1. PROJECT SPECIFICATION

For this final project I decided to make a simple game that requires 2 human players. It's a game where each players controls their own characters on screen with the goal of eliminating the other player by inflicting enough damage by having its opponent's health reaches "0". Simply put, it's a game where two players are stuck on screen, forced to survived until one of them has been defeated. To defeat the opponent, players' characters on screen are given weapon which are all ranged weapon, there are no melee attacks here. Weapon types are differentiates by the behavior of its projectiles/bullets. In that case, to win the game, each player has to fire bullets and make sure it hits the opponent. For the characters, the game uses png images of people holding bows or guns or something similar.

At the beginning before the game starts, both player 1 and player 2 are given lists of available characters and weapon types to choose. Player 1 got to pick first. Each player may not choose the same option, which means both players does not have the same attributes since each weapon types or characters has unique value for some things such as "max health" or "movement speed". When the game starts, player 1 will spawn on the left side of the screen facing right, while player 2 will be on the opposite side of the screen facing left.

Different weapon types has different ways of eliminating the target, for instance some of them has different reload time (which means you may not fire again and have to wait depending on the weapon type) while others may fire 3 bullets of different directions as soon as it fire. As for the gameplay, players may jump as many times as possible throughout the map, even players may move its sprites while airborne and fire at the same time. However, players may not get out of the map, player may also pushes the other player when colliding, a feature I implemented to make things more interesting. For instance, if you are airborne and there is another player right below you, you would land on top of its head instead of landing on the ground. Similarly, if you move right while there is another player right next to you, you would push him further to the right as long as he's not pressing any key to move.

For player 1, the controls are on the left side of the keyboard, which is key "a" to move left, key "d" to move right, key "w" to jump, "r" to fire, and "t" to switch direction. Note that key "s" for going down isn't needed since players falls down automatically due to gravity. As for player 2, naturally it's on the opposite side of the keyboard, it uses the arrow keys to move left, right, as well as for jumping, while also uses "." For switching directions and "/" for firing. Different characters may have different values for its movement speed and how high it jumps, and also how fast it falls while on air differentiated by its attributes. More importantly, different characters also have its own png images, which means they all look different. The game may also be paused by pressing the key "esc".

As I have stated in the above paragraph, players can change the direction where they are facing, either “right” or “left” by pressing its respective key. Where the sprite is facing affects the direction of the bullets when firing. For instance, when the game starts, player 1 would spawn on the left facing right. Since the sprite is facing right, the bullets being fired would also be directed to the right. But what happened if your opponent somehow ended up behind you, to the left that is? You would need to turn around, which is why changing directions is really useful.

In this game there are no levels. On the top left you would see the health bar of player 1 along with its numerical value below it, while on the top right you would see the same information but for player 2. Once the bar is empty, a winner has been decided. When the game ends, player may continue playing again (a rematch) or quit the game. On a different note, I’ve also implemented a walking animations for the sprites/characters, though not all are supported.

2. SOLUTION DESIGN

How I made a game like I described above is to create classes for each weapon types and sprites to make every single one of them unique. This allows them to have different values such as bullets travel speed or the size of the sprite itself, including values that I have mentioned before such as “movement speed” or “max health” or “reload time”. All of these information are stored in csv files. “char_database.csv” is a csv file stored in the same folder as where the main program is located, and contains list of available characters along with its attributes. I use the same concept for the weapon types, it uses “weap_database.csv” to store list of weapons and its attributes that could be picked by player 1 and player 2 before the game is initialized. The csv file is designed in such a way to match the positional arguments for the custom classes when being read, thus enabling to create new objects just by reading the file. Basically, each line contains one object along with its attributes, ordered based on positional arguments. The program will read it as dictionary and save it into a local variable in a form of list containing all of those information, storing each objects inside it. This is done by using ‘for loop’. This is mainly how characters and weapons are different from each other. Two classes are being used, one is for characters and the other is for weapon types. These two classes are mostly unrelated but each character has an attribute called “weapon” which takes one object from weapon class. At the beginning its value would be “None”, since the weapon could be anything depending on what the player chooses at the beginning.

To make the game operational the use pygame module is necessary since it contains many functions beneficial for creating games, which allows the program to trigger a certain behavior based on what keys you press for example. The program uses pygame module to draw texts or rectangles or circles or even buttons to click that may appear on screen. For example, clicking pause will cause the program to display various buttons in the form of text which could be clicked with mouse. The health bar that are being displayed for each characters are also rectangles drawn locally inside the program,

including the numerical value below it. Even the bullets are just drawn small circles colored black. Almost everything is drawn through the function 'draw' in pygame, with the exception of the background, sprites, and tutorial menu, which are png files stored outside the program.

