DECK-COMMANDER

CHAPTER 1: Introduction

1.1 Inspiration

Our first inspiration for Deck Commander came from the 1984 Duck Hunt game. The idea of making a first-person shooter game was exciting but a bit of a challenge. The animations and the movement of the ducks, in addition to the random appearances of the dog, were over the top for two beginner programmers. That's why the idea dropped very quickly.

Image 1.1: Duck Hunt (NES) by gamefragger.com

The final inspiration for Deck Commander comes from Space Invaders, which was published in 1978. In this version, there is a modern approach to the classic game.

1.1.1 More about Space Invaders

Space Invaders is a 1978 shoot'em up arcade video game developed and released by Taito in Japan, and licensed to Midway Manufacturing for overseas distribution. Commonly considered one of the most influential video games of all time, Space Invaders was the first fixed shooter and set the template for the genre. The goal is to defeat wave after wave of descending aliens with a horizontally moving laser cannon to earn as many points as possible.

1.1.2 The difference between Deck Commander and Space Invaders

- Both games have many waves of attacks and multiplayer modes
- In Deck Commander, the player has a database to compete with.
- Contrary to the classic, where the aliens had a horizontal movement, Deck Commander's enemies come from different directions.
- There are no obstacles to protect the spaceship and the player has to counter the attacks.
- There is no mystery ship with extra points.
- Both games belong to the fixed shooter category. That's because the player is restricted to a single-axis movement.
- In Deck Commander the aliens don't shoot lasers.

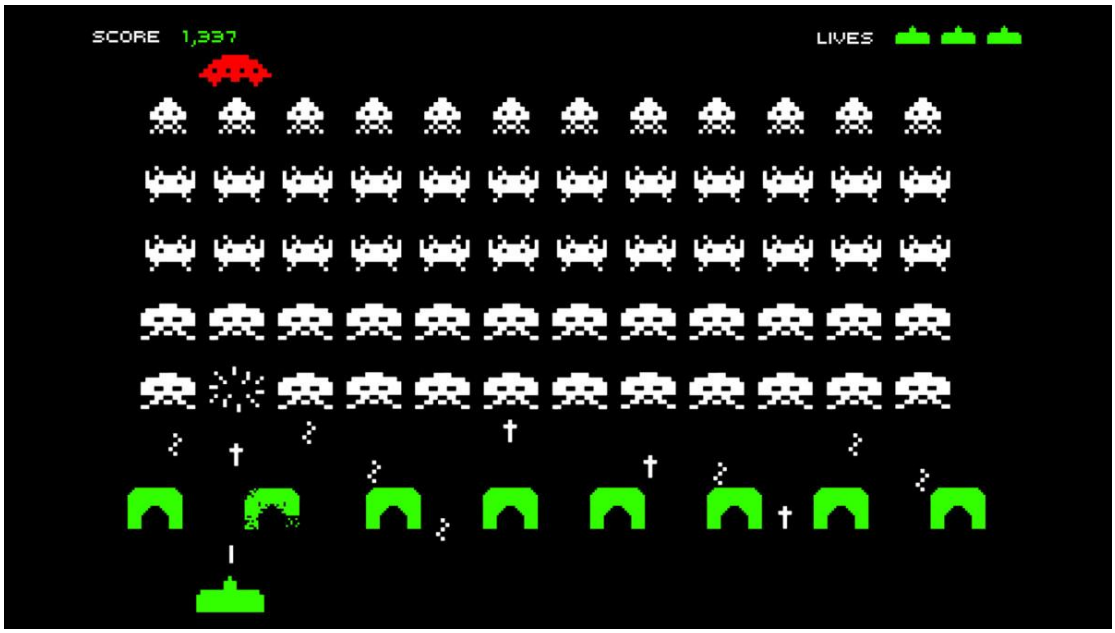


Image 1.2: The original Space Invaders by levelskip.com

1.2 Story

We allied as one when we discovered an alien civilization in 2150. Unfortunately, after never-ending negotiations, the Barh clan refused a peace offering with our people. Most of our spaceships returned to our base. The final destination was Earth. Before departure, some alien ships started the attack. Our finest pilots gathered to protect the base.

"This is the captain speaking. We are under attack. I repeat, the base is under attack."

Image 1.3: Deck Commander Single Player mode

Mission Objective

Try to buy some time by attacking the enemies. The main objective is to get a high score.

1.3 Gameplay Mechanics

- **HEALTH BAR**

The player has a health bar that shows the remaining health.

- **AMMO BAR**

  The player has an ammo bar that shows the remaining ammunition.

- **AIM/SHOOT**

By pressing the “Left Mouse” key the spaceship shoots lasers. That changes in the multiplayer mode where we use the “W” key.

