BINUS UNIVERSITY
BINUS INTERNATIONAL

Assignment Cover Letter

(Individual Work)

Student Information:
Name: Rafian Athallah Marchansyah
Student ID: 2440042380

Course Code : COMP6056 **Course Name :** Program Design Methods

Class : L1AC **Name of Lecturer(s) :** Jude Joseph Lamug Martinez
Major : Computer Science

Title of Assignment : Face Recognition Attendance
Submission

Type of **Pattern**

Assignment **:** Final Project

Due Date : 31-1-2021 **Submission Date :**

The assignment should meet the below requirements.

1. Assignment (hard copy) is required to be submitted on clean paper, and (soft copy) as per lecturer's instructions.
2. Soft copy assignment also requires the signed (hardcopy) submission of this form, which automatically validates the softcopy submission.
3. The above information is complete and legible.
4. Compiled pages are firmly stapled.
5. Assignment has been copied (soft copy and hard copy) for each student ahead of the submission.

Signature of Student: (Name of Student)

Rafian Athallah Marchansyah

"Hit or Run Game"

Name: Rafian Athallah Marchansyah

ID: 2440042380

I. Background

In the first semester at Binus International. The first few lessons of "Introduction to Programming" and "Project Design Methods" were all about teaching us about the basics of the programming language of Python. Near the end we were given a final project around November that was due in 2 months, about programming our own program that showcased our skills in using Python.

For this specific project, I had wanted to try my hands on creating a game using PyGame from scratch. I was inspired by the popular mobile game "Subway Surfer" to create a game where players would have to dodge things in the incoming opposite lanes in order to survive while trying to get the highest score. This was my first time ever learning about PyGame, so I was eager to try out this project.

II. Project Specification

Purpose:

Create a fun and challenging game using my own abilities from scratch using PyGame

Audience:

Any type of gamer that is interested in playing a "bullet-hell" type of game (a lot of dodging high speed projectiles being thrown your way)

Aim:

To create a functioning game that works like a "bullet-hell" type of game which also meets my personal requirements that is both fun and challenging for the player who plays it

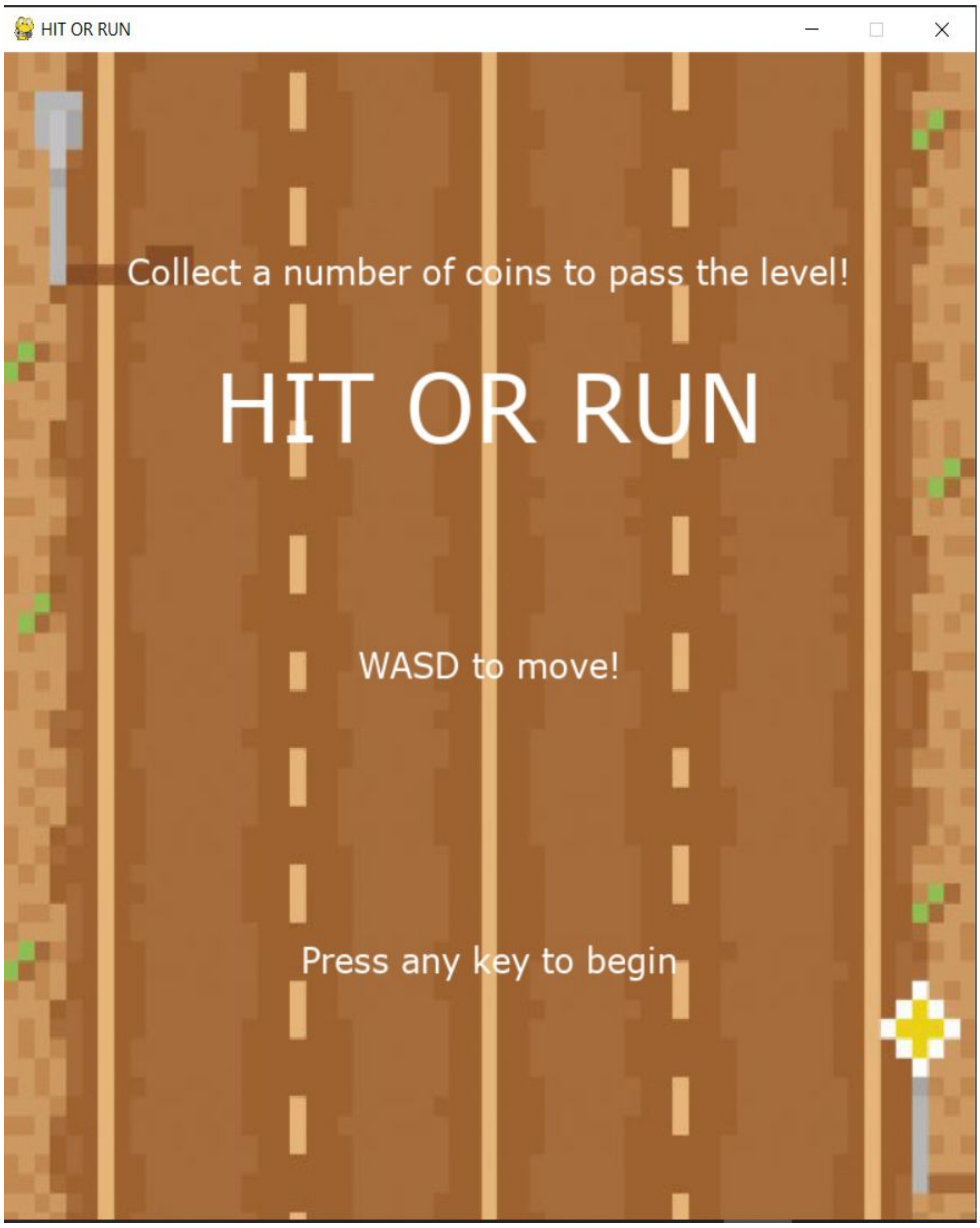
Personal requirements:

- Functioning enemy wave spawning system
- Proper collision function between sprites
- Scaling difficulty as level increases
- Coin collecting mechanism to add to the challenge of the game

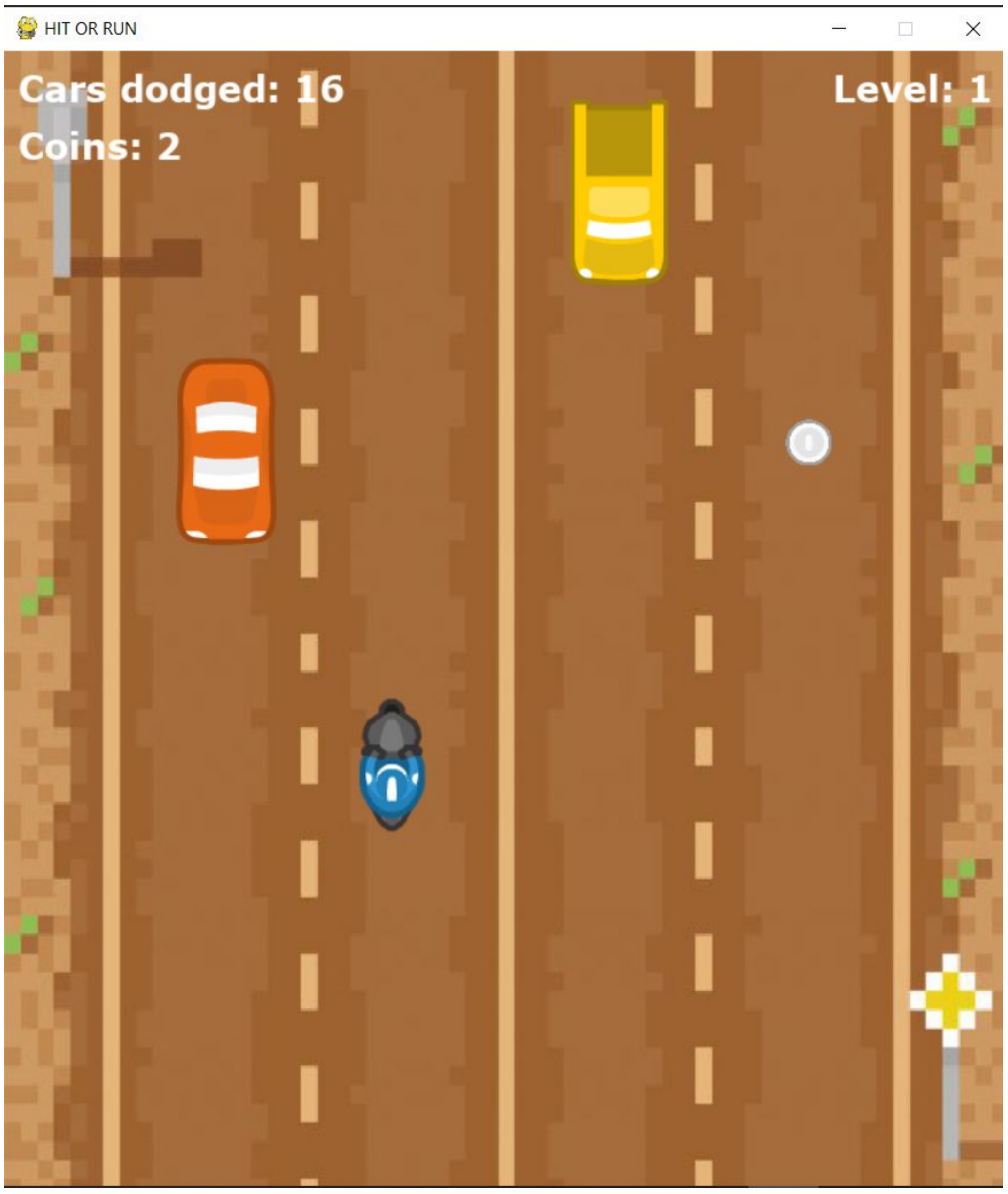
II. Program Description

“Hit or Run” is a program built using python 3.8, as well as the library module of PyGame. This program is a simple mini-game that can be played by any type of player that requires a little bit of skill to play and is looking for a challenge.

