are false to ensure that these do not stack on top of one another as this could lead to breaking the methodical approach to attacking one another and makes every impact and hit feel more punishing.

To check if a key is being pressed, the function uses the *if key[pygame.K_(KEY NAME)]* which will return True or False if that key is currently being pressed down. Which control scheme they check with is done with an *if* function to check if their *player* argument is 1 or 2.

Movement is done by adding dx += (RIGHT) or -= (LEFT) *SPEED (or SPRINT* if SHIFT is held) which is set in the *config* module.

Other than that the *move* function deals with basic gravity and ensuring characters are on the screen at all times.

Cooldowns of attacks and rolls are also managed here by comparing if the current time - their stunTime or attackTime which is given in their specific functions are larger than a integer which measures milliseconds before allowing them to attack by making

their action boolean back to *False* to allow them to do the action again.

➔ **Gravity:** Gravity is done using *self.vel_y += GRAVITY* where GRAVITY is set in the *config* module and *dy += self.vel_y*. *dx* and *dy* are used instead of directly moving the rect to allow it to remain in the screen as well as make the movement at a constant speed and only having to deal things in a scalar manner instead of in a vector manner.

➔ **Ensure Fighter on Screen:** To ensure the rect is in the screen it checks if adding dx will make its rect.left be smaller than 0 (go off the left side of the screen) or if the rect.right is larger than the *WIN_WIDTH* (goes off the right side of the screen.) Similarly this function also checks if adding *dy* to the rect.bottom will make it be larger than *WIN_HEIGHT - 126* (Which is calculated to be the height of the ground.) to ensure it does not go off the bottom.

If any of these are true then the *dx* or *dy* will automatically be made to be 0 - its rect.left/right/bottom so that the sprite remains on screen.

➔ **Checking to Face Target:** To ensure that the *fighter* is always facing the target this function takes argument *target* which is the

class of the opposing *fighter* and checks if the target's rect.centerx is larger than the *self.rect.centerx* (self is to the right of the enemy) and turns *self.flip* to *True* if that is the case, this *self.flip* will be used in other functions to show that the character is flipped.

iii. *update()* function

The main purpose of the *update* function is to change the image of the *fighter* depending on the action that is being done. This consists of many if's particular for the booleans to see if they were *True.* This includes if *self.health <= 0, self.hit, self.stun, self.attacking, etc.*

If they are true then it will call the *updateAction* function which takes an argument of *action* and sets the *self.action* to this *action* number and starts the *self.frameIndex* from 0. Essentially turning the current image to be the leftmost image and the *actionth* row from the top of the sprite sheet, which row corresponds to which action can be seen through the *spritesheetguide.txt* in the sprites/players folder. This function also takes the time and saves it to *self.updateTime.*

The current time is subtracted with updateTime is compared with the animationCooldown which is 50 before it adds the frame index by 1 to move to the next image in the list of that *actionth* row from the top of the spritesheet and takes the updateTime again.

iv. *roll()* function

Turns *self.rolling* to *True* so they cannot spam rolling and creates a *Rect* object that is 1.5x of the width of the *fighter* and is either placed to the left of the center with distance of its width spanning the left side of the fighter or right at the center spanning to the right of the fighter depending on if it's flipped through the formula self.rect.centerx - (2 * self.rect.width * self.flip).

v. *attack()* function

Turns *self.attacking* to *True* so attacks cannot be spammed until the cooldown and creates a *Rect* object named *attacking_rect* that is 2x of the width of the *fighter* with the same flip method of roll. This function takes *target*, which is the opposing *fighter* object, as an argument.

Checks with an *if* on whether this rect collides with the *target*'s *rect*, when True then the *target's health* will decrease by 10 and force the target to enter the *hit* state where they cannot move for 50 milliseconds. Counts the *targets* stunTime and will be counted when they end up being stunned.

Right after it uses an *if* statement to check whether the rect also hits the *target's parry Rect*, if so then it will turn *self.stun* to be True and start

the stunTime with *pygame.time.get_ticks()*. Not allowing movement until they get attacked from the *target*.

B. *Game* Class

The Game Class is the class that acts as the Scene Manager and is placed in the *main* module without any arguments. The main functionality of this class is to manage between the game and the main menu, for keeping track of a currently running game through its rounds and also for initializing pygame itself.

i. *mainMenu()* function

This function simply fills the screen with a background color and then draws text of the game name via the *drawText()* function which simply render and blits the font. There are also buttons through the button class that are blitted on the image as well.

