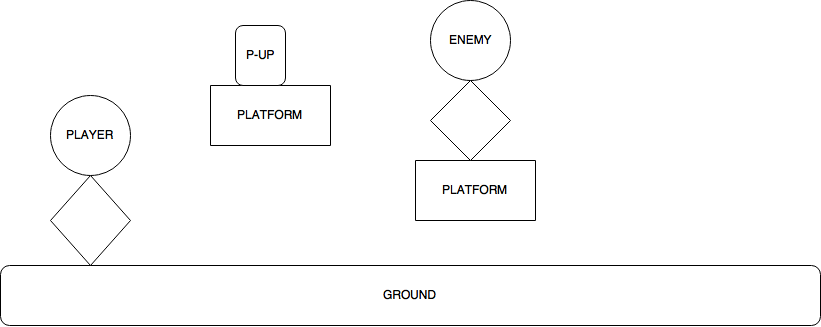
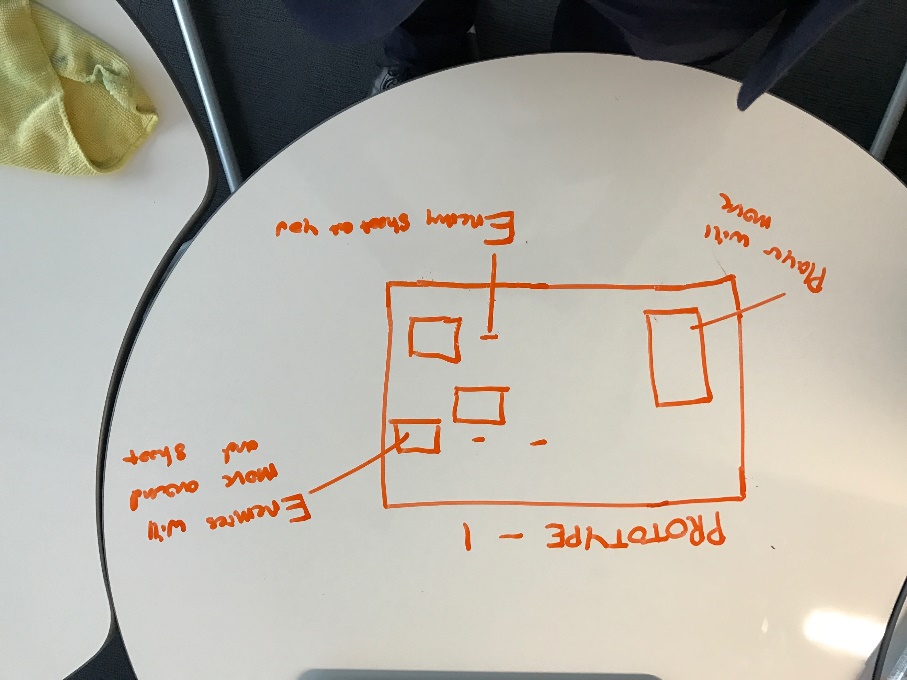
## FARHAN HABIBIE FINAL PROJECT BINUS SEMESTER 1

## Interface

Here is the design of the Interface:



This is an initial design of the map that I have come up with after the first interview. It’s a very simple design without any colour. The colours for each of these objects will be decided at a later point in time and another design of this will be added. The background colour will also be included at a later point unless the end user specifies that he wants the background to be an imported picture.

This design consists of the basic platforms, a power up, an enemy, the ground and the background.

This is a hand drawn diagram that I have drawn to show the basic interface of the game. In this first prototype, I had only included the player and the enemies and some bullets that may be fired by the enemy or bullets that may be fired by the player. This may all be subject to change depending on how the interviews go.

## Data Structures

### Classes Objects

In order for the game to work classes need to be utilised so that the coding of the game would be easy. I will first start of by defining the classes needed for the creation of the map. I will draw a series of rectangles which represent, platforms which would be under the class name of wall because I think that would make it easier as wall is easier to type than platforms. The map would also include power-ups and health-packs which would be both drawn as different coloured rectangles. Each different class would have similar attributes but some may have different attributes. These can be seen by the class diagrams bellow.

### Platform Class:

The platform class is defined by the name WALL. The attributes it holds are the

Colour – Colour of Wall

Width – Width of Wall

Height – Height of Wall

X\_Position and Y\_Position – Of the Wall

### Player Class:

The player would be coloured in white as mentioned above and would be in the shape of a small square with height of 20 and width of 20.

### Size screen

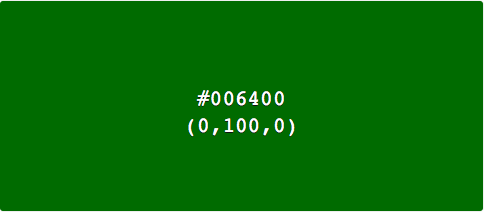
The size of the screen is defined through using global variables that have been defined at the top of the program. The screen will have a width of 800

### Health

The health bar will be drawn as a rectangle on the top right corner of the screen. The health bar’s width and height will be defined by bar\_height and bar\_width. The health bar will have a black background and a red healthbar.

### Colour Scheme

In terms of the colour scheme for my project, I chose one that would resemble the overall theme of metal gear solid (a jungle like theme). This would mean that simple single colours are enough to give the overall feel of the metal gear solid type of game. For the background I chose a dark green colour because I think that it fits the theme of a jungle as it has a lot of dark green leaves.

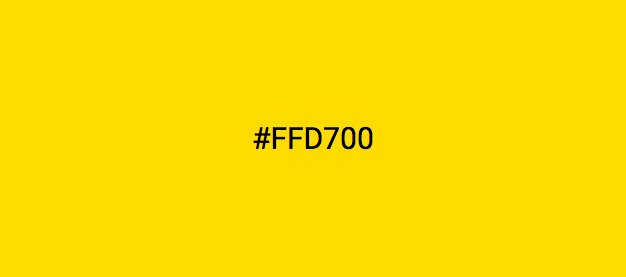
For the platforms I chose a crème colour scheme because I think it’s a simple colour.