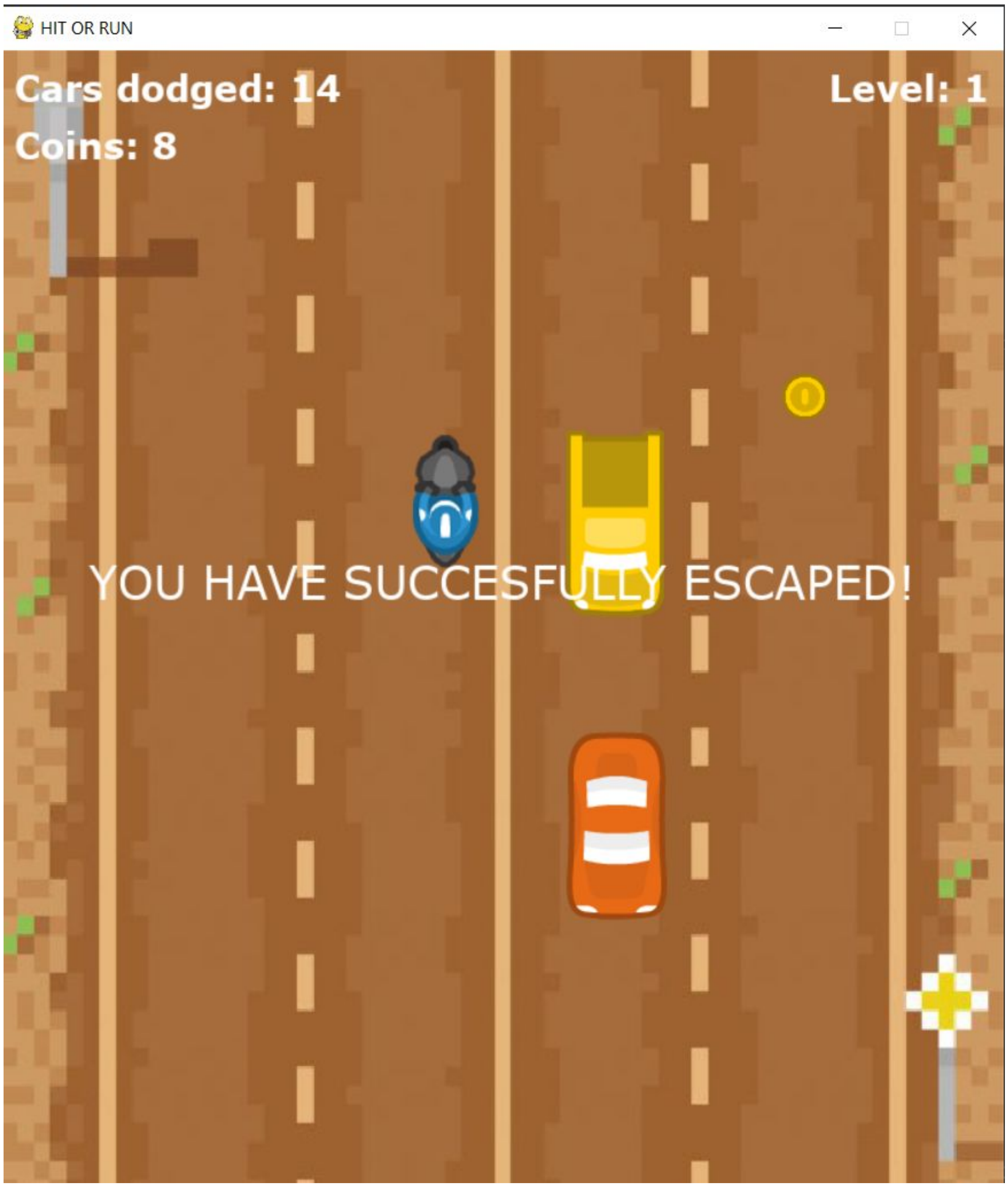
III. Program preview



On launch, the player is greeted with the main menu screen telling the controls of this game and the objective players need to complete to advance to the next level, which is collecting coins while dodging incoming cars.

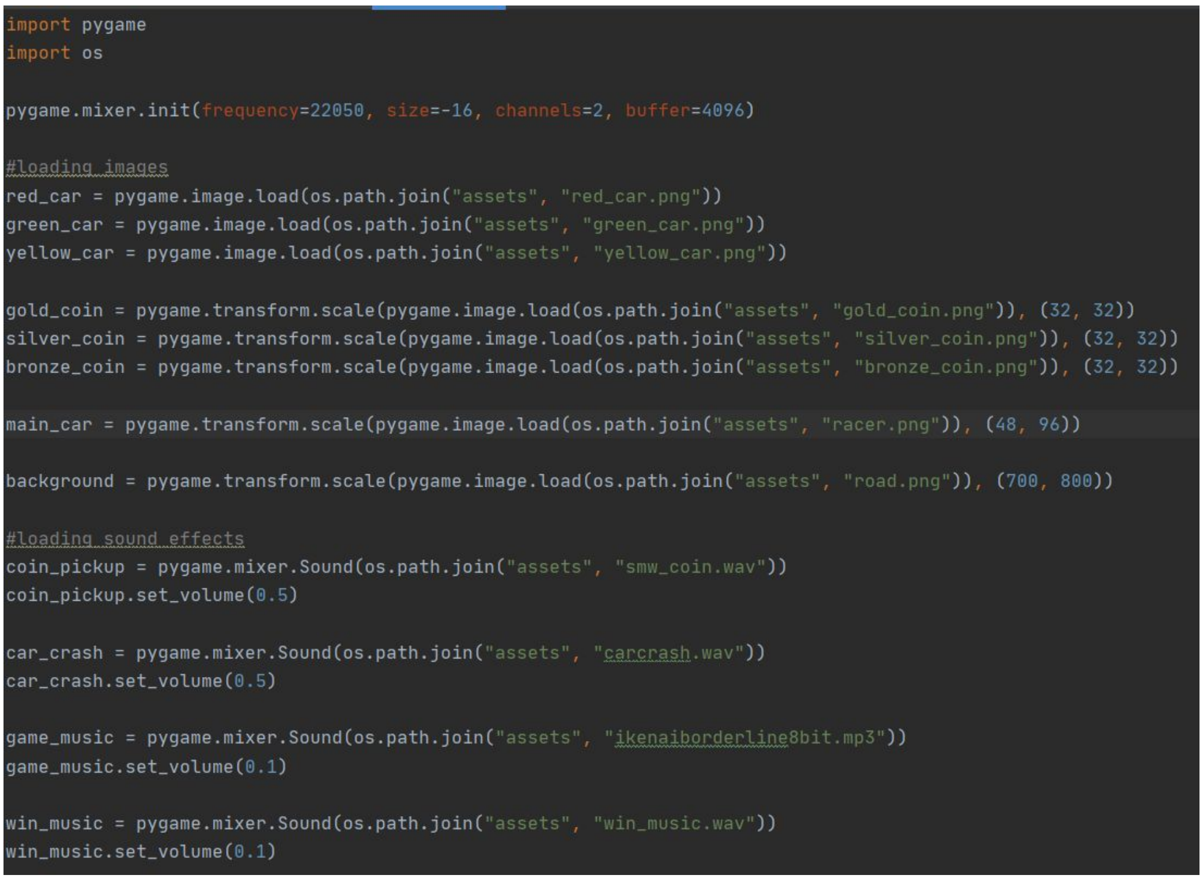


When the game is started, players can control the player (biker) using WASD. different vehicles will start spawning from the opposite side of the screen and will start making their way down. Using the controls, players need to make their way through entire waves of cars coming down. Making contact with a single car will cause the game to be over, requiring you to restart the game. Another thing that will be spawning with the cars are coins, which are required to be collected by the player in order to advance to the next level. A set amount of coins are required to advance to the next level. Picking up the coins is as simple as colliding with the coin itself.