The movements of sprites and its bullets are recorded with its coordinate. In fact, the collision mechanic is possible by checking where the sprite is located based on coordinates, further will be explained below.

3. HOW IT WORKS

Each character has attributes called "height" and "width", which means the characters that appears on screen may not be the same size. They also have attributes called coordinate, which is used to track where are they located throughout the map. These attributes are vital in the implementation of collision mechanic, and also its movement (including jumping and falling down). Its movement is done by changing the coordinate either by adding it with a value or subtracting it when the key is pressed, and the value that are being used is its own attribute called "movement_speed". This attribute only applies for going left and right, each character has another attribute called "jumping", which values determined how high the character will jump every time the key is pressed. They also have attribute "falling" which is how fast the character will fall while on air. In conclusion, the movement speed of each character is dependent on the value in its attributes, allowing each of them to be truly unique. These values are stored in csv files as well.

Now let's talk about the collision mechanic. As I've mentioned before, player may pushes the other when colliding. When going to the edge of the screen, player also can't get out of the map, as if they are bumping into something. This is done also by using its "height", "width", and "coordinate" attribute, and the movement method in its class. Each movement function has 4 arguments which value is the limit of how far your character could go based on x-axis (left and right) and y-axis (up and down). Normally, the default value would be "0" and the value of screen height and screen width, which are the coordinate of the edge of the screen so that players can't get out of screen. However, if conditions are met, the limit could be changed and becoming the other player's coordinate to make it so you may bump into them.

Since the coordinates for each sprite is located on the top left of its image, the value of its width and height are necessary to make the right adjustment in order to make collision happen correctly. For instance, the program will check that if player 1 is going right and its x-coordinate and players 2's x-coordinate is the same (y-coordinate is also being checked), then the collision would happen and may pushes the other player, except it won't. To make it work, it has to be player 1's x-coordinate plus its width and players2's x-coordinate that's the same since the coordinates always starts on the top left of itself for some reason. The same logic is also applied when falling or jumping, since you may bump into the other player if it's located right above. In conclusion, the far left of the

sprite is its x-coordinate, the far right is x-coordinate plus its width, the top of its head is y-coordinate, and the bottom of its feet is y-coordinate plus its height. With these data provided, we could make this sprite a 'rectangle' based on those values, thus allowing the sprite to act as if it's an actual object that can be touched.

The value of the attribute width and height are in pixels based on the actual size of the image, which are also stored in csv files. Given that the height and width might be different for different sprites, that means some characters might be easier to fire at or push. Damage are inflicted only if the opponents' bullet touches your 'rectangle' (somewhere between its x-coordinate to x-coordinate plus width, and between y-coordinate to y-coordinate plus height). When that happens, your health is deducted based on your opponent's weapon type's damage, which is an attribute for weapon class, also stored in csv file.

How does bullet work? When a player presses a key to fire, a bullet will appear and start traveling based on where the player is facing. If it's facing right, the bullet will go right from where the sprite is standing. How fast the bullets travels are based on its weapon attribute "x_speed", which means some weapon types might have bullets that travel slower or faster. When firing, what appears on screen are actually drawn circle colored black that emerged on the sprites' coordinate that's holding the weapon. Much like the sprite, since the only way to make things move on screen is by changing its coordinates (either by adding or deducting across the x-axis or y-axis), keeping track of bullets that are already being fired are necessary.

The coordinates of those bullets that had already been fired are then stored in its own weapon attribute called "fired_bullets_list", which is originally an empty list. Those coordinates on that list are constantly being added or deducted with the value of "x_speed" (depending on which direction, left or right) in order for it to keep moving across the screen. Every time you fire a bullet those new bullets' coordinates are stored on "fired_bullets_list" too, which means this list includes all bullets' coordinates that has been shot. The weapon class has a method called "update_bullet_list" which updates the content and value of attribute "fired_bullets_list". Much like the movement method for sprite, once the coordinate hit a certain limit, it will be removed from the list. It's removed only if the coordinates reached the outside of the edge of the screen, and also when it hits the opponent. This is necessary to keep the list small and organized.

The method "update_bullet_list" has an argument which is the enemy sprite (in a form of object). When it hits the opponent, the opponents' health would be called and reduced by the weapon's attribute called "damage".