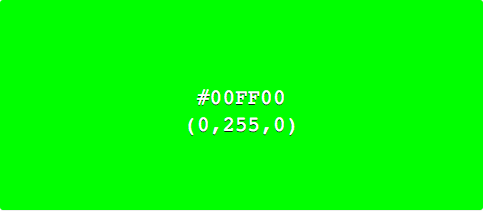
I think that it can contrast and stand out from the dark green colour of the background.



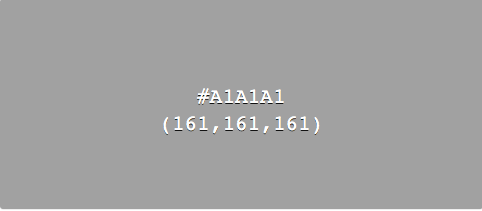
For the colour of the enemies I chose a red colour. This is because the colour of red is synonymous with danger and the colour red stands out easily which means that the player could easily differentiate from the player and the enemy.

As for the player I chose a white colour because I think that the colour white is pure and is a good representation of the player and I think that the colour white would also stand out against all the other colours here.

For the powerups I'm chosing a golden colour because the golden colour I think is quite attractive for the player and would want to make the player to run and get the power up.



For the colour of the health packs I will be choosing a very light green colour. This is because light green is a very eye catching colour and that usually in games it signifies health and in general something that is good.

The bullets would be coloured grey because I think that grey stands out from the background of the game and that grey is a very fitting colour for a bullet in a game as it sort of resembles the silver shining colour of some bullets in real life.

## Algorithms

### Displaying screen

This is the algorithm used to display the screen of the game:

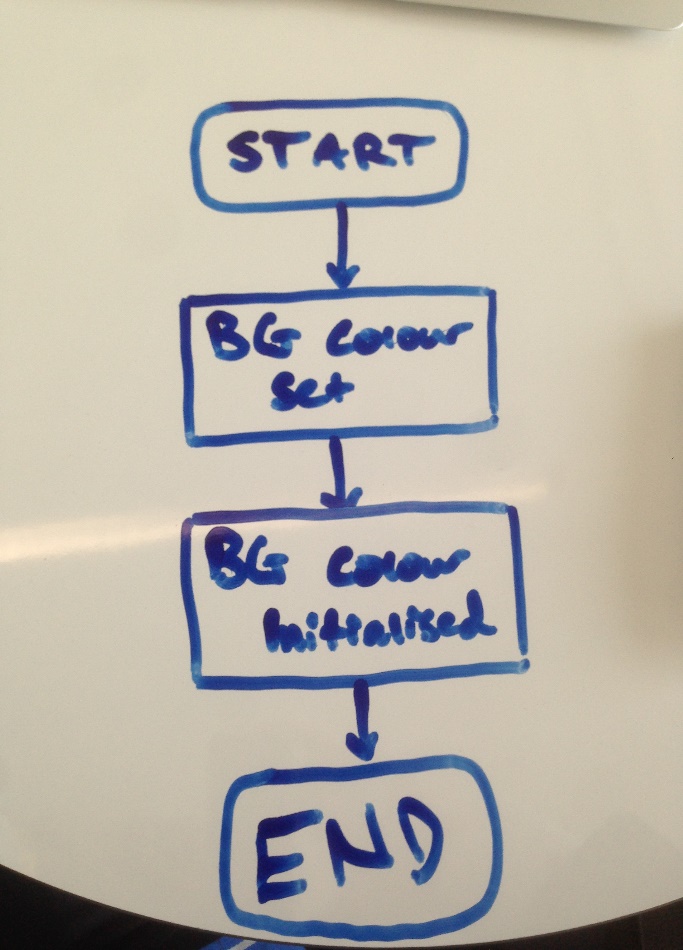
### Initialising the map

The map gets initialised by drawing in the platforms and drawing in the player at the correct positions. Platforms will be drawn in X\_Position and Y\_Position

### Initialising the class objects

Since this is a non-functional prototype, there are no algorithms involving classes that need to be initialised. The only algorithms that gets initialised is the drawing code.

### Flowchart and Pseudocode for background:

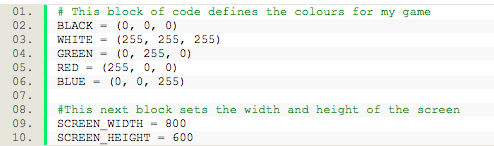
This flow chart shows the process of how the background colour gets initialised. It starts by defining the background colour to any colour (in this case blue). Then once the program gets run, the blue background will appear on the 800 x 600 screen.

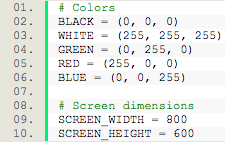
PSEUDOCODE:

Screen%20Shot%202017-03-17%20at%2010.02.08.png

### Pseudocode for colours:

The colours will be defined through declaring the as variables:

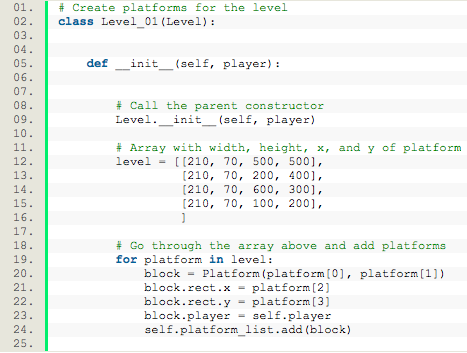




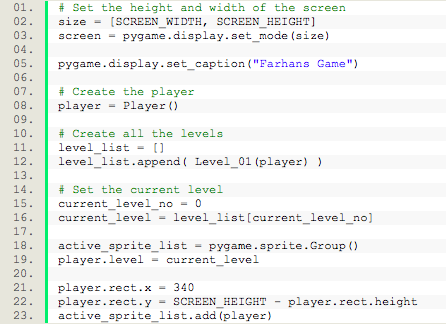
The first part of this code is basically just defining the colours that will be used through out the game. These are declared as global variables because it will make it easier to use through out the game as I would not need to re define the colours for each class that will be made. These variables are given names that are specific to the colour itself so that there wont be any confusion.

The second part of this code defines the screen width and the screen height. These value will be fixed and have been chosen by the end user (ADD REFERENCE FROM INTERVIEW). When this code is run an 800 x 600 screen pops up which is where the game will be played on.