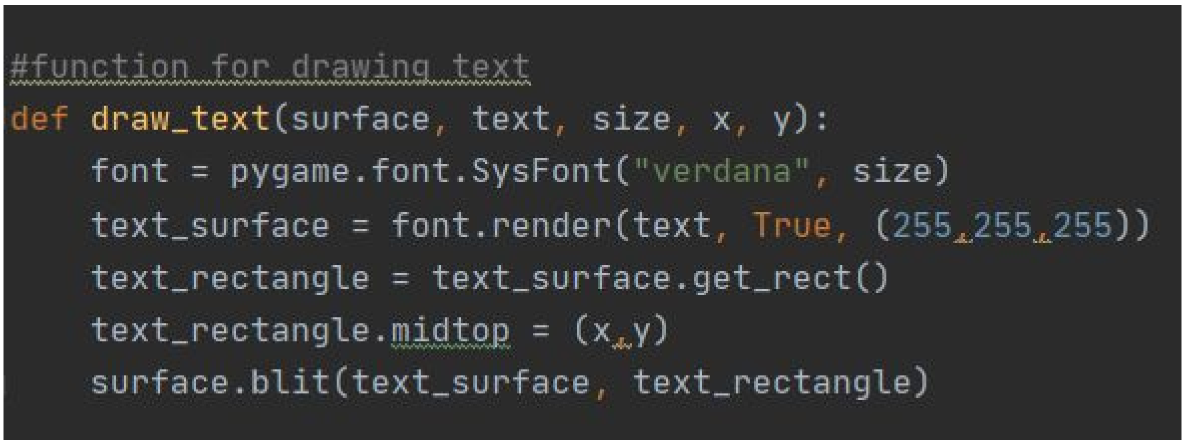
After a certain amount of levels have been cleared, the game will end, telling the players that they have successfully escaped the barrage of incoming traffic. The game is now over.

IV. How the program works



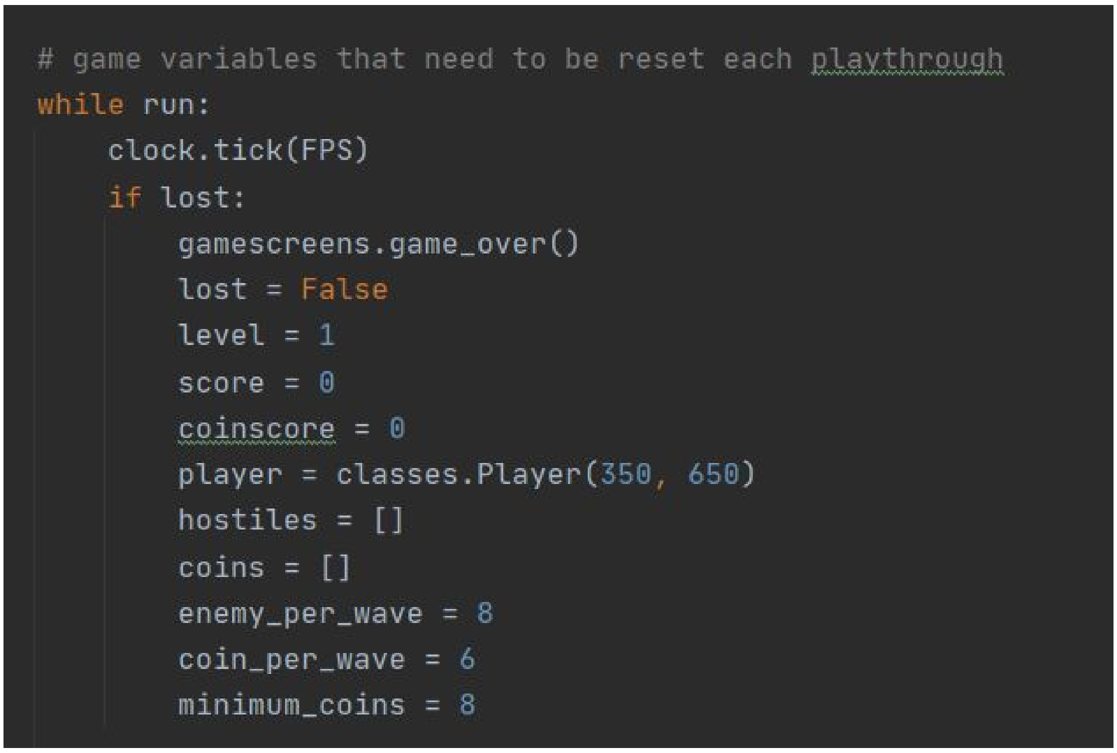
```python
import pygame
import os

pygame.mixer.init(frequency=22050, size=-16, channels=2, buffer=4096)

#loading images
red_car = pygame.image.load(os.path.join("assets", "red_car.png"))
green_car = pygame.image.load(os.path.join("assets", "green_car.png"))
yellow_car = pygame.image.load(os.path.join("assets", "yellow_car.png"))

gold_coin = pygame.transform.scale(pygame.image.load(os.path.join("assets", "gold_coin.png")), (32, 32))
silver_coin = pygame.transform.scale(pygame.image.load(os.path.join("assets", "silver_coin.png")), (32, 32))
bronze_coin = pygame.transform.scale(pygame.image.load(os.path.join("assets", "bronze_coin.png")), (32, 32))

main_car = pygame.transform.scale(pygame.image.load(os.path.join("assets", "racer.png")), (48, 96))

background = pygame.transform.scale(pygame.image.load(os.path.join("assets", "road.png")), (700, 800))

#loading sound effects
coin_pickup = pygame.mixer.Sound(os.path.join("assets", "smw_coin.wav"))
coin_pickup.set_volume(0.5)

car_crash = pygame.mixer.Sound(os.path.join("assets", "carcrash.wav"))
car_crash.set_volume(0.5)

game_music = pygame.mixer.Sound(os.path.join("assets", "ikenaiborderline8bit.mp3"))
game_music.set_volume(0.1)

win_music = pygame.mixer.Sound(os.path.join("assets", "win_music.wav"))
win_music.set_volume(0.1)
```

My program first loads all assets that I use. All images and music are loaded in first and the volume for each sfx/music is set to a certain volume. The modules my program uses other than PyGame are os, time, and random.

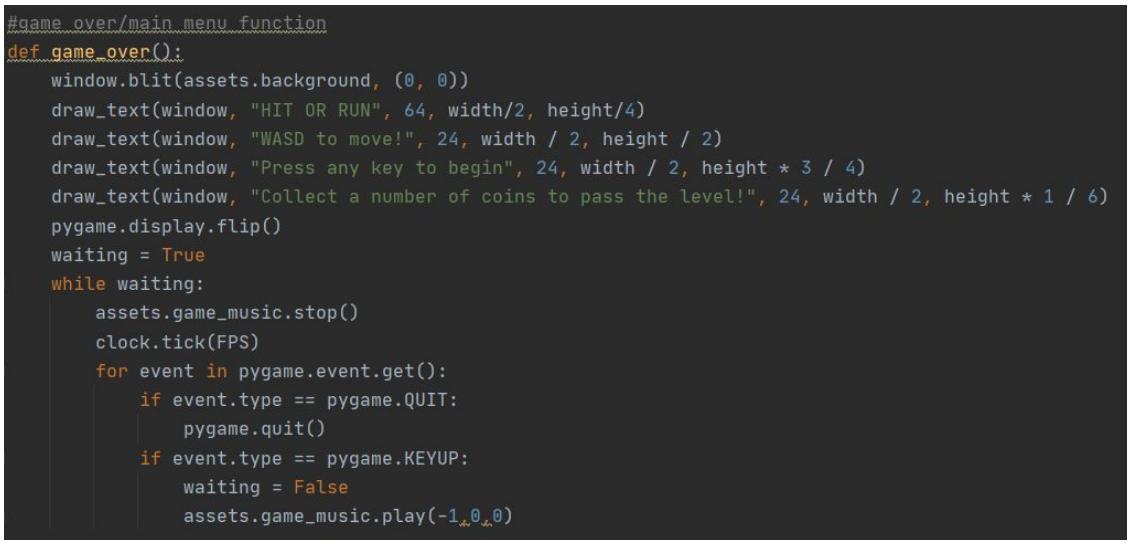


```python
#function for drawing text
def draw_text(surface, text, size, x, y):
    font = pygame.font.SysFont("verdana", size)
    text_surface = font.render(text, True, (255,255,255))
    text_rectangle = text_surface.get_rect()
    text_rectangle.midtop = (x,y)
    surface.blit(text_surface, text_rectangle)
```