- **DIRECTION OF THE ENEMIES**

The enemies appear from different directions. As the game progresses, their speed increases.

- **RELOAD**

The ammo reloads by pressing the “S” button.

- **PLAYER'S MOVEMENT**

The player moves horizontally by pressing the "A" and "D" buttons to move left and right.

- **MULTIPLAYER**

The second player uses the arrows to move and attack.

- **MAIN MENU**

The main menu includes the single-player and multiplayer modes as well as the player's previous scores in a database.

- **ROUND**

Informs the player of the current round.

- **GAME OVER**

Informs the player of the final score.

- **USERNAME**

The player can save his/her current score and share it with friends.

1.4 Graphics

Pixel art and simple graphics were used in this project. This decision was made to keep the original vibe of an 8-bit and 16-bit era. The player and enemy ships, the planets, and the base were made in KRITA.

CHAPTER 2: CODE

2.1 Classes Files

When we started thinking about the way this game was going to be coded, we separated the game into seven different classes, each class representing an element of the game. So in the final build, Deck Commander uses the following classes:

1. The Player class is used to declare a player within the game.
2. The Hitbox class represents the item that is being defended by the player.
3. The Projectile class is used to code the projectiles that are shot by the player's spaceship.
4. The Gamemode class stores various elements of the game.
5. The Scope class is only visible when the player is repairing the ship.
6. The Target class is used to manipulate the behavior of the enemy targets.
7. The Target Spawner class creates a spawn point for the enemy targets.

Each of the classes referenced above is stored in seven different Python files, shown in the following image:

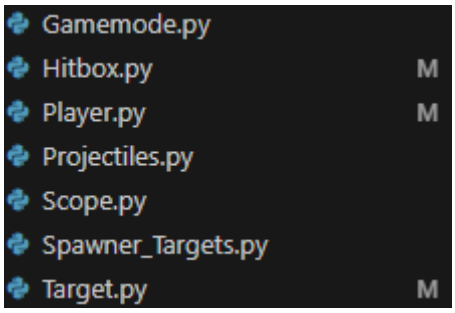


Image 2.1: Class files

2.1.1 Gamemode class

As previously mentioned, the Gamemode class is responsible for storing variables used to manipulate various elements of the game, such as the target speed. The following variables are set during the initialization of this class:

- Title: This variable is set in case we want to develop more game modes for this game. The basic game mode is called "Defend the Ship". As the title suggests, the player must defend the cargo ship from an imminent attack.

- Round: This variable represents the current round that the cargo ship has survived.

- Round_change: This variable is used to increase the damage, spawn speed, and movement speed of the enemies (Targets). For example, in the current game mode developed, every three rounds, the targets get buffed in these particular attributes.

- Game_round_change_score: This variable is responsible for increasing the current round in the game mode.

- Spawn_time_inc: This variable represents the amount by which the spawn time of the enemies is decreased.

- Damage_inc: This variable represents the amount of damage buff given to the enemies.

- Speed_inc: This variable represents the increase in the speed of the enemies.

- T_spawn_time: This variable represents the current time it takes for the spawner to spawn the enemies.

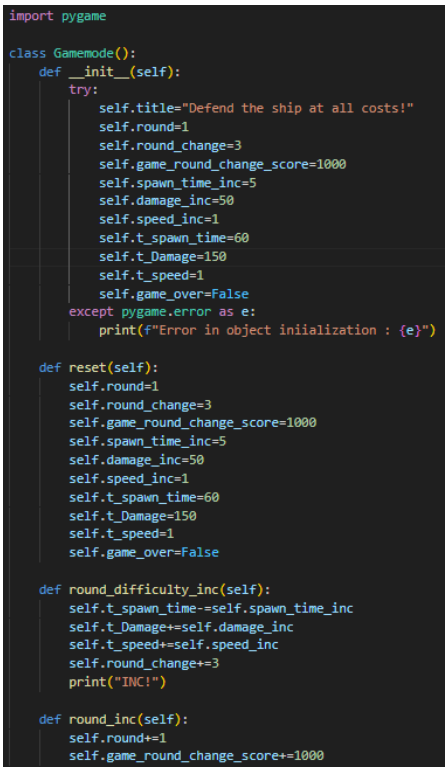
- T_damage: This variable represents the current damage that the enemies deal.

- T_speed: This variable represents the movement speed of the enemies.

- Game_over: This variable is a boolean that checks if the game is over.

These variables can be manipulated to adjust the game difficulty and provide a challenging gameplay experience.