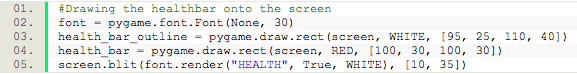
This code imports the pygame library into python. The pygame library allows easier access to pre programmed commands that can be used in the game. One such commands would be the draw command. This allows things to be drawn onto the screen.



This piece of code shows how the platforms are drawn onto the screen. Its takes the platform width and height and the x and y coordinated of the platforms.



This piece of code shows how the levels are drawn and where the player will be drawn. At the bottom it shows that the player will be drawn at the x value of 340 and y value of the bottom of the screen.



This piece of code shows how the health bar is drawn onto the screen. It takes a starting font, which in this case would be a default font from python. Then it draws the first outline of the health bar. After drawing the outline of the health bar, it then continues by drawing the actual health bar it self. After this it follows by using screen.blit to draw out the words “HEALTH” itself.

### The player class:

In the second iteration, the player class has changed quite a lot. Since the first iteration was a non functioning prototype, the player class had only 3 attributes and methods and those were Colour, Width and Height. However, now in the first iteration, since the player is able to shoot and move around and has collision detection, the list of attributes and methods increase. One example would be the addition of self.change.x and self.change.y. These are used to help the player move left and right and jump up or down.

The player would be coloured in white as mentioned above and would be in the shape of a small square with height of 20 and width of 20.

The player can be moved by using the arrow keys with left being moving left, right being moving right and up for jump. In addition to this the space key would be used to shoot the bullets. The direction the bullets travel in depends on which way the character is facing.

Through the use of collision detection, the player would not be able to pass through the platforms created with the map.

The player would also not be able to go off the screen by adding boundaries with the collision detection. This would mean that the edge of the screens would have to be treated as walls.

The player will lose some health when colliding with an enemy and if it collides with an enemy multiple times then the player would lose a life.

If the player happens to shoot an enemy and kills that enemy then the players score will be added by +100. The player will advance once all enemies have been killed.

Collision detection would be used in order to detect whether or not the player has collided with power-up or a health pack and if the player has then the health pack or the power-up would disappear.

### The bullet class:

The bullet will be a small have a colour of grey as defined above and will have a width of 5 and a height of 2.

Following the player and the enemy classes, the bullet will also have collision detection. This is used so that if the bullet collides with an enemy or a platform it will disappear. However it is not as important for the bullet to have collision detection with the edges of the screen because if the bullet goes off screen it can just keep on travelling and will not be seen again.

If the bullet collides with the enemy then it will dissiapear and take away a portion of the health of the enemy. This can be done by assigining a damage value to the bullet and taking that value away from the enemies health when the enemy has been hit.

### The power-up class:

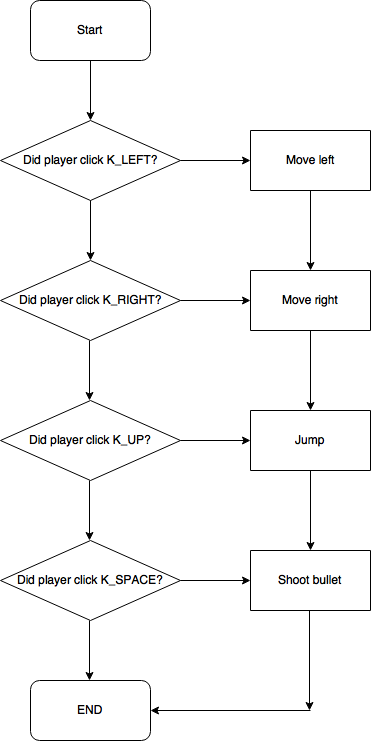
The power-up will have the same colours as how it has been specified above. The size of the power-ups will be determined by the variables of width and height. The powerups will appear at different locations depending on the level.

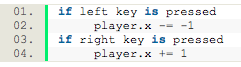
### The platform class:

The platform class is defined by the name WALL. The attributes it holds are the colour, width, height, X\_Position and Y\_Position.

## Algorithms Design

### Player Movement

This is the flowchart for the basic logic of player movement. As you can see, if the user has clicked the LEFT or RIGHT key then the player will move to either the left or the right. If the user has clicked the UP key then the player will jump up. Lastly, if the user has clicked the SPACEBAR key then the player will shoot a bullet in the direction that the player is facing in. As can be seen by the flowchart, the movement is done by IF statements defined by the diamond boxes so that the game can check whether or not the key has been pressed.

This is the pseudocode for the player moving left or right:

As can be seen in the screenshot above, this is the basic pseudocode for player moving left or right. The first if statement checks to see if the left key is pressed or not. If it is pressed then the players x position gets changed by -1. If we are looking at the coordinate axis of a game, -1 would be in the left direction as it is not in the defined positive direction. This is set by the pseudo variable for the players position (player.x) being set -= -1. The number will keep decreasing because “-=” used. The “-=” means that as long as the left key is pressed, then the players x position will keep on decreasing instead of only decreasing by 1, once. The second if statement checks to see whether or not the right key has been pressed. If the right key has been pressed then the players x position gets changed by +1. This means that it will move in the defined positive direction of the game screen. The number will keep increasing as the right key is pressed because the “+=” command is used. This means that it will keep on continually adding 1 to the players position as long as the right key is pressed.

### Player Jumping

../Screen%20Shot%202017-03-19%20at%2021.38.24.png

This is the basic pseudocode for the player jumping. This if statements checks to see if the player has clicked the up key or not. If the up key is pressed then the player.y position will get changed by +2. I chose 2 as the number for this because if 1 is used, then the acceleration upwards will be to slow resulting in a slower jump. As with the case above, “+=” is used so that the players.y coordinate will keep on increasing. The only problem with this right now is that, if the player presses the up key then the player will keep on moving upwards essentially flying. This problem will be solved later on once the gravity function has been defined. The

### Gravity

../Screen%20Shot%202017-03-19%20at%2023.13.36.png

This is the pseudocode for gravity. What this if statement is doing is it is basically calculating the gravity for the player. If the player is still on the ground then the player is able to accelerate faster up by 1. However if the player jumping (player.y>0) then the y value will start to decrease by a value of 0.35 until the players reaches the ground equaling 0 again.