Since the directions of where the bullets are going to travel are based on where the sprite's facing, some adjustments need to be made so that bullets won't change directions mid-air. For instance, when the player fire to the right but then suddenly turns around, the bullet should keep going right.

The weapon class also has a subclass called “threeway”. The only differences from its parent is that every time it fires, it fires 3 bullets with slightly different directions, one is normal (horizontal), the others are horizontal too while slightly vertical, up and down. This subclass utilizes an attribute called “y_speed” which determines how high or low the vertical bullets will travel. If the value is “0”, then it will fire normally (horizontally).

Another subclass is called “Doublefire”, which as the name suggests, fires 2 bullets consecutively every time the player hit the key to fire. That’s the only differences with its parent class. Another one is called “Quadruplefire”, which fires immense amount of bullets every time the player hits the key to fire, and that’s also the only difference of this subclass with its parent class.

When the game is initialized, player 1 and player 2 are given choices of what characters or weapon types to pick. Those that have been picked are popped from the list (removed), which means each player will never pick the same available weapon type or available character. This is done through command prompt, where the player has to input which ones to pick. After each player is done picking them, the pygame window will finally starts, displaying the basic tutorials of what key to press and what to do. This is done by displaying an image tutorial while pausing the game. There will be a button below it called “ok”, clicking it will start the game for real. When the game is paused, almost everything is halted, by using a Boolean variable called paused that returns true when the key for pause is pressed, and when it’s true, most of the code will be halted since most has if statement that only works if not paused. When paused, there would be multiple buttons appearing on screen, which are resume, restart, instructions, and exit. Clicking resume would unpause the game, clicking restart will restart the battle (reset everything, the starting position of each sprites including returning its health to max), clicking instructions will displays the same basic tutorials that was presented at the beginning, and clicking exit will close the program.

When a winner has been decided, the game will be halted and displays the text “P1 WINS!” (if player 1 wins), or else “P2 WINS!”. When that happens, a text at the top of the screen will appear displaying “try again?”, along with clickable buttons “yes” and “no”. Clicking yes results in the same outcome as clicking restart when pausing, which is basically a rematch, while clicking “no” will close the game for good.

How about the walking animation? How does that work? It works by switching the images for sprites constantly with their legs on different positions, one is closed and the other is wide open. By keep switching these two images while moving, it looks like the sprite’s leg is moving on screen. When a key for opposite direction is pressed, a reversed image of the original would be used instead, including its movement images. If the key to change direction is pressed again, then the image would be back to its original one. This is also done by using Boolean variable called “reversed1” for player 1 and “reversed2” for player 2. If those values are true, then the reversed images would be used instead, which

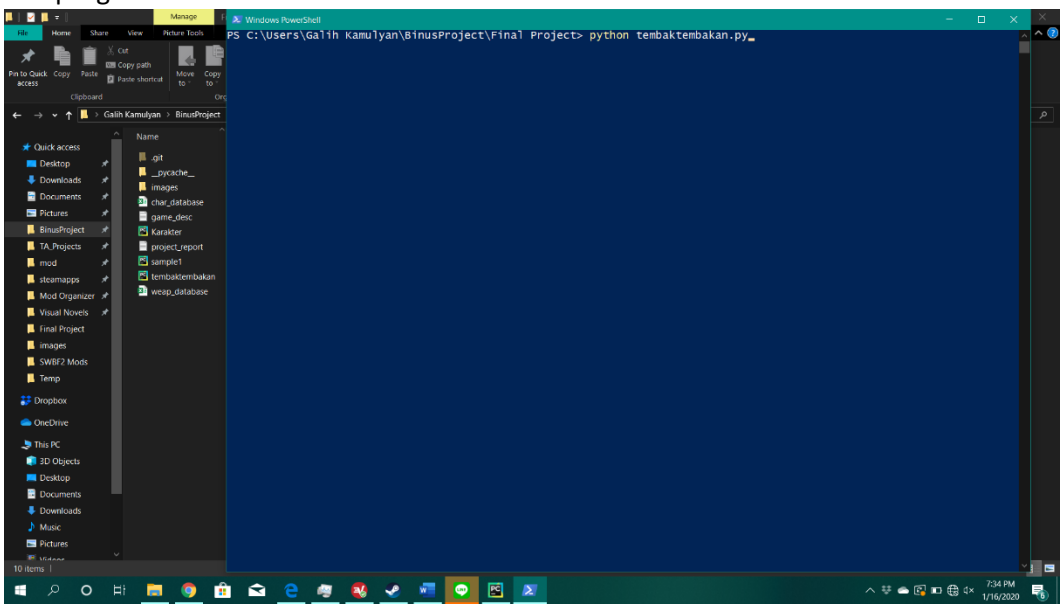
happens when the player press the button to change the direction where its facing. Pressing the button again will make the Boolean value False since it returns to its original direction.