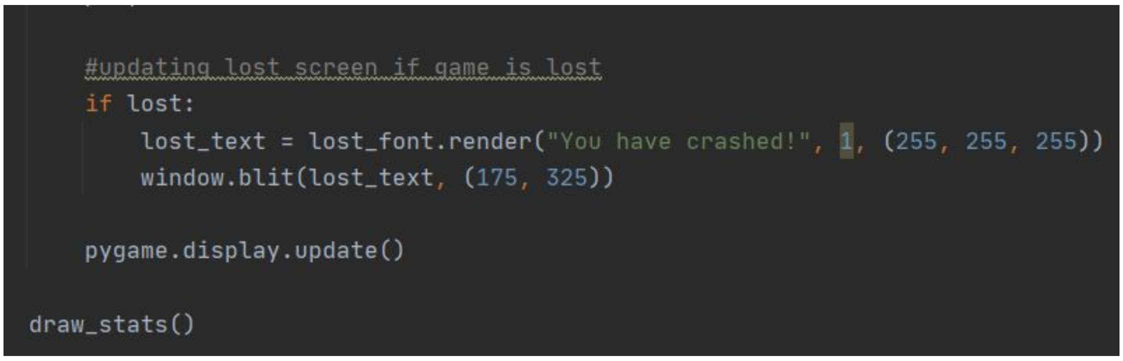
I also have a function that allows me to draw text on screen easier, which I used in the main menu and game over. I mostly used the font “verdana” for all my texts. Text_rectangle and .midtop are used so can be easier to center the text boxes in the middle of the screen.



```python
# game variables that need to be reset each playthrough
while run:
    clock.tick(FPS)
    if lost:
        gamescreens.game_over()
        lost = False
        level = 1
        score = 0
        coinscore = 0
        player = classes.Player(350, 650)
        hostiles = []
        coins = []
        enemy_per_wave = 8
        coin_per_wave = 6
        minimum_coins = 8
```

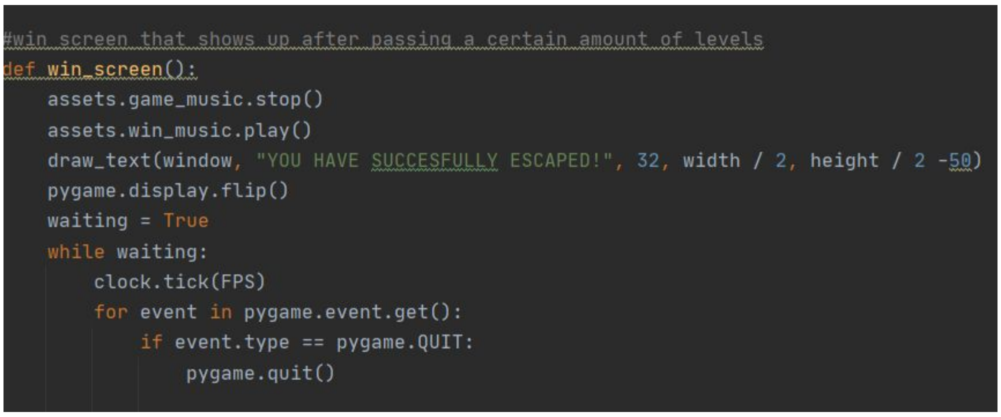


```python
#game over/main menu function
def game_over():
    window.blit(assets.background, (0, 0))
    draw_text(window, "HIT OR RUN", 64, width/2, height/4)
    draw_text(window, "WASD to move!", 24, width / 2, height / 2)
    draw_text(window, "Press any key to begin", 24, width / 2, height * 3 / 4)
    draw_text(window, "Collect a number of coins to pass the level!", 24, width / 2, height * 1 / 6)
    pygame.display.flip()
    waiting = True
    while waiting:
        assets.game_music.stop()
        clock.tick(FPS)
        for event in pygame.event.get():
            if event.type == pygame.QUIT:
                pygame.quit()
            if event.type == pygame.KEYUP:
                waiting = False
                assets.game_music.play(-1,0,0)
```

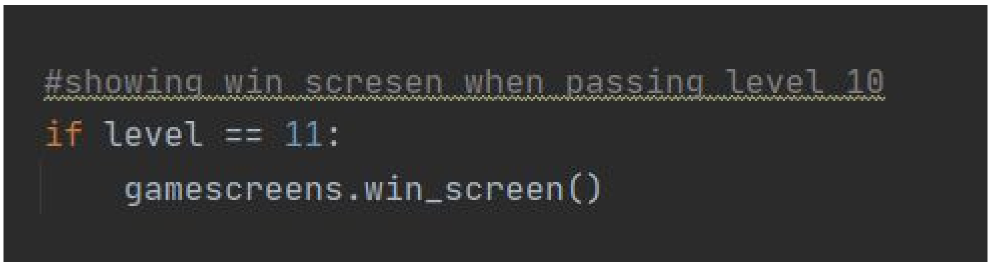


```python
        #updating lost screen if game is lost
        if lost:
            lost_text = lost_font.render("You have crashed!", 1, (255, 255, 255))
            window.blit(lost_text, (175, 325))

        pygame.display.update()

    draw_stats()
```

After all assets are loaded and modules imported the program runs through the game_over() function, which acts as both a main menu and a game over screen for the game. Text is being drawn with different sizes, and coordinates which have previously been centered makes it easier to place in the screen. While in the main menu, the game is put into a waiting state, and when any key is inputted, the game loop will start and the game can be played. When the player crashes, or loses, he will be put in the lost state which will trigger the game_over() function and reset everything and go back to the waiting state again, which is why this function is used as both a main menu and game over screen.



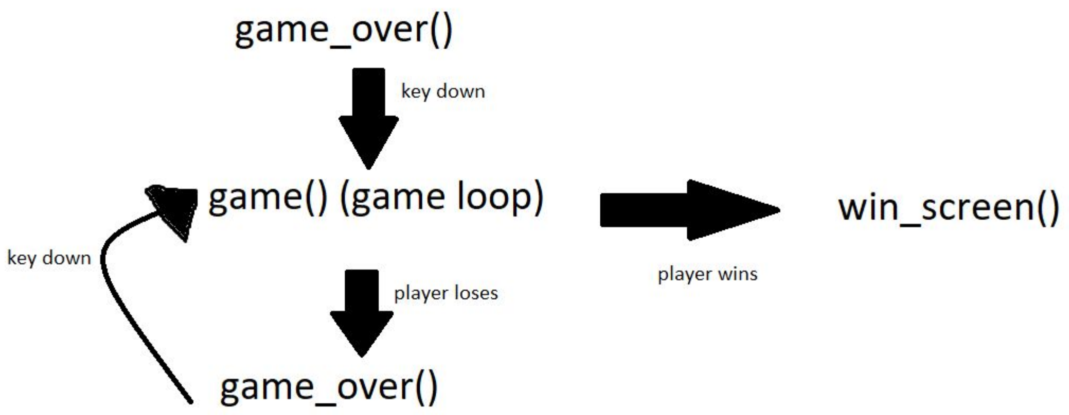
```python
#win screen that shows up after passing a certain amount of levels
def win_screen():
    assets.game_music.stop()
    assets.win_music.play()
    draw_text(window, "YOU HAVE SUCCESFULLY ESCAPED!", 32, width / 2, height / 2 -50)
    pygame.display.flip()
    waiting = True
    while waiting:
        clock.tick(FPS)
        for event in pygame.event.get():
            if event.type == pygame.QUIT:
                pygame.quit()
```



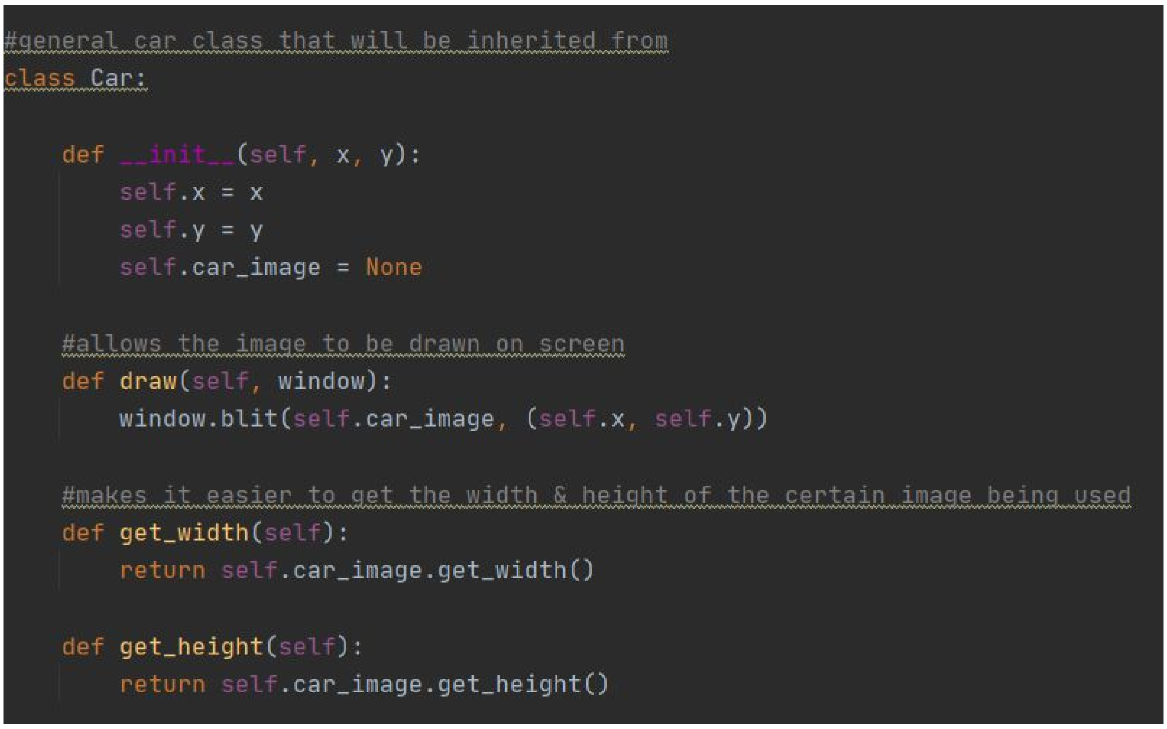
```python
#showing win scresen when passing level 10
if level == 11:
    gamescreens.win_screen()
```