### Bullet

This is the flowchart for shooting the bullet. As can be seen 2 decisions have to be made. The first one is whether or not the space key has been pressed if yes then it goes to the second decision where it checks the direction the player is facing and shoots the bullet accordingly.

Pseudocode for dimensions and colour of bullet

../Screen%20Shot%202017-03-19%20at%2022.44.57.png

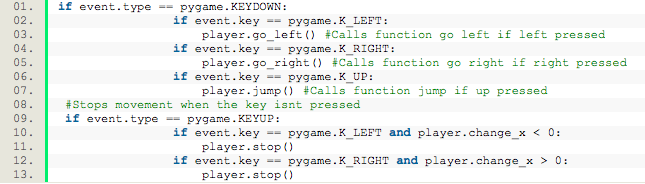
This piece of pseudocode is for the drawing of the bullet. The bullet will have a width of 5 and height of 3. The colour will be silver since it has been specified by the end user. The bullet will need to be drawn first because, if the bullet isn’t drawn then the player will not be shooting anything.

../Screen%20Shot%202017-03-19%20at%2022.30.56.pngPseudocode for shooting bullet

This is the pseudocode for shooting the bullet. The first if statements checks to see if the space key has been pressed and checks the whether or not the player is facing the left direction. If both are true then the player will shoot a bullet in the negative direction at a speed of 6. The direction of the bullet is again determined by the “-=”. This means that it will essentially shoot a bullet ‘backwards’ instead of ‘forwards’ as the positive direction is defined as going to the right. Again “-=” is used so that the bullet continually keeps on going left. The second if statement checks to see if the space key has been pressed and checks if the player is facing the right direction. If both of these are true then the player will shoot a bullet to the right at a speed of 6. Bullet direction is determined by the “+=” meaning that it will shoot a bullet ‘forwards’. “+=” is used so that the bullet keeps on going towards the right direction instead of being shot and just stays in the same position.

## Implementation

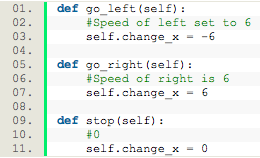
### Key stroke for movement



This block of code is the basic code for used to check for whether or not the user has pressed an arrow key or not in order to control the player. It uses an event type called pygame.KEYDOWN which is built into the pygame library to check whether or not the user has pressed a key or not. It then uses if statements to check each of the different key strokes in order to execute functions that actually makes the player move.

Then the next block of code uses the event type called pygame.KEYUP to check whether or not the user has released the key. If the user has released the key and the player was already moving, then it will then execute the function player.stop in order to stop all movement in the direction that the player was moving in when the key was pressed down. The other condition used in this, which for moving left was player.change\_X < 0 is used as, when the player is moving left, it is essentially decreasing its X value because it is going in the minus direction e.g. when moving left it may be “ -= -2”, meaning that the more it moves to the left the lower the value of X. Then when moving right, since the X value keeps on increasing because it is moving in the positive direction as for example it uses “+= 2”, it checks to see if the X value is greater than 0 (which it will be when moving to the right). Then for both of these if statements, if both conditions are met, only then will it execute the player.stop function.

### Moving left or right #1

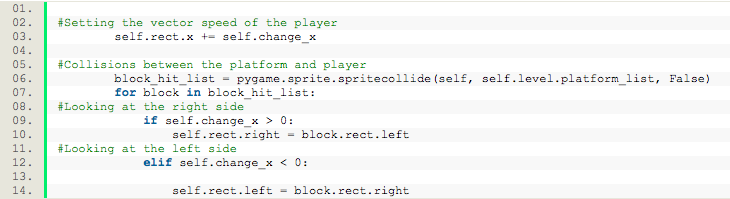


This block of code defines the functions go\_left, go\_right and stop, all passing the parameters of self, which is the player. When the user hits the left key, the go\_left function will execute which sets the players self.change\_x value to -6. This means that the player will keep on going left as the value is negative and will move left at a speed of 6.

The go\_right function will execute when the right key is being pressed down. When the right key is being pressed down then it will set the self.change\_x value to +6. Since this value is positive, it means that the player will move to the right according to the coordinate axis and will move to the right at a speed of 6.

The stop function will only occur when the left key or the right key has been released. The stop function will set the self.change\_x value to 0, meaning that it will not move at all as it has 0 speed. This also means that the player will stop at the position it was last left on. This will not reset the position of the player as it is not minusing or adding any value to the players current x position but instead just setting it back to 0. This means 0 speed essentially.

### Moving Left or Right #2

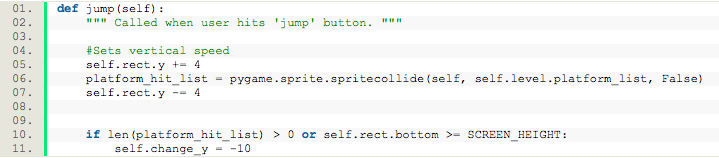


The top line of this block of code is an easier method to implement player movement as all it takes is to change one variable which is self.change\_x in order for the player to move left or right. The self.change\_x variable keeps on changing when the user has clicked left or right as it is a variable which has been defined before with different values for moving left or right.

The second block of code here is collision detection for the player so that the player would not be able to move through platforms or through the end of the screen. It first starts by creating a block hit list. It then executes the pygame.sprite.spritecollide event which has been provided by the pygame library and passes through the parameters of self which is the player, self.level.platform\_list which is the group list for all the platforms that are being drawn onto the screen and False for when there is nothing to collide against.

The next part after initiating the block\_hit\_list is a for loop, which keeps on looping through the block\_hit\_list and keeps updating itself whenever a change has been made. In this for loop, there is an if statement which checks if the player is moving to the right with the condition self.change\_x > 0 (which it will be when moving to the right). If this condition is met then it will set the right side of the player to the left side of whatever is being hit. The elif part does the same thing except the opposite where when moving left, it will check if self.change\_x < 0 (which it will be when moving left), and will set the left side of the player to the right side of whatever is being hit.