There is also a Boolean variable called “moving1” (for player 1) and “moving2” (for player 2), which is always false, unless the player presses the key to move right or left. It only becomes true when the key is being held. When the value becomes true, the program will keep switching images of the sprite to make it looks like it’s walking. If the value is not True, then the image would be static since the sprite is just standing there unmoving. If the direction is changed, then it will use the reversed version for both standing and its walking images. Sprites’ standing and moving images and its reversed are both stored in the main folder and loaded with its path provided in the csv file and will be used if those conditions above are met.

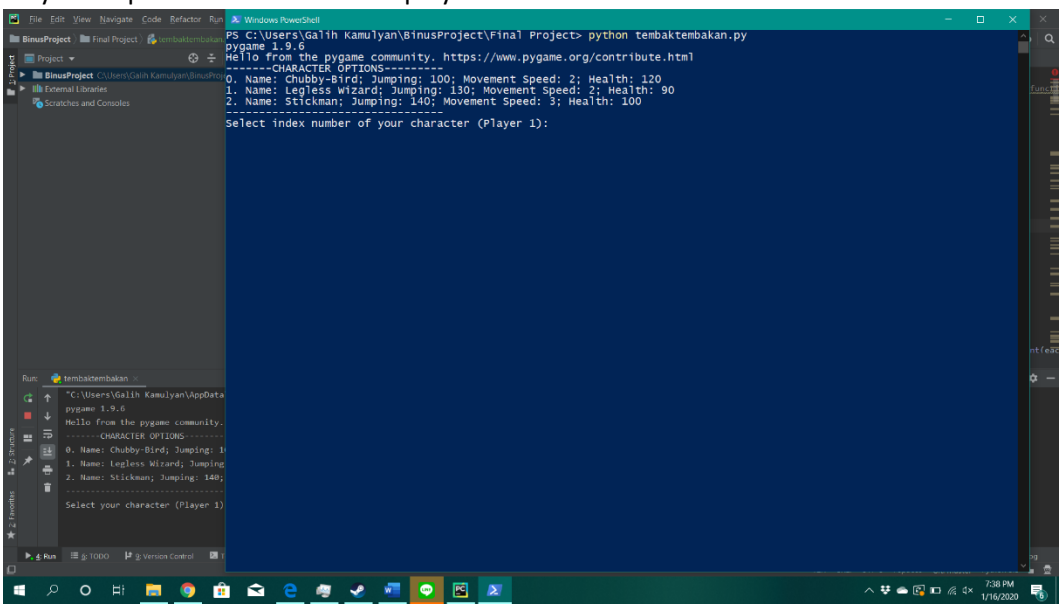
How does health bar works? It works by calling method “get_health” which exists inside the sprite class, constantly. It also calls the attribute max_health which also exists for every sprite. With both data provided, it’s possible to display numerical values below it as well which is how much health it has left in comparison to its maximum health.