The other state the game can be put in is the win screen, where after the player reaches a certain level, they will be shown the win screen, indicating that they have won. Text is displayed showing that the player has escaped, meaning that they won, the in-game music will be stopped and the winning music will be played. The player can then quit the game after they have finished.

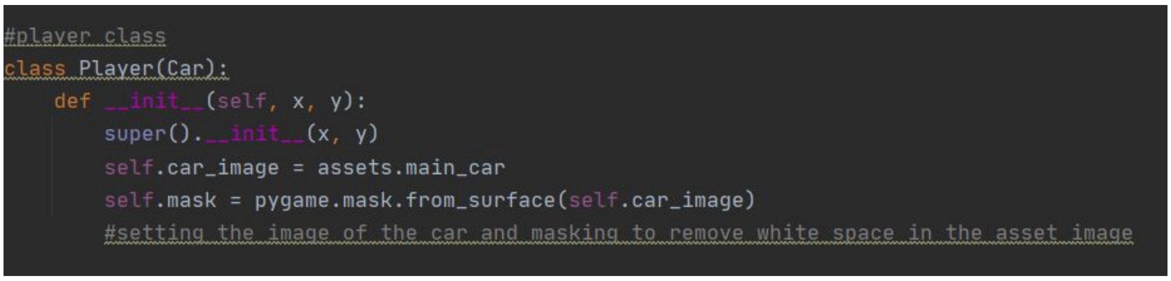
A diagram showing how this all works:



V. Code explanation

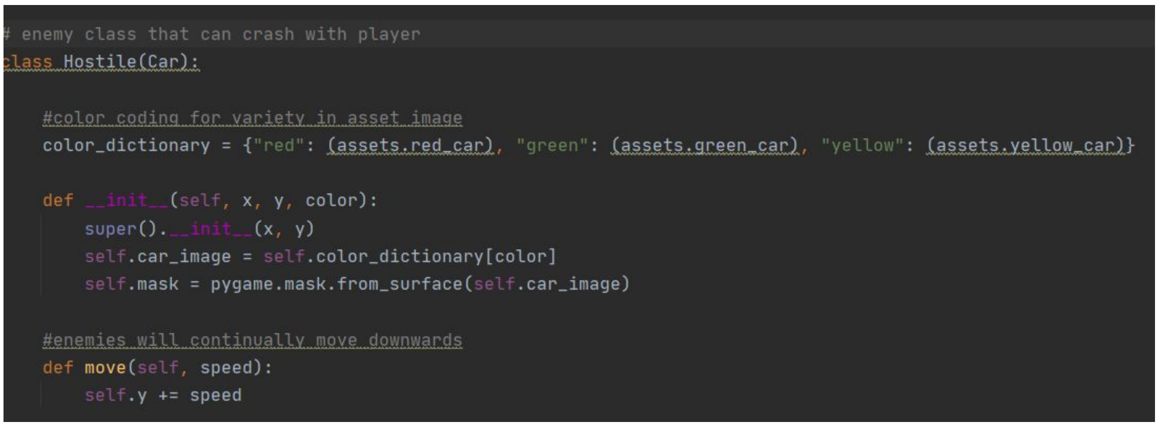


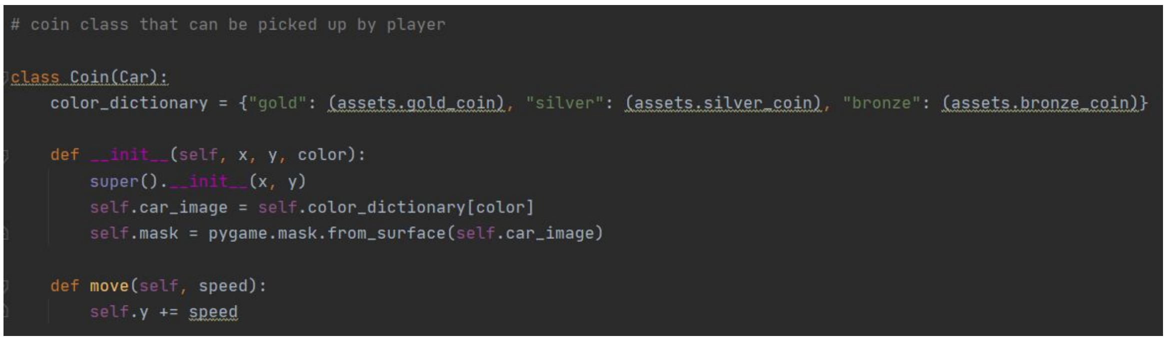
```python
#general car class that will be inherited from
class Car:

    def __init__(self, x, y):
        self.x = x
        self.y = y
        self.car_image = None

    #allows the image to be drawn on screen
    def draw(self, window):
        window.blit(self.car_image, (self.x, self.y))

    #makes it easier to get the width & height of the certain image being used
    def get_width(self):
        return self.car_image.get_width()

    def get_height(self):
        return self.car_image.get_height()
```



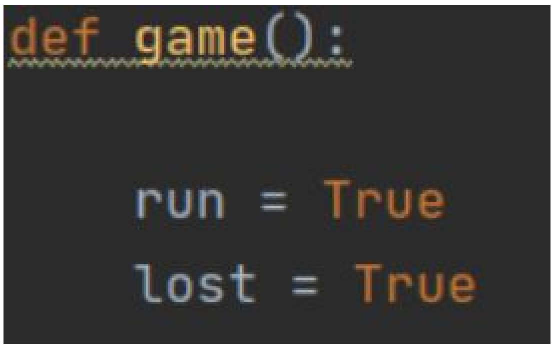
```python
#player class
class Player(Car):
    def __init__(self, x, y):
        super().__init__(x, y)
        self.car_image = assets.main_car
        self.mask = pygame.mask.from_surface(self.car_image)
        #setting the image of the car and masking to remove white space in the asset image
```

The sprites in this game use a general class, which will be inherited from by the other subclasses. The Car class is a class that is used for the sprites in this game, mainly being the player's vehicle, enemy vehicles, and the coins. Their coordinates can be set as well as their image which can be loaded with the assets that have been loaded before. They have a draw function, where the assets that have been loaded can be drawn using .blit on the sprite's coordinates. Get_width and get_height are used to find the dimensions of the sprite, making it easier for the collision function which will be explained later on. The player subclass uses pygame.mask so that the collision function makes it detect collision on pixel perfect, meaning that the sprites will not collide with the white space in the assets' images.