### Jumping #1



This block of code defines the function jump which passes through the parameter self, where self is the player. It is executed when the user clicks the up key.

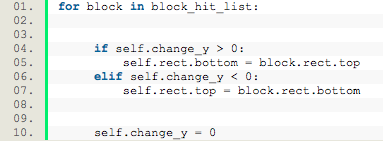
When the user clicks the up key, then the players self.rect.y gets changed to 2 which makes the player jump up. It will then use collision detection with the parameters of self.level.platform\_list for all the platforms in that group list and false for when it touches the ground. This is so that when the player lands back down it doesn’t go through the platform and doesn’t go through the ground.

After this, it will decrease the self.rect.y value by -2 again upon landing so that it moves down 2 pixels meaning that it will land nicely.

The if statement in this block of code is used to check if the player is on a platform or if the bottom of the player is at the bottom of the screen. If these conditions are both true then the player will be able to jump by a value of -10.

The value is -10 because when looking at the coordinate axis of a screen, it starts at the top right corner and going down is the increasing value for the y coordinate. This means that if the y value is -10 then essentially it is going upwards because the y value is decreasing towards Y = 0 which is at the very top of the screen. When the bottom of the player is >= Screen height then this basically means that the player is not at the top of the screen and there is still space for the player to jump, where as if the Y value of the player is = Screen height then this means that the player is at the top of the screen and it would not be able to jump up.

### Jumping #2



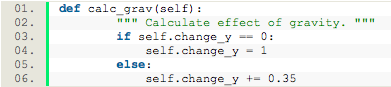
The top line of this block of code is an easier method to implement player jumpingas all it takes is to change one variable which is self.change\_y in order for the player to jump. The self.change\_y changes when the user has clicked the up arrow key as it is a variable which has been defined before with the value for self.change\_y.

The second block of code here is collision detection for the player so that the player would not be able to jump through the platforms or land through the floor. It first starts by creating a block hit list. It then executes the pygame.sprite.spritecollide event which has been provided by the pygame library and passes through the parameters of self which is the player, self.level.platform\_list which is the group list for all the platforms that are being drawn onto the screen and False for when there is nothing to collide against.

The next part after initiating the block\_hit\_list is a for loop, which keeps on looping through the block\_hit\_list and keeps updating itself whenever a change has been made. In this for loop, there is an if statement which checks if the self.change\_y value is > 0. If this condition is met then the bottom of the player sprite will be set to the top of whatever is being hit. The elif part checks if the self.change\_y value is < 0. If it is then the top of the player sprite will be set to the bottom of whatever it is hitting.

After this the value of self.change\_y will be reset back to 0, allowing the player to jump again.

### Gravity

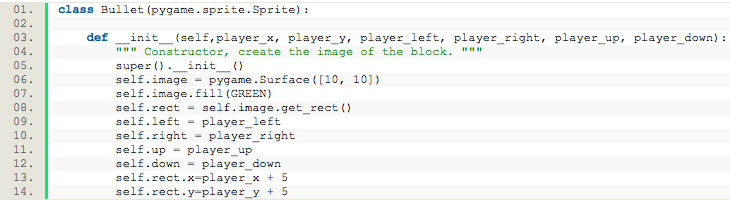


This is the function for calculating the gravity in the game. The first line defines the function and passes the parameter of self which is the player. The next if statement is the calculation of gravity. Firstly it starts by checking if the player self.change\_y value is 0. This is essentially checking if the player is not on the air as the speed of the y value is 0. If it is then they can set the speed value of the player to 1 when the jump key has been pressed.

The else statement in this code only occurs when the player is already in the air. When the player is in the air. When the player is in the air, then the y value will keep on increasing by 0.38, meaning that the player will slowly fall down.

Although it seems counter intuitive to increase the Y value to make the player fall, the reason for this is because the lower part of the screen has a higher y value than the top of the screen, so if thought about logically then increasing the y value of the player would mean that the player would fall down towards the bottom of the screen instead of flying up.

### Bullet #1



This first block of code is creating the bullet class, and initialising the drawing of the bullet itself. The class bullet takes in the parameter pygame.sprite.Sprite. Then the definition of the initialisation takes in the parameters: self – for the player, color – to define the colour of the bullet, and player = True which is a Boolean variable set above.

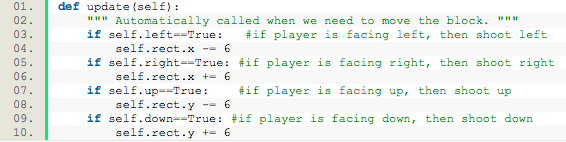
The dimensions of the bullet have been set by pygame.Surface which takes in the desired width and height of the bullet.

The colour of the bullet is determined by self.image.fill which takes in the variable color to fill it with the desired color.

The bullet will be drawn onto the screen by the line which says self.rect = se.f.image.get\_rect. This essentially gets the shape of the bullet and draws the bullet onto the screen.

The self.player = player line links the variable self.player to the player class so that it can get attributes from the player class to use.

### Bullet #2



All this does is it sets the speed of the bullet depending on which way the player is facing. When the player is facing right then the bullet will be fired at a speed of +6 meaning that it will be increasing. When the player is facing the left it will fire at a speed of -6. This means that it will be firing decreasingly. When the player is facing up then it will fire a bullet at a speed of -6 in the y direction. This means that the bullet will be fired upwards. Lastly, when the player is facing down then it will fire a bullet in the y direction but the speed will be positive.

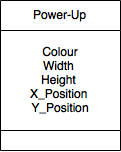
This is due to the way coordinate axis are calculated in this game.

## Design

### Interface

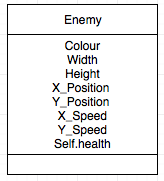
For the interface of the third iteration, I will be discussing how the score, the lives and the health bar will be situated on the screen, and how the start screen should look like and how the end screen will look like.

### Data Structures

The power-up:

The power up class will have a colour set as the one specified above for the power up. The width and height will be 5 x 5. This is because the power up should be square. This will be set by the variables Width and Height. The power up will appear at set positions depending on the level. This is determined by the X\_Position and the Y\_Position.