4. EVIDENCE OF WORKING PROGRAM

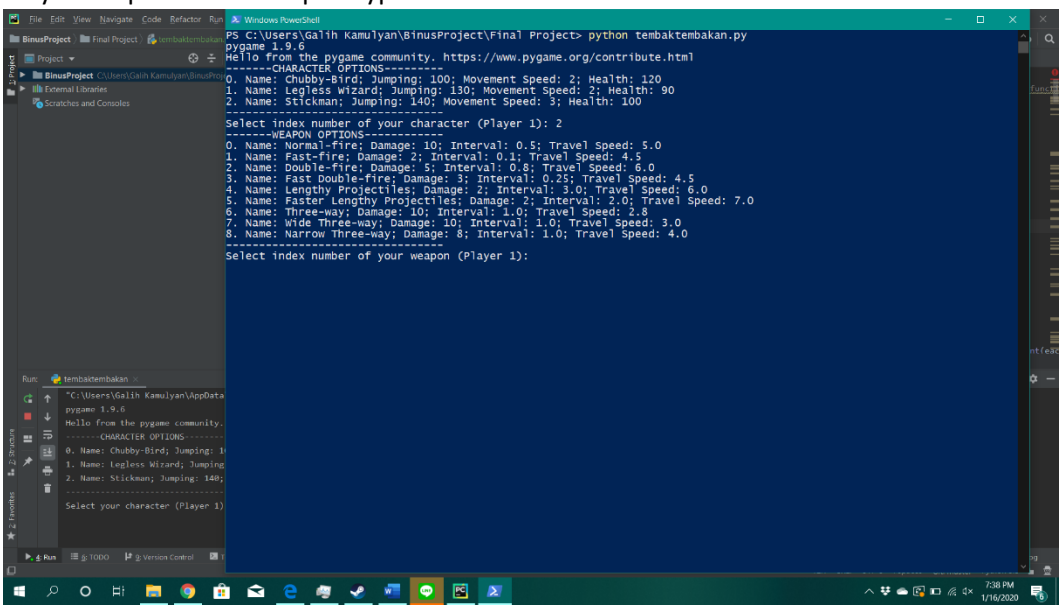
a. The program is executed



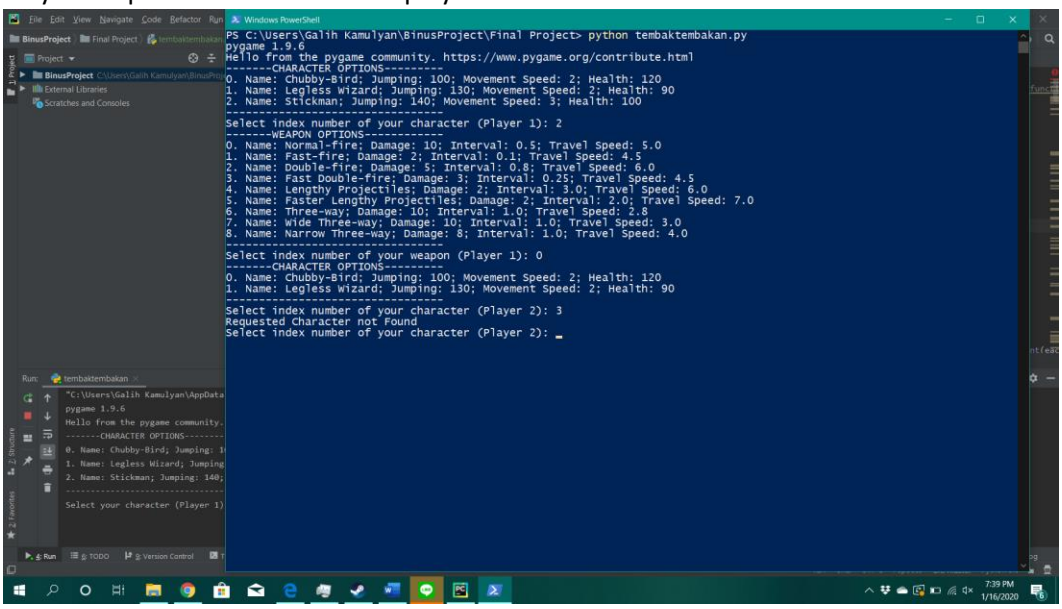
b. Player 1 input which character to play as



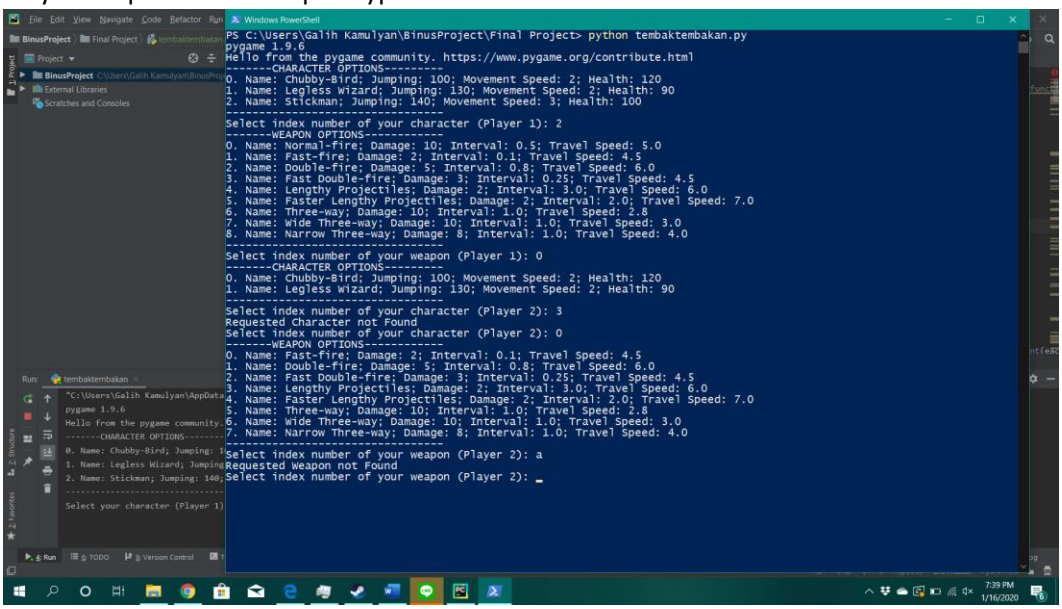
c. Player 1 input which weapon type to use