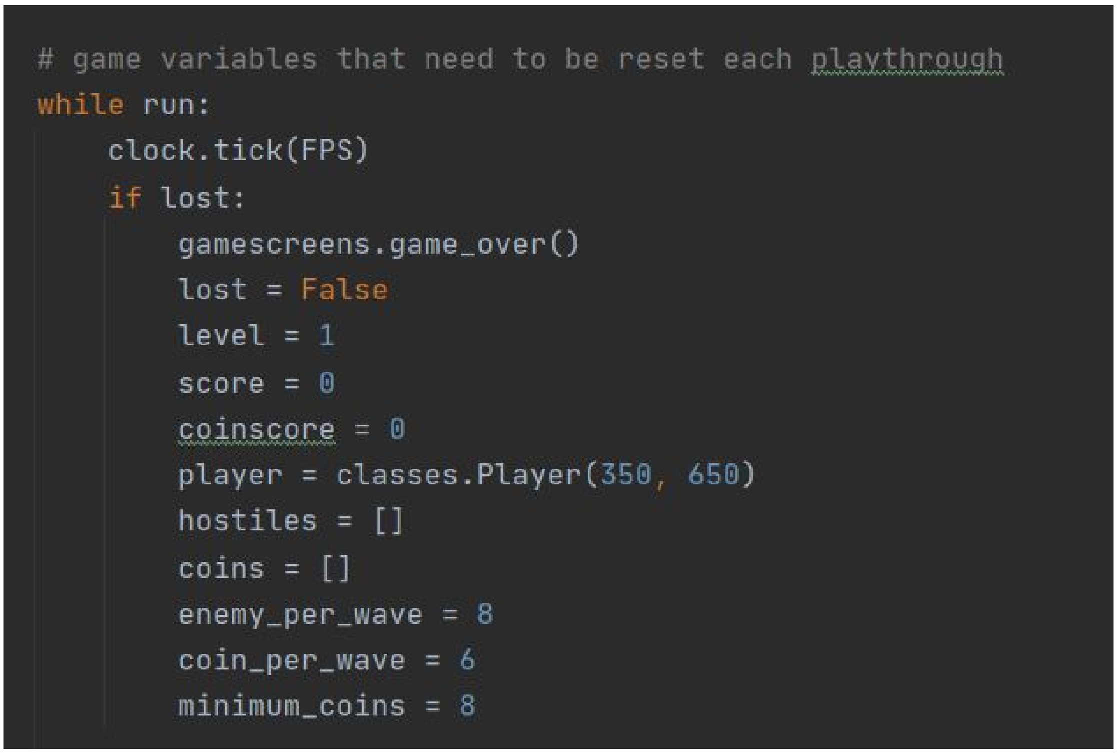

```
# enemy class that can crash with player
class Hostile(Car):

    #color coding for variety in asset image
    color_dictionary = {"red": (assets.red_car), "green": (assets.green_car), "yellow": (assets.yellow_car)}

    def __init__(self, x, y, color):
        super().__init__(x, y)
        self.car_image = self.color_dictionary[color]
        self.mask = pygame.mask.from_surface(self.car_image)

    #enemies will continually move downwards
    def move(self, speed):
        self.y += speed
```


```
# coin class that can be picked up by player

class Coin(Car):
    color_dictionary = {"gold": (assets.gold_coin), "silver": (assets.silver_coin), "bronze": (assets.bronze_coin)}

    def __init__(self, x, y, color):
        super().__init__(x, y)
        self.car_image = self.color_dictionary[color]
        self.mask = pygame.mask.from_surface(self.car_image)

    def move(self, speed):
        self.y += speed
```

Next up is the hostiles and coin subclass. 3 separate types of images are prepared for each subclass, to add more variety and color to the game, although they do not serve any purpose other than decorational. A dictionary is used to colour code each asset and the image will be assigned to it, and will be randomized which I will explain when I explain the spawning behavior. These subclasses also have a move function, which moves them down (since it's + to the y-axis) according to the speed.

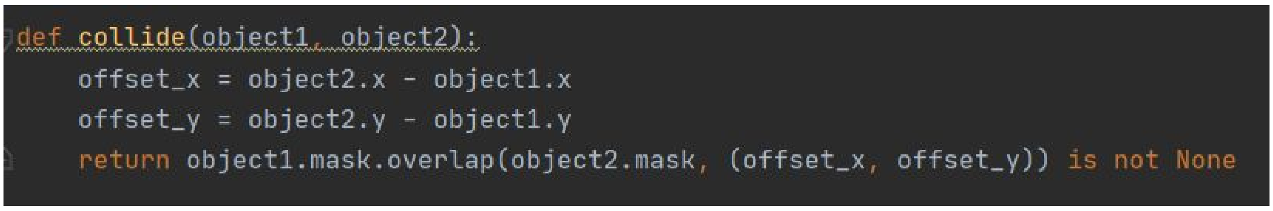


```python
def game():

    run = True
    lost = True
```



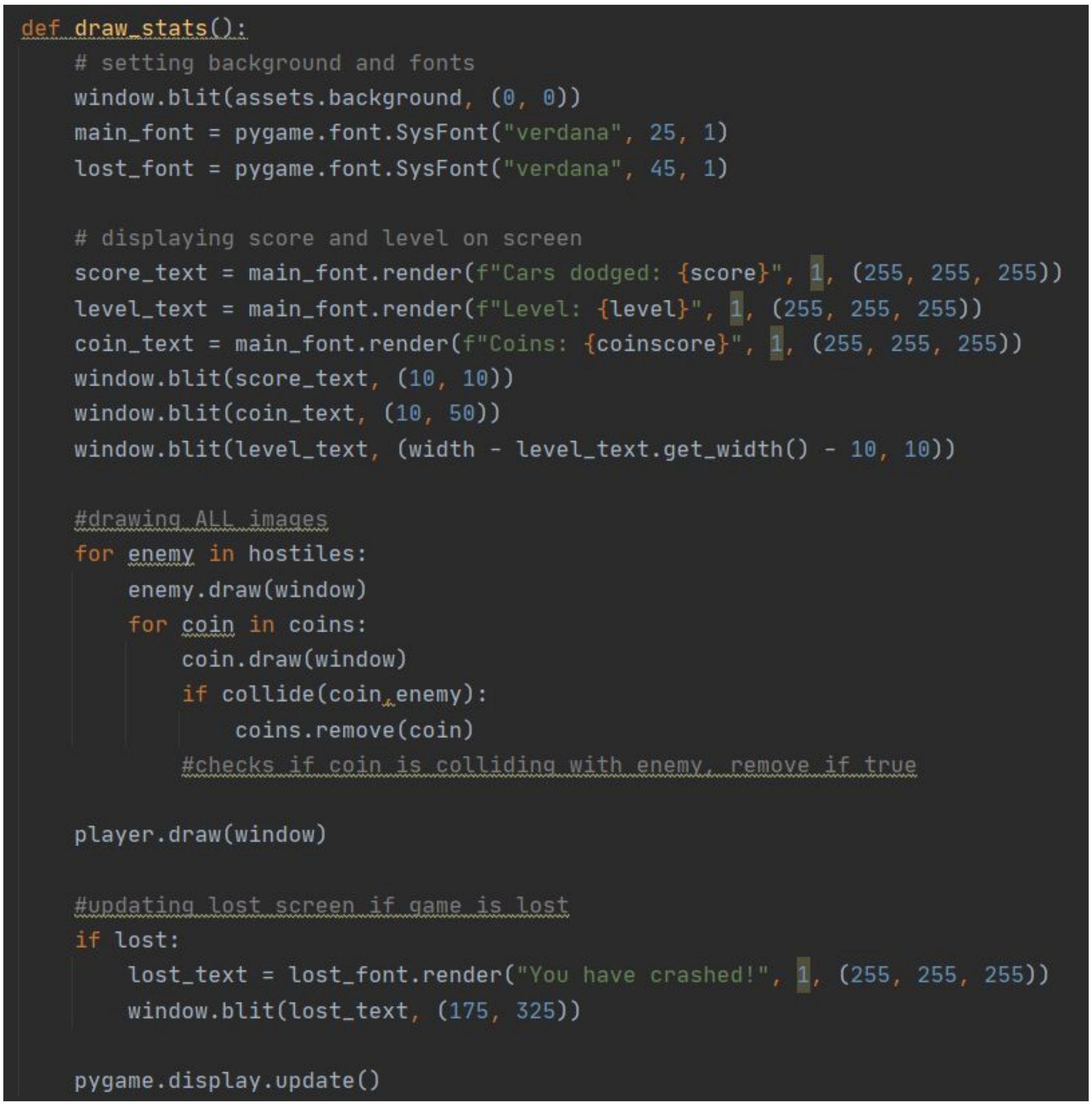
```python
    # game variables that need to be reset each playthrough
    while run:
        clock.tick(FPS)
        if lost:
            gamescreens.game_over()
            lost = False
            level = 1
            score = 0
            coinscore = 0
            player = classes.Player(350, 650)
            hostiles = []
            coins = []
            enemy_per_wave = 8
            coin_per_wave = 6
            minimum_coins = 8
```

This is the game loop that triggers after the game_over() function which I have explained previously. Each scoring variable such as levels, score and coin score is reset and all the classes are created, where there is a player, hostiles, and enemies. There is also a set variable for the amount of enemies and coins which spawns in a wave, and a variable for the minimum amount of coins needed to be collected by the player to advance to the next level.