The power ups will also have collision detection. The purpose of the power ups collision detection is so that, when the player touches the power up, it can disappear and trigger the function to make the player stronger. An example of what a power up could do would be that it could speed the player up. Collision detection will also be used so that if the power up §§§§§§§§§§§§§§§§is spawned on a platform then it would not fall through the platform but instead would stay on the platform.

The enemy class:

For the enemies, there will be one type of enemy and that is the Red enemies. These enemies will be a square, however they would be slightly larger than the player itself. These enemies will have a width of 30 and a height of 30.

Just like the player, collision detection would be used so that the enemies would not be able to go through the platforms and would also not be able to go off screen.

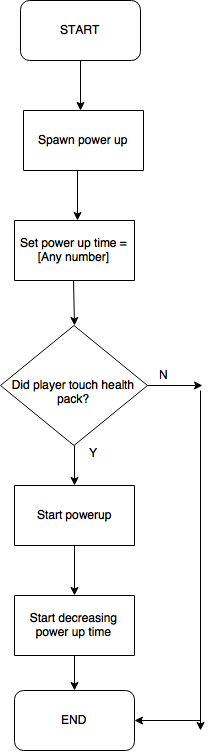
If these enemies happen to get hit by a bullet shot by the player then the enemies will die and get deleted of the screen.

Each level will have varying amounts of enemies, with the idea being that in the first level there is less enemies than in the last level.

The stronger enemy will be made by changing the value of self.health of the enemy class. For example The weaker enemy will have a self.health value of 50 where as the stronger enemy will have a self.health value of 100-150.

The lives the player has will not need to be implemented in a class. This is because all it needs to do is be drawn onto the top right hand corner of the screen. This means that the lives will just be text that would have a number next to the word ‘Lives’, that would decrease every time the player loses a life. The lives can be initialised just by declaring a variable named lives = 3, and just printing the number of lives on the screen.

The score of the player also will not need to be implemented as a class but can also just be drawn on the screen normally. The score will also just be text that will be drawn in the middle of the screen, and will have a number next to it that will go down every time the player has been hit by an enemy bullet. The score can be functional by having it implemented the same way as the lives was implemented. A variable called score will be set to 0 and that number can just be printed each time.



## Algorithms

### Power up #1

This is the flowchart depicting the basic logic for the power ups. There will be 3 more flowcharts after this flowchart that will be demonstrating how:

1. The spawning algorithm for the power ups will work
2. How the power up timer will work
3. How the power up can be applied to the player

This first flowchart only shows that, when the game starts a power up will be spawned at a location in the map. Then as the power up has been spawned, then a power up timer will be created along with the power up. Then a decision has to be made. If the power up has been ‘touched’ by the player then it will start the power up and change the stats of the player. When the power up has started, the power up timer will slowly decrease for the duration of the power up until the time reaches 0. Once the time reaches 0 then the power up will end. However the other decision is when the player does not touch the power up. If the player does not touch the power up then the power up will just stay on screen and nothing will happen unless the player decides to get the power up.

### Power up #2

This second flowchart shows the decision making behind how the power ups will be spawned. Firstly a number between 1 to 5 will be chosen randomly. These numbers ranging from 1 to 5 will correspond directly to a position on the map. For example, 1 could be the position X = 200, Y=30.

Then if that number between 1 to 5 has been chosen, it will spawn the power up in the location that corresponds to the number that has been chosen.

The number will be randomly chosen through the use of a different library within python that allows for randomisation. An example of this library would be the random library in python.

Once the power up has been spawned, then the function will end so that no more than 1 power up can be spawned at the same level.

### Power up #3

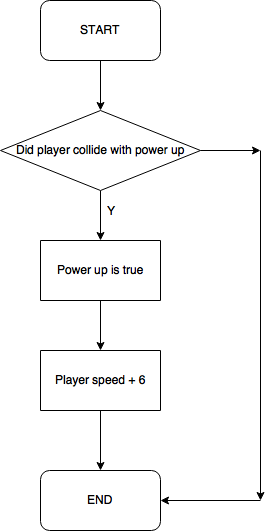
This next flowchart is the basic process of how the power up timer will be implemented. Firstly it starts by setting the power up timer to 0. This means that when the player picks up the power up it will have a start time of 0.

Then the power up timer will be incremented by whatever number that has been set to increment by. For example it can be incremented by 1 every frame or by 2 every frame.

Following this, a decision has to be made. The decision is made by using the function mod. The mod function makes it so that, if a number is ‘modded’ by the set number then it will find the remainder of that number.

If the answer to that decision is no (FALSE) then the power up will be kept on, and the player will have all the benefits of the power up. However if the answer to the decision is yes (TRUE) then the power up will be turned off, and all status changes to the player will be reversed back to the original values and it will end.

### Power up #4

This last flow chart for the power up shows what actually happens to the player once the player has picked up the power up. When the player has picked up the power up, then the power up will be true.

This will then mean that in this case, the players speed will be increased by 6 from its previous value.

Then once the timer for the power up has ended then this function will end and reset the players speed back to the original value.

### Screen%20Shot%202017-03-27%20at%2022.41.09.pngRandom location

This is the pseudocode to make the power up appear at different locations throughout the map. It starts by initialising an array between 1 to 5. These numbers all each correlate to different x and y positions on the map. Then once a number between 1 to 5 has been chosen, it will go through a couple of conditions. If the number chosen is 1 then it will create a power up at the location corresponding to that number. Else if the number is 2 then it will create a power up at the location that corresponds to the number 2. This will keep on going between 1 to 5. Only one power up will be created at these locations. Since it is random, it is entirely possible for a power up to be created at the same position.

### Screen%20Shot%202017-03-28%20at%2010.40.31.pngTimer

This is the pseudocode for the power up timer. It starts by setting a variable called powerup time = 0. Then the power up timer will be incremented by 1 each time until it reaches 10000. The power up time will keep on being incremented through the use of a loop. Once the power up time has been set to 0 and has been incremented by 1, then it will create the effect of the power up. Once the power up has reached 10000 it will be modded by 10000 which is the same value. Since it is being modded by the same value, the remainder of the number is 0. Once it reaches 10000 and gets modded by 10000, equaling 0, the power up timer will be set to off then the power up will be turned off.