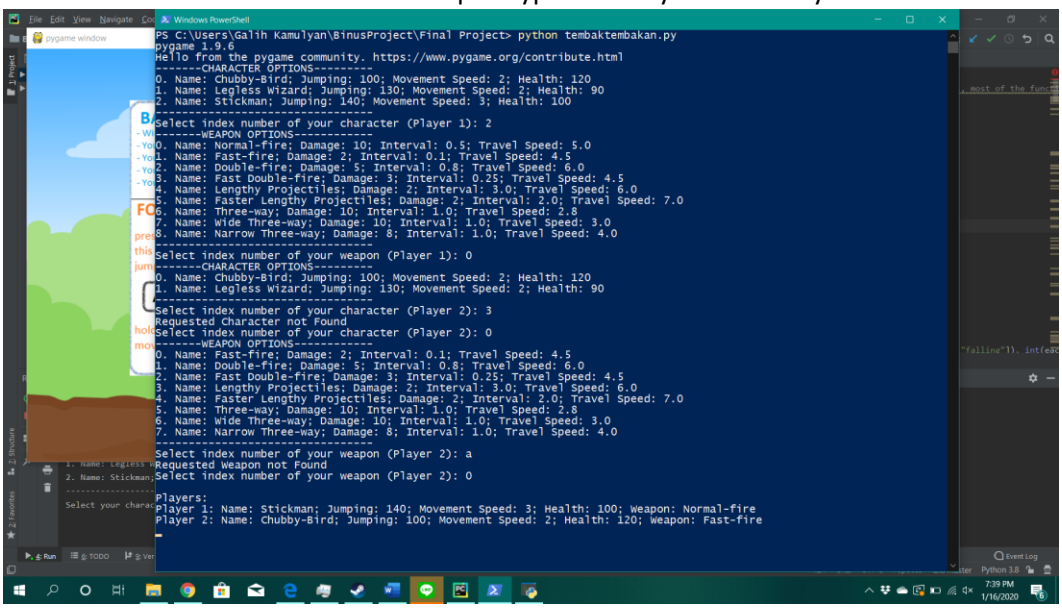
d. Player 2 input which character to play as



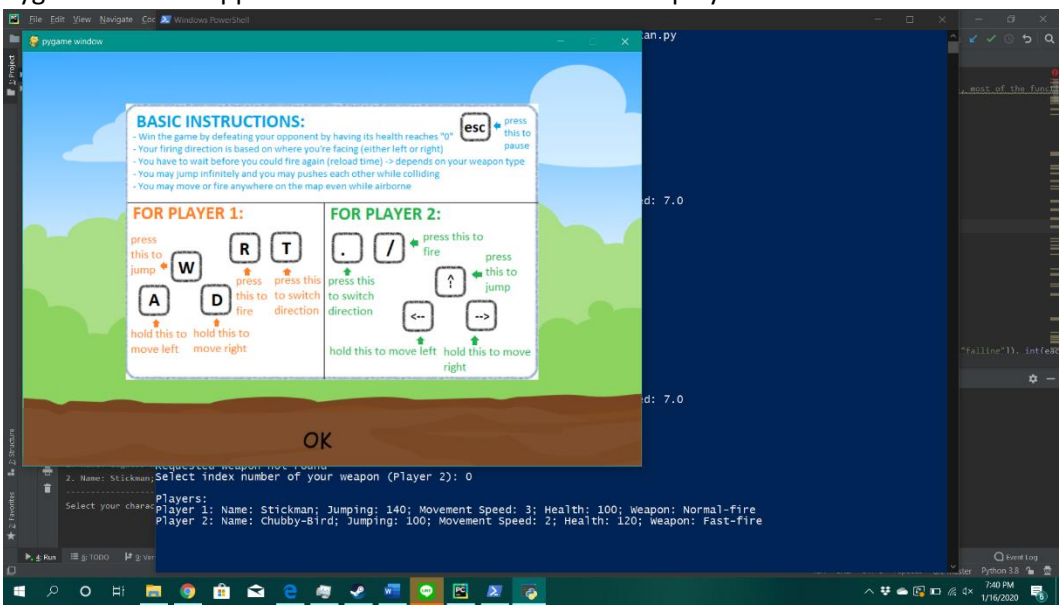
e. Player 2 input which weapon type to use