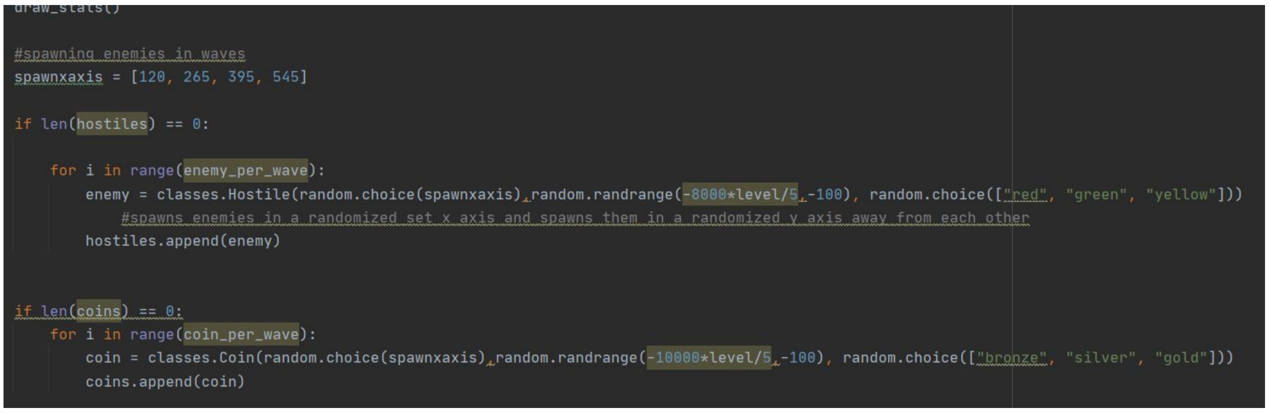
```python
def collide(object1, object2):
    offset_x = object2.x - object1.x
    offset_y = object2.y - object1.y
    return object1.mask.overlap(object2.mask, (offset_x, offset_y)) is not None
```

For the collision function, I used a calculation where the offset of both sprites are used, which are both coordinates subtracted. Since sprite coordinates start at the top left, the offset is used instead when returning if the objects are overlapping each other, making the collision detection very pixel perfect.



```python
def draw_stats():
    # setting background and fonts
    window.blit(assets.background, (0, 0))
    main_font = pygame.font.SysFont("verdana", 25, 1)
    lost_font = pygame.font.SysFont("verdana", 45, 1)

    # displaying score and level on screen
    score_text = main_font.render(f"Cars dodged: {score}", 1, (255, 255, 255))
    level_text = main_font.render(f"Level: {level}", 1, (255, 255, 255))
    coin_text = main_font.render(f"Coins: {coinscore}", 1, (255, 255, 255))
    window.blit(score_text, (10, 10))
    window.blit(coin_text, (10, 50))
    window.blit(level_text, (width - level_text.get_width() - 10, 10))

    #drawing ALL images
    for enemy in hostiles:
        enemy.draw(window)
        for coin in coins:
            coin.draw(window)
            if collide(coin,enemy):
                coins.remove(coin)
            #checks if coin is colliding with enemy, remove if true

    player.draw(window)

    #updating lost screen if game is lost
    if lost:
        lost_text = lost_font.render("You have crashed!", 1, (255, 255, 255))
        window.blit(lost_text, (175, 325))

    pygame.display.update()
```

The next function that occurs in the game loop is draw_stats, which constantly updates all the player, hostile, and coin sprites as well as background image using display.update() according to their coordinates wherever they are on their screen. It also updates the text showing level, the amount of cars which the players have evaded, and how much coins the player has collected. This function also checks to see if the player has entered the lost state as explained previously, and will update accordingly. For updating enemy and coin sprites, it also checks to see if both have collided with each other, and removes both sprites if they are colliding.

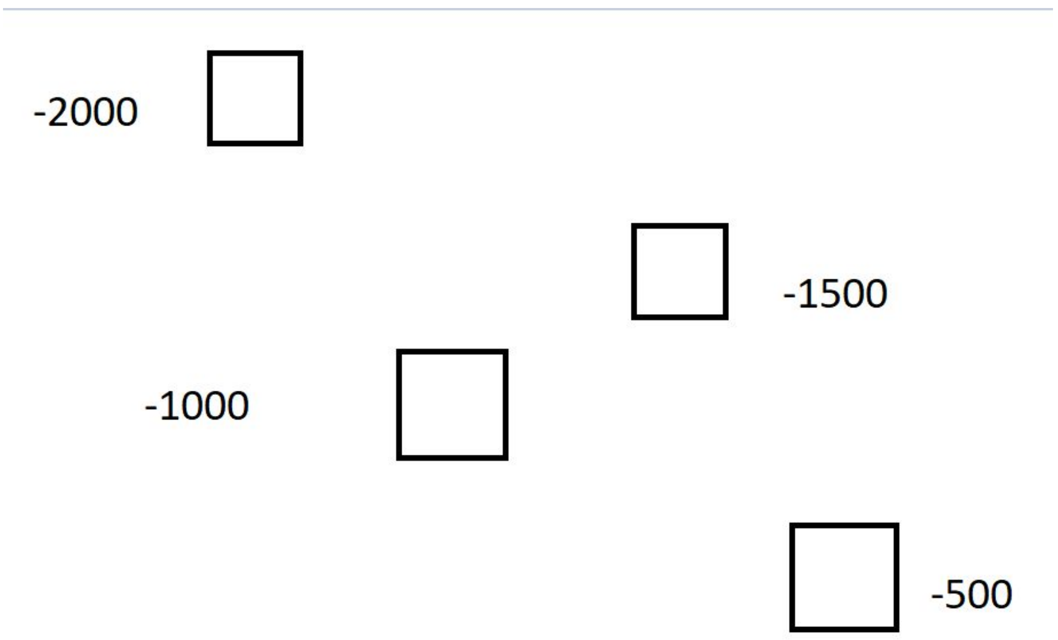


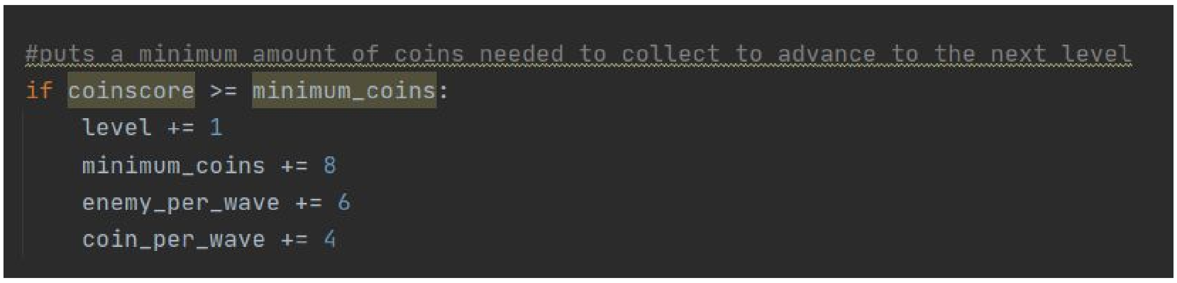
```
draw_stats()

#spawning enemies in waves
spawnxaxis = [120, 265, 395, 545]

if len(hostiles) == 0:

    for i in range(enemy_per_wave):
        enemy = classes.Hostile(random.choice(spawnxaxis),random.randrange(-8000*level/5,-100), random.choice(["red", "green", "yellow"]))
            #spawns enemies in a randomized set x axis and spawns them in a randomized y axis away from each other
        hostiles.append(enemy)


if len(coins) == 0:
    for i in range(coin_per_wave):
        coin = classes.Coin(random.choice(spawnxaxis),random.randrange(-10000*level/5,-100), random.choice(["bronze", "silver", "gold"]))
        coins.append(coin)
```

Next is the spawning behavior for the enemies and coins. An if loop is used to check if there are currently any hostile cars on the screen, and will pick a random colour from the dictionary set in the subclass before, and spawns a number of enemies using append() based on the variable that we set before. This same behavior is used for the coins, using different values.

For the coordinates where they spawn, all of them have a set x-coordinate which is the spawnxaxis list shown, where each coordinate is the lane that they will spawn in based on the background image used for the game. For the y-coordinate, the value is a random number which represents how far away the sprite spawns on top of the screen, which gives the illusion that the sprites spawns one by one instead of in a wave. The value of this number gets lower as the level increases, meaning the sprites will spawn at a faster rate. There is also a known bug with this spawning behavior, which I will explain in the next section of the report.

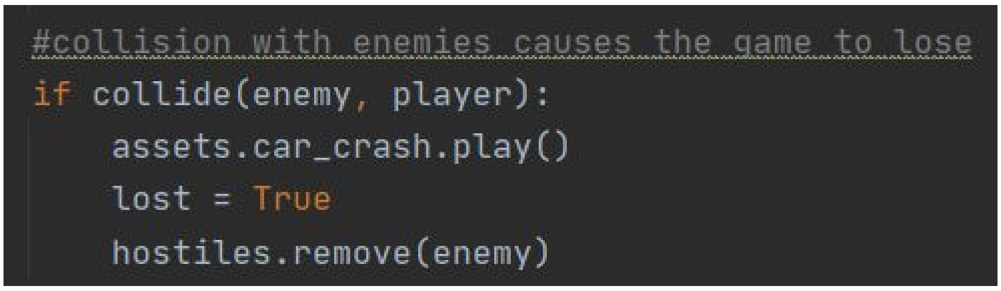
A small diagram that shows how the sprites spawn on the y-axis based on the randomized numbers (numbers represent y-coordinates):



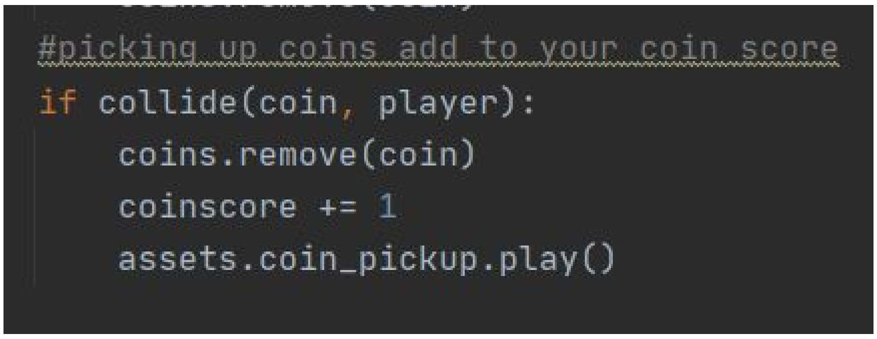


```python
#puts a minimum amount of coins needed to collect to advance to the next level
if coinscore >= minimum_coins:
    level += 1
    minimum_coins += 8
    enemy_per_wave += 6
    coin_per_wave += 4
```

The loop also checks to see if the minimum coin variable that we set before has been met. If the player has collected enough coins, they will advance a level, the variable for number of sprites spawn will also increase, meaning more sprites will spawn per wave, as well as the amount of coins needed to advance to the next level increasing.