These buttons have a function called *clickCheck()* and if this is true then it will change the *self.state* from “menu” which is what it is initialised to “local” which is used at the bottom of the module as there is an if statement that if game.state is equal to “local” then it will run the localBattle function which is where the game actually begins.

ii. *localBattle()* function

This function will be where the fighters are updated and drawn several amounts of times equal to the FPS. To check if 3 seconds have passed before battle there is an if the variable *self.introCount* is less than or equal to 0 otherwise it will count down after *pygame.time.get_ticks()* - *self.lastCountUpdate* (which is a get ticks initialised at run time) is equal or larger than 1000 milliseconds before decrementing and updating the *self.lastCountUpdate* to be the current time.

Once *self.introCount* has reached 0 or less than 0 the fighters will have access to their *move()* function and can attack each other until the health bar which is drawn has reached full red or 0.

To check scores and win condition it will check first using an if statement to see whether the variable *self.score* which is a list at index 0 is equal to 3, likewise it checks using an elif statement to see whether index 1 is equal to 3. Should any of these be true it will display that either PLAYER ONE or PLAYER TWO has won based on index with index 0 being player one and index 1 being player 2 then it will reset by creating overriding the fighter_1 and fighter_2 object initial parameters to default and creating these objects again.

In another statement it checks if the variable *self.roundOver* is False and only turning this variable to True should any variable of fighter_1 and fighter_2 reach 0, adding to the score of the one still alive then counting a

self.roundOverTime which is equal to the current moment *pygame.time.get_ticks()*. Once *self.roundOver* is true it will display that the round has ended and count down by 3 seconds before resetting by subtracting the current *pygame.time.get_ticks()* and *self.roundOverTime* and making sure its larger than the variable *self.roundOverCD* which is initialized at the same class.

Diagrams

A. Activity Diagram

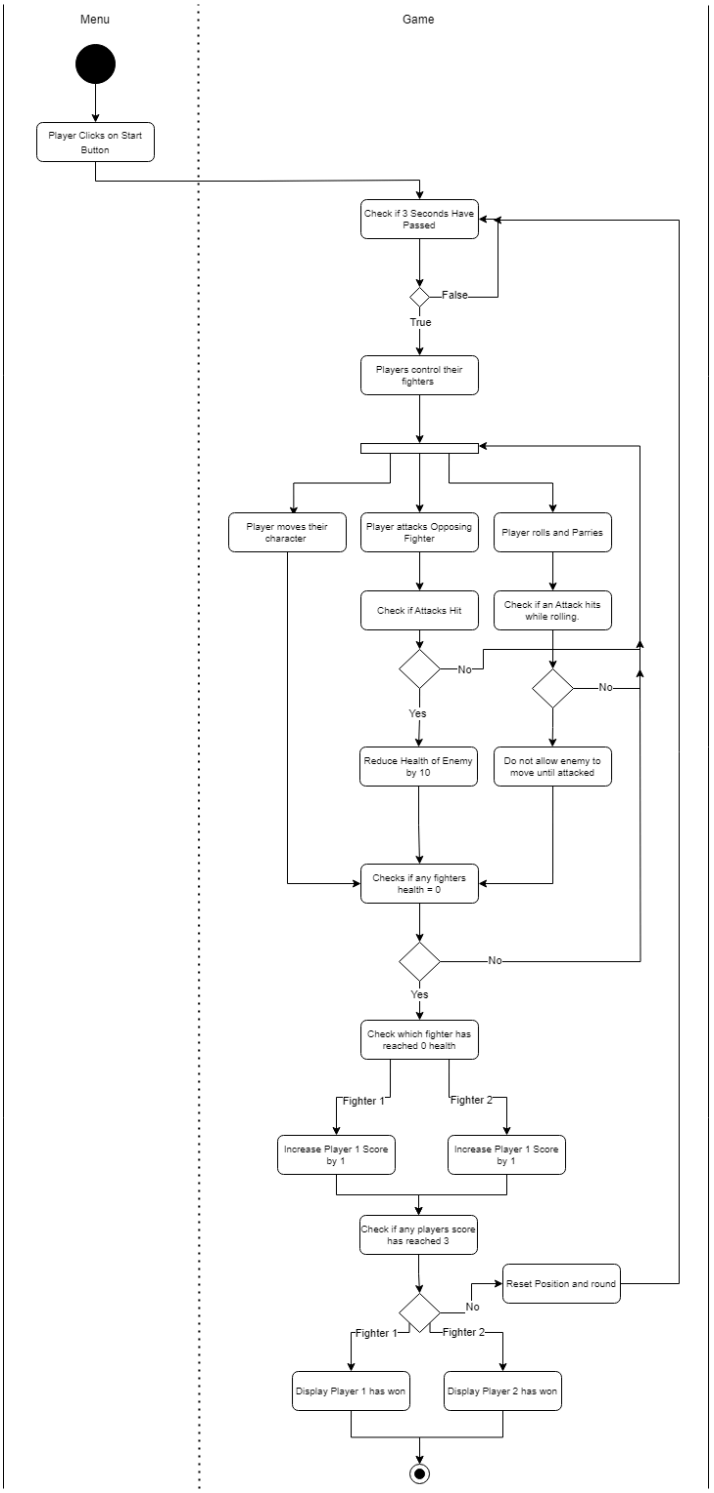

Figure 1. Activity Diagram

B. Class Diagram

Diagram below does not contain the pygame module as that module is not directly made by me.