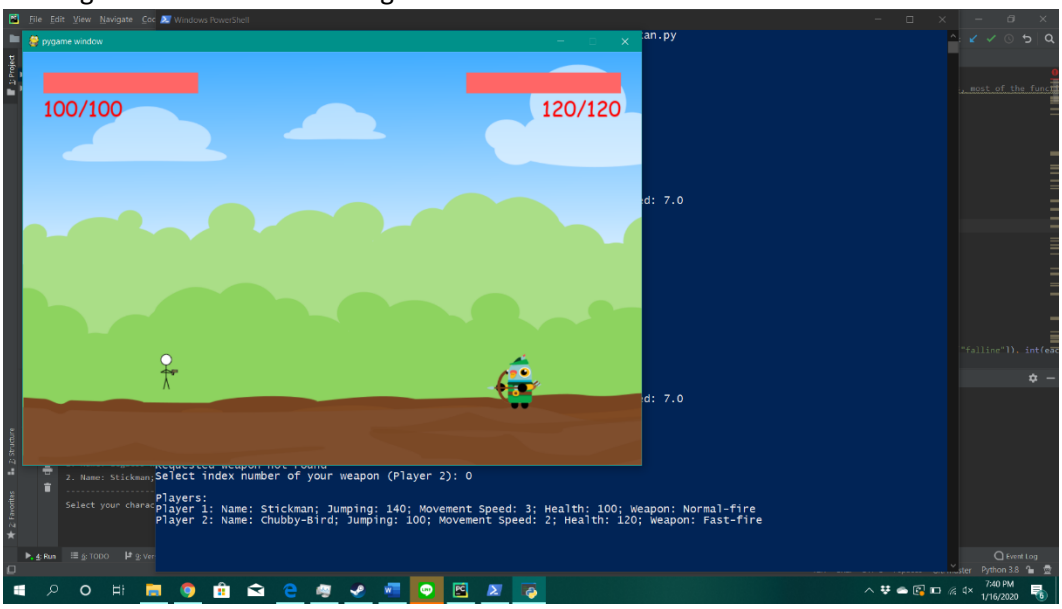
f. Shows the chosen characters and weapon types for Player 1 and Player 2



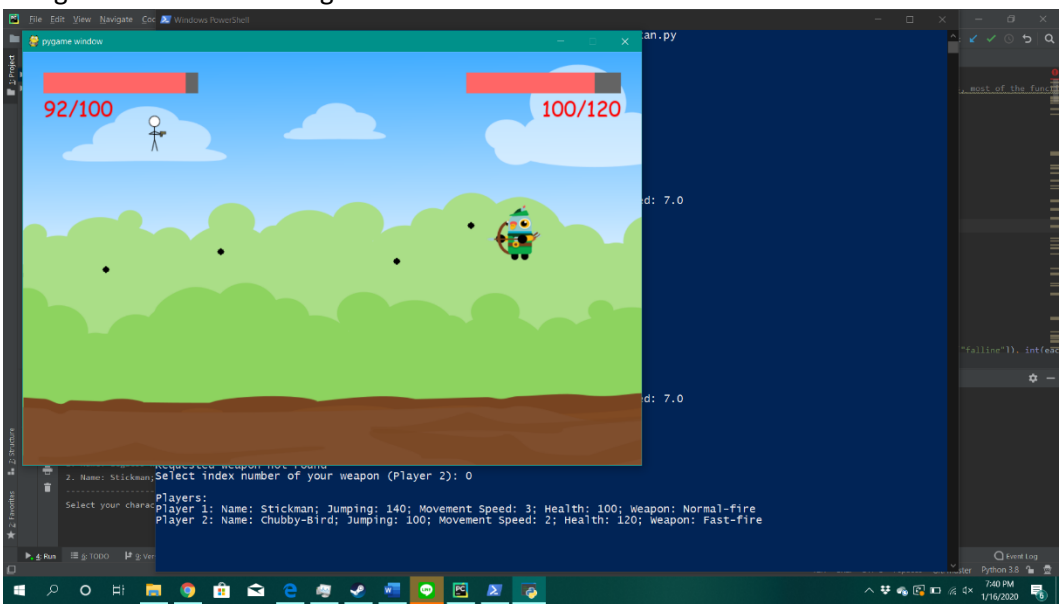
g. Pygame window appears with basic instructions how to play