### Effect

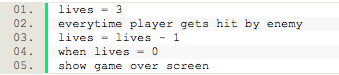
Screen%20Shot%202017-03-28%20at%2010.47.52.png

This is the pseudocode for the effect that the power up will have on the player. The effect it will have would be increasing the speed of the player. Firstly, there needs to be collision detection. Collision detection is needed for collision with the platforms so that it does not go through the platforms, and collision with the player so that it knows when the player has touched the power up. Once collision with the player is detected, it will start the timer. As long as the timer is true, the effects of the power up will work. In this case, the speed of the player will be increased with the power up. The speed of the player is increased by setting the player speed = to the players original speed + the offset, where the offset is the added speed to the player making the player faster. Once the power up timer has run out, the speed of the player will be set back to the normal player speed by taking away the offset from the players speed.

### Lives

The lives will be implemented as drawing code on the screen. Theres no need to make a class for lives because all it is, is text on the screen that will be displayed at the top of the screen. As for how the lives will go down, there will be a variable named lives that will first start by equaling 3. Then once the player gets hit by an enemy, the number of lives will keep decreasing by 1. Once the number of lives has reached 0 then the game over screen will be shown.

### Lives Decreasing



This is the pseudocode for how the lives work. As can be seen, the lives will firstly be set = 3. Every time the player has been hit by the enemy, then the lives will be lives = live – 1, which means that the number of lives will decrease. Once the number of lives has reached 0, then the game over screen will be shown, indicating that the player has lost. This does not need to be done in a loop, but can instead just be done in a normal if statement where it checks if the player has been hit by an enemy. This is because the variable lives is always changing every time the player gets hit and will keep constant with lives = lives – 1. This means the value for lives will go from 3, to 2, to 1, to 0 without having to have it in a loop.

### Screen%20Shot%202017-03-28%20at%2011.30.24.pngDisplaying

The first line of this pseudocode is just printing the word lives and having the actual number of lives displayed next to it. The font size of this will be set to any size and it will be printed at any specified position on the screen. This is shown by (x,y). For example, since the end user wants it to be displayed in the middle, the position of the x and y could be (300,50) which would be the top middle of the screen. The colour of this could be set to any colour and the actual font of the word itself can be set to anything as well.

### Score

The score will also be implemented as drawing code on the screen. The score doesn’t need to be made in a class because it will just be text on the screen that will be updated every time the player kills an enemy. For the logic of how the ­­­score will go up, a variable named lives will start by holding a value of 0. Once an enemy has been killed, the score will be added for 1.

### Score Increasing

Screen%20Shot%202017-03-28%20at%2012.34.34.png

This is the pseudocode for how the score will be counted. First the score is set to 0. Then every time the player kills an enemy, the score gets added by the value of score that has been assigned to the enemy. For example if the score of the enemy killed is 1, then it will be score = score + 1. Doing the score like this means that every time the score is added, it wont go back to 0 but instead stay with the same value of score after it had been added. E.g. If score = 1 and enemy is killed then, score would be; score = 1 + 1. = 2.

### Displaying Score

Screen%20Shot%202017-03-28%20at%2013.01.29.png

The first line of this pseudocode is just printing the word score and having the actual score of the player displayed next to it. The font size of this will be set to any size and it will be printed at any specified position on the screen. This is shown by (x,y). For example, since the end user wants it to be displayed in the top right, the position of the x and y could be (500,50) which would be the top middle of the screen. The colour of this could be set to any colour and the actual font of the word itself can be set to anything as well.

### Enemy Basic Flowchart

This flowchart shows how the enemy class will be drawn onto the screen. Firstly it starts by calling the enemy function, then from that function it draws the enemy at the positions (x,y). Once that has been done, it refers back to the main loop of the program, where the actual drawing code is so that the enemy can be drawn onto the screen.

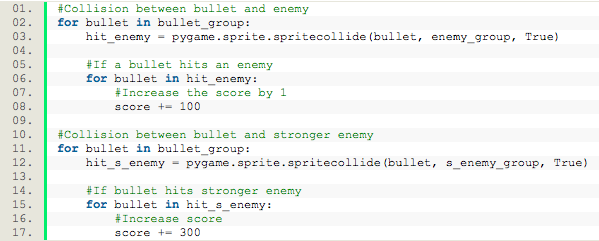
### Enemy Movement Flowchart

This flowchart shows how the enemy will be moving. Basically all it is showing is that first the enemy will move to the right from the top of the screen. Then once the enemy has moved right and reaches the end of the screen, a decision will be made where, if the enemy has reached the end of the screen, it will change direction and go to the left. This will also happen once the enemies have reached the left of the screen. They will also change direction to the right. Then the function will end.

### Score

../Desktop/Screen%20Shot%202017-05-12%20at%2010.34.04.png

This is how the score gets drawn onto the screen. It is just one line of code that sets the word score onto a screen and draws it at the position 400,10.

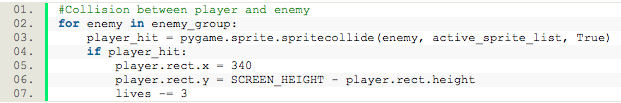


In this block of code, when the bullet hits the enemy, the enemy will disappear from the list and the score will be added by 100. The bullet sprite will disappear and the enemy sprite will also disappear from the screen. If the bullet hits the stronger enemy then the stronger enemy will disappear and a score of 300 will be added.

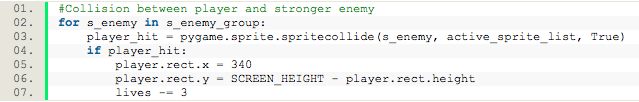
### Lives

../Desktop/Screen%20Shot%202017-05-12%20at%2010.43.57.png

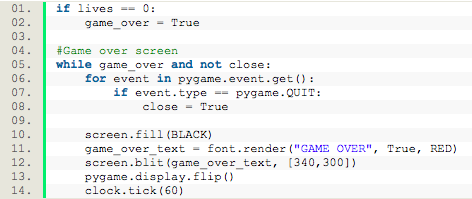
This block of code starts by setting the value of lives to 3. It will then draw out the string “Lives:” and display the number of lives left on the screen by converting the value into a string. It will then draw the lives at the position (600,40). The last part draws it on the screen.