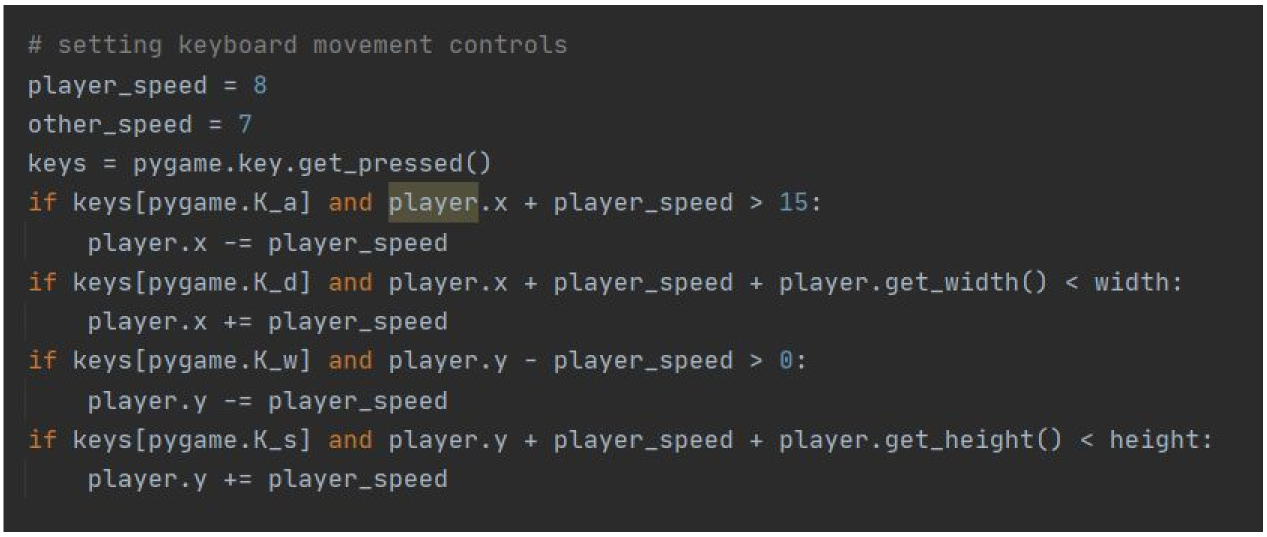


```python
#collision with enemies causes the game to lose
if collide(enemy, player):
    assets.car_crash.play()
    lost = True
    hostiles.remove(enemy)
```



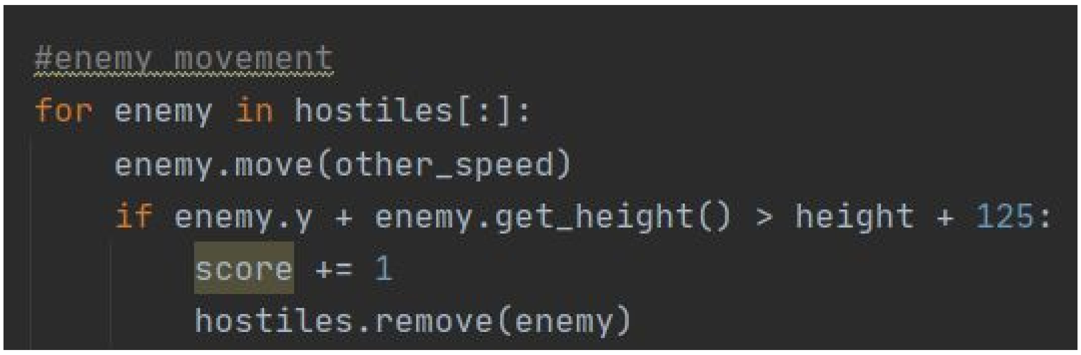
```python
#picking up coins add to your coin score
if collide(coin, player):
    coins.remove(coin)
    coinscore += 1
    assets.coin_pickup.play()
```

For collision with any sprite, the game checks to see if the player has either collided with an enemy or a coin. If the player collides with an enemy, the SFX will be played and the lost state will be True, causing the game_over() function to take over. If the player collides with a coin, the coin will be removed, the player's coin score will update on the screen, and SFX will play.

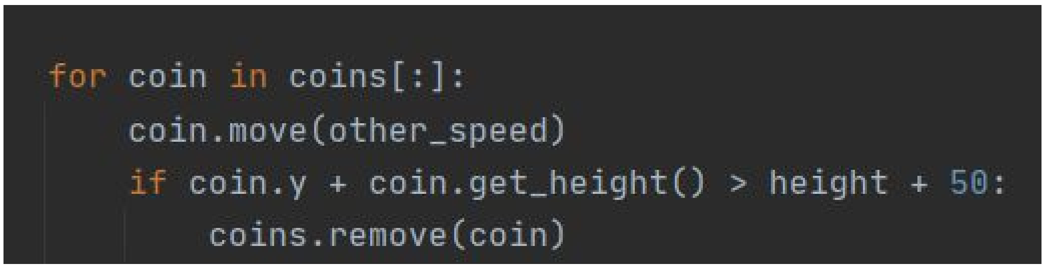


```
# setting keyboard movement controls
player_speed = 8
other_speed = 7
keys = pygame.key.get_pressed()
if keys[pygame.K_a] and player.x + player_speed > 15:
    player.x -= player_speed
if keys[pygame.K_d] and player.x + player_speed + player.get_width() < width:
    player.x += player_speed
if keys[pygame.K_w] and player.y - player_speed > 0:
    player.y -= player_speed
if keys[pygame.K_s] and player.y + player_speed + player.get_height() < height:
    player.y += player_speed
```

Next is the movement section. The player has a set speed variable and enemies/coins have a different speed. Player movement is just as simple as moving the players based on the direction and what key is pressed, based on the player speed set. It also checks to see if the player has hit the set boundary in the screen using the width and height of the screen, where the player's sprite will not move if he has reached that boundary, preventing them from going off screen.



```
#enemy movement
for enemy in hostiles[:]:
    enemy.move(other_speed)
    if enemy.y + enemy.get_height() > height + 125:
        score += 1
        hostiles.remove(enemy)
```



```
for coin in coins[:]:
    coin.move(other_speed)
    if coin.y + coin.get_height() > height + 50:
        coins.remove(coin)
```

Lastly, there is the loop to check the movement of enemies and coins. These sprites will constantly move downwards based on their speed. I originally intended for the enemies to have increasing speed, but that was too difficult so I took it out. The loop also checks to see if the enemy or coin sprites have reached a certain height downwards off screen, and removes them when they reach that point.

VI. Lessons that Have Been Learned

I have learnt a lot of valuable lessons throughout this project. The most obvious being that I've learned the basics on using the PyGame library module, where I learned how to draw text and spawn images on the screen, add music, player movement responding to keyboard inputs, and a lot of basics in game-making that i thought was going to be simple to implement, but turned out to be very complicated to code.

Sadly, it turns out that my game had a bug that is very difficult to fix (and I am not skilled enough to find a workaround) which is unfortunately due to the nature of how my classes are coded. In some instances during the game, both enemy cars and coins have a chance of spawning on top of each other, coin on coin or car on car, and will go down together. The reasoning for this being that since they have a set spawn y-coordinate, there is a chance that it can spawn while another coin/car is passing by in that y-coordinate, causing them to go on top of each other and look very buggy. It's not fundamentally game-breaking, but still an unsightly bug. I've fixed half of the problem, when a coin spawns on top of the car and the other way around, it will be removed. But, I do not know how to fix when a car/coin spawns on top of another of the same kind. If my collision function is used to check a collision of an object with itself, then that object will always constantly be colliding with itself. I look to find a fix to this bug one day.

VII. Project link

https://github.com/rafianathallah/PDMFinalProject