In addition to the variables mentioned earlier, we also need to consider the functions of the gamemode class. Apart from the initialization function (_init_), there are two other important functions: “reset” and round_difficulty_inc. The reset function is used to reset the class variables whenever required, such as at the start of each game. The round_difficulty_inc function increases the game's difficulty every time the player reaches a round where the target must be buffed. In this particular game, the target is buffed every three rounds. Lastly, there is the “round_inc” function, which enhances some of the gamemode's attributes after each round increase.



```
import pygame

class Gamemode():
    def __init__(self):
        try:
            self.title="Defend the ship at all costs!"
            self.round=1
            self.round_change=3
            self.game_round_change_score=1000
            self.spawn_time_inc=5
            self.damage_inc=50
            self.speed_inc=1
            self.t_spawn_time=60
            self.t_Damage=150
            self.t_speed=1
            self.game_over=False
        except pygame.error as e:
            print(f"Error in object iniialization : {e}")

    def reset(self):
        self.round=1
        self.round_change=3
        self.game_round_change_score=1000
        self.spawn_time_inc=5
        self.damage_inc=50
        self.speed_inc=1
        self.t_spawn_time=60
        self.t_Damage=150
        self.t_speed=1
        self.game_over=False

    def round_difficulty_inc(self):
        self.t_spawn_time-=self.spawn_time_inc
        self.t_Damage+=self.damage_inc
        self.t_speed+=self.speed_inc
        self.round_change+=3
        print("INC!")

    def round_inc(self):
        self.round+=1
        self.game_round_change_score+=1000
```

Image 2.2: Gamemode Class

2.1.2 Player Class

This class is the most complex in the project, using many variables and attributes. The playerID variable identifies if the player is controlled by player1 or player2 (multiplayer). The image and body_image variables load the players’ sprite in the game.

During the start of the game, the player's starting position needs to be established as a default value. To achieve this, the position of the player is set

with the help of position, posx, and posy values. It is important to note that posy remains constant as the player can only be moved horizontally.

*The convert_alpha() method is used to improve the performance of the game. This method is used to convert surfaces to the same pixel format as used by the screen by keeping the alpha channel.

We are using the 'rect' variable to create a rectangle for each player, which makes it easy to manipulate the player's body. This variable is set using the 'get_rect' method provided by Pygame. With this method, Pygame automatically generates the rectangular shape of the given image.

To begin with, it's important to set the default values for the player's position, score, max ammo, and ammo variables. The max ammo variable determines the maximum amount of ammo each player can reload. Additionally, the speed variable and the score increase variable should be defined. The score increase variable determines how much the player's score increases every time they hit a target.

```
import pygame
import math

from pygame.locals import *
from Var.Constants import *
from Assets.Sound_effects import reload_sound


class Player(pygame.sprite.Sprite):
    def __init__(self,playerID):
        super().__init__()
        try:
            self.playerID=playerID
            if (self.playerID==1):
                self.image=pygame.image.load(LINK_ASSETS_SPACESHIP)
                self.body_image=pygame.image.load(LINK_ASSETS_SPACESHIP)
                self.posx=DISPLAY_WIDTH/2 + 50
            elif(self.playerID==2):
                self.image=pygame.image.load(LINK_ASSETS_SPACESHIP2)
                self.body_image=pygame.image.load(LINK_ASSETS_SPACESHIP2)
                self.posx=DISPLAY_WIDTH/2 - 50
            self.image=self.image.convert_alpha()
            self.body_image=self.body_image.convert_alpha()

        except pygame.error as e:
            print(f"Error in object iniialization : {e}")
        #self.image=pygame.transform.scale(self.imageload,(100,100))

        self.image_rotated=self.image
        self.rect=self.body_image.get_rect()
        self.rect_rotated=self.rect
        self.offset=self.body_image.get_height()-30

        self.posy=DISPLAY_HEIGHT-self.offset
        self.pos=(self.posx,self.posy)
        self.rect.center=self.pos
        self.canfire=True
        self.ammo_supplies=15000
        self.maxammo=5
        self.score=0
        self.scoreinc=150
        self.ammo=self.maxammo
        self.IsReloading=False
        self.speed=10
```

Image 2.3: Player Class Initialization

This class utilizes several methods including reset, reload, movement, and update. The reset method sets the basic variables of the class to their default values.

```
def reset(self):
    self.posx=DISPLAY_WIDTH/2
    self.posy=DISPLAY_HEIGHT-self.offset
    self.pos=(self.posx,self.posy)
    self.ammo_supplies=15000
    self.maxammo=5
    self.score=0
    self.scoreinc=150
    self.ammo=self.maxammo
    self.IsReloading=False
    self.speed=10
```

Image 2.4: Player Class reset

The reload method updates the ammo count of the player to the maximum value when the player presses the assigned button. The if statements are used to identify the player who is reloading(either player1 or player2), and if the player's ammo has reached