This block of code is checks to see whether or not the player has been hit by an enemy. It uses collision detection to do so. If the player is hit, it will reset the position of the player to the start and the number of lives will decrease by 3. This means that it will trigger the game over.

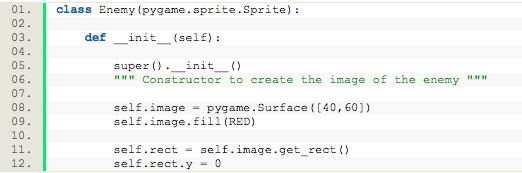


There is also collision against the stronger enemy.

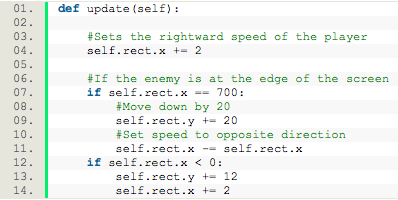


When lives hits 0, then the game\_over Boolean will become true and the game will end and show the game over screen.

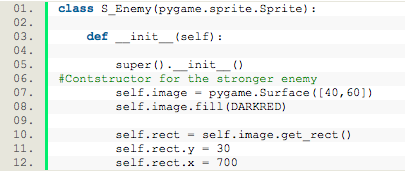
### Enemies

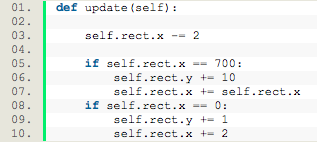


This block of code is what sets the basic values for the enemy. It is given dimensions of 40x60 so that it is a rectangle. It is then given a colour through the self.image.fill(color) and is then drawn onto the screen.

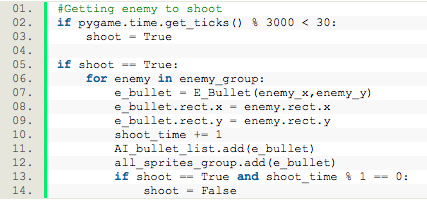


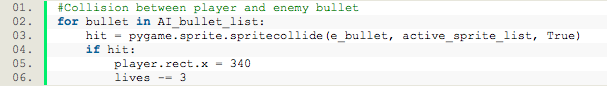
This next block of code is what helps the enemies move left and right. It uses an if statement to check the current position of the enemy. If the enemies position is greater than the screen size then it will change direction. If it is smaller than the screen size then it will change direction to the opposite way. This is all respective to the X coordinate. Depending on where the enemy is on the screen, it will then move down by a y value of 12.



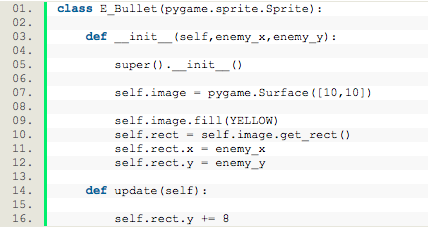
This block of code constructs the stronger enemy. The stronger enemy has the exact same dimensions as the normal enemy class. However the difference here is that the stronger enemy has a darker red colour and is drawn higher up on the screen.

This is the update function for the stronger enemy. The difference here with the normal enemy is that the stronger enemy moves much faster than the normal enemy and the way it moves is the opposite of the normal enemy. While the normal enemy is drawn on the top left hand side, the stronger enemy is drawn on the right hand side.

This piece of code is how I got the enemies to shoot at the player. It starts with a Boolean variable called shoot and the calculation of the time interval of when the player will shoot. Once that time interval is reached then shoot will be true.

It will then shoot the bullet at the current position of the enemy. Then we have a shoot timer to determine how long the enemy should shoot for. If there was no shoot timer then the enemy would just be continuously shooting forever. The shoot timer starts at 0 and is incremented by 1 according to the frame count. It then uses the mod function to see if 1 mod 1 = 0. When it reaches 0 then shoot = false. When shoot = false then the enemy will stop shooting the bullet.

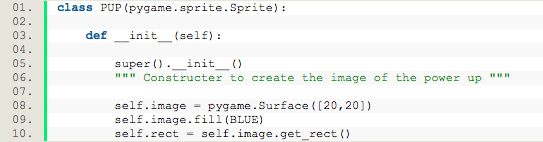
This block of code is the collisions between the enemies bullets and the player. When the player has been hit by the enemies bullet, then the player will lose 3 lives essentially meaning that the player will die.



This is the constructor function for the enemies bullets. It is named E\_Bullet for enemy bullet. The dimensions of this bullet is the same as the dimensions of the players bullets, however the difference here is that the enemies bullet colour is yellow.

The update function for this bullet just shows the speed at which the bullet will be fired at. In this case the enemies bullet will be shot at a speed of 8.

### Powerup

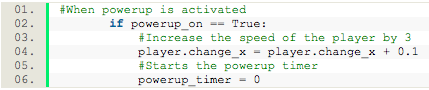


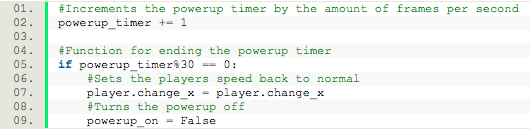
../Desktop/Screen%20Shot%202017-05-12%20at%2010.46.16.pngThis is the constructor function for the power up. The power up has dimensions of 20x20 meaning that it is a cube. The power up has a colour of blue.

This block of code shows how the collisions between the player and the power up has been made. It is between the player and the power up group.

../Desktop/Screen%20Shot%202017-05-12%20at%2011.33.21.png

This initialises the variables needed for the power up to work.



This block of code is the condition needed for the power up to have its effect. It starts by seeing whether or not power up is equal to true. If it is then it will make the players speed increase by 0.1. Although this value is small, the change in speed is noticeable. After this it will set the power up timer to 0.

The power up timer will then increment by one each frame meaning that it is incremented by the amount of frames per second.

It then uses a mod function again for the power ups timer. Once it reaches 30 and since 30 mod 30 = 0, it will make the players speed normal again and the power up Boolean variable will be set back to false meaning that it is turned off.