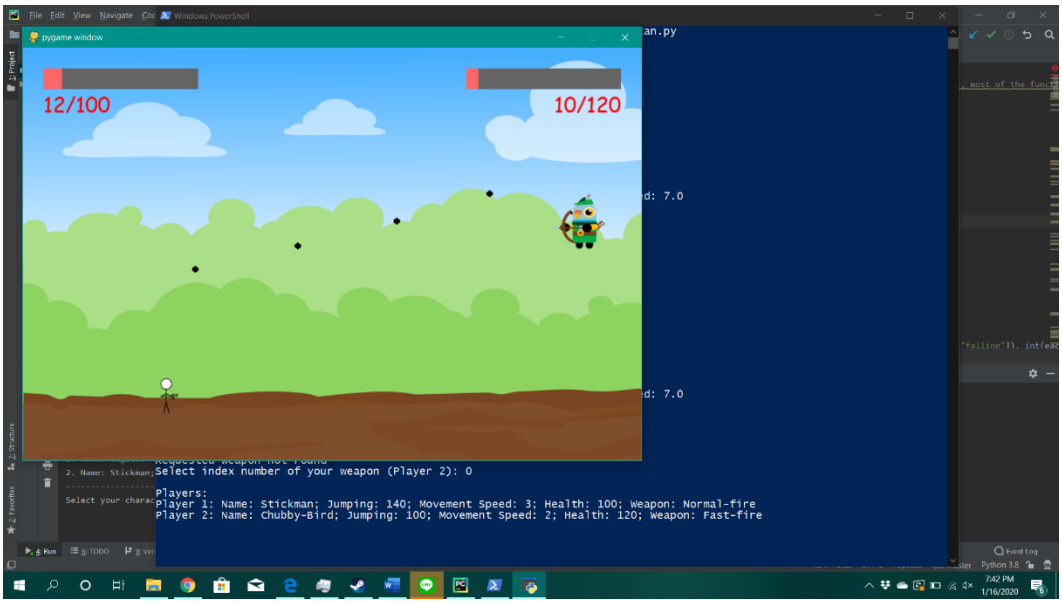


h. Clicking ok button will start the game

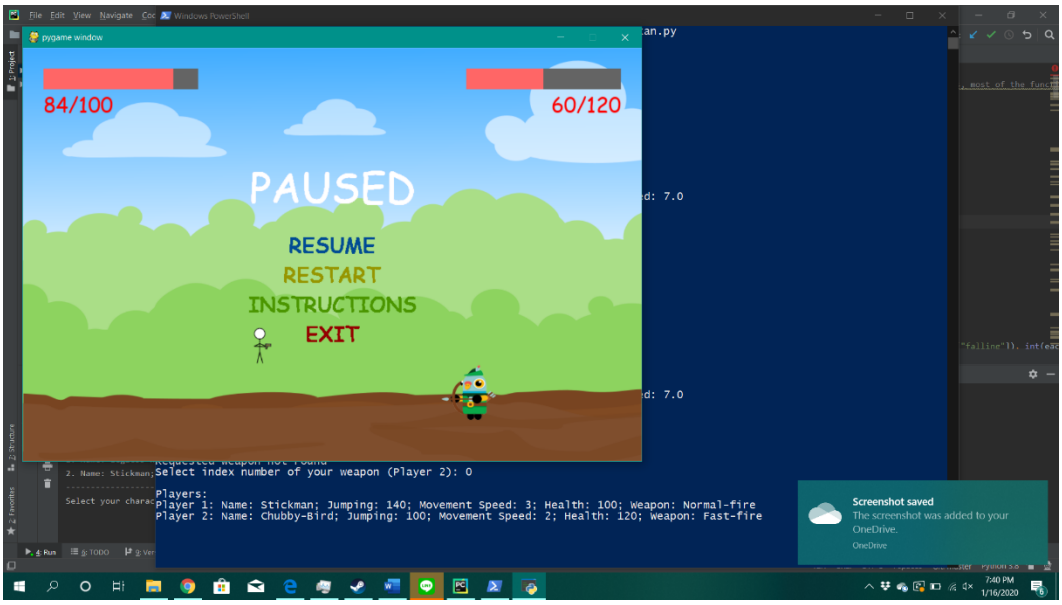


i. The game when it's running





j. The game when it's paused



k. The game when a winner has been decided