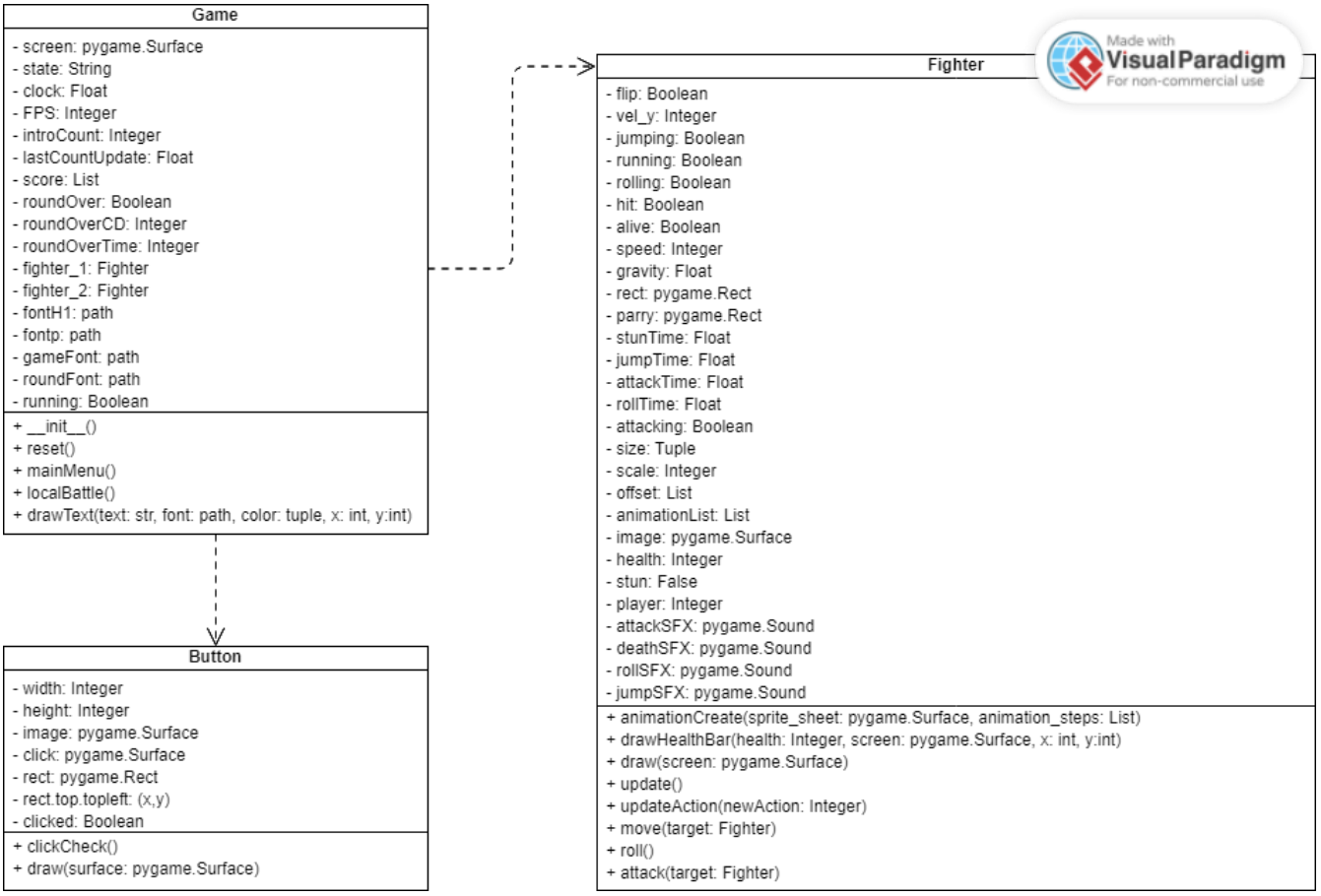

Figure 2. Class Diagram

Modules

1. pygame: pygame is a low level gaming programming engine that runs on python and handles the entirety of taking in images as sprites to display to the screen, player management and control, audio playing through the pygame mixer, as well as font displaying and time and FPS management. This entire project is entirely done by this module. The UI, the game itself, and the main menu are done using only pygame.

Program Functionality

A. Github Repository

The repository can be found and accessed through entering the following link: https://github.com/rafipy/Unto_Death

B. Screenshots of Working Program

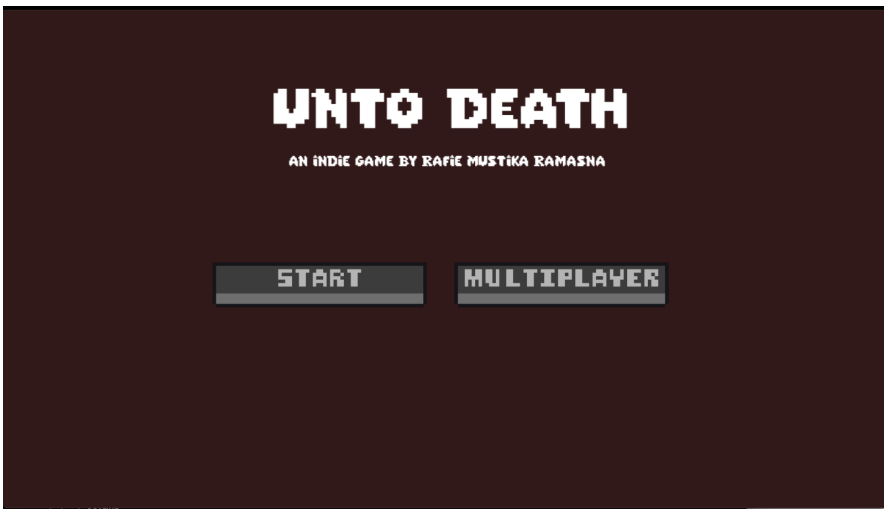

Figure 3. Main Menu

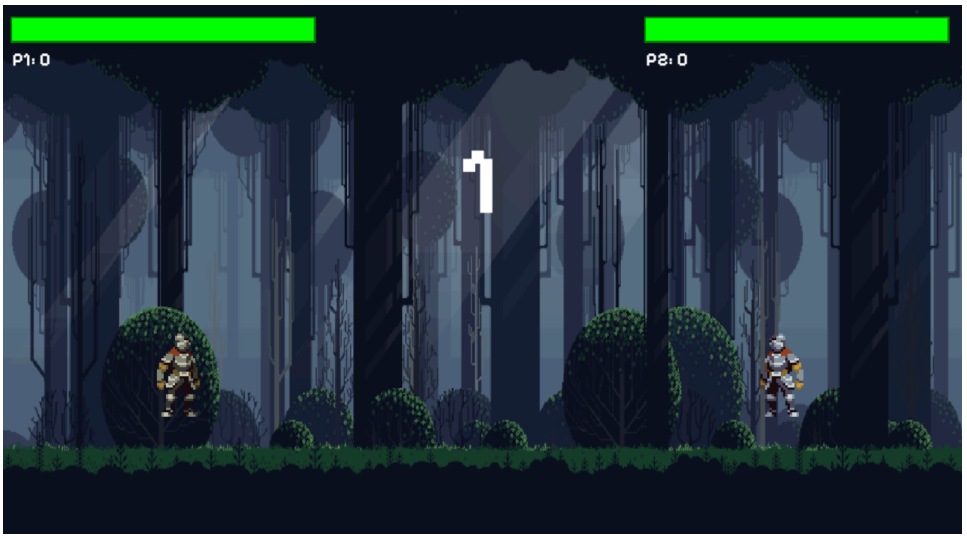

Figure 4. Pregame Countdown

Figure 5. Round in progress, Players can move, attack, and roll

Figure 6. Round ends when player dies

Figure 7. Game ends when a player wins 3 rounds.

Closing

A. Lessons Learned

Through this project I stepped into the world of game development and design as it challenged me to come face to face with exactly how animations are made and run from a sprite sheet to the screen, I learned how to create rigid bodies from scratch, the functions required to create a functional player character, and also the thought process and pipeline starting from nothing until the finished product of a fighting game.

I’ve learnt about many things too about python in general and its versatility in doing many different tasks. In a sense, I’ve also learned the logic of games as I’ve created the class and activity diagrams.

B. Future Improvements

The process of making a game is at the end, a never ending process built upon design and testing again and again. The current build of the game fails in an idea I tried to attempt and that is LAN play using sockets, this is something that I could work on in the future with a more improved version of the game. Other than that a requirement to add a lever block system and better physics could have been implemented to give a much more whole experience and allow for players to have a more whole experience.

References

Coding With Russ (2022) *Street Fighter Style Fighting Game in Python using Pygame* https://www.youtube.com/watch?app=desktop&v=s5bd9KMSSW4

aamatniekss (2021) *Fantasy Knight - Free Pixelart Animated Character* https://aamatniekss.itch.io/fantasy-knight-free-pixelart-animated-character

edermunizz (2017) *Free Pixel Art Forest* https://edermunizz.itch.io/free-pixel-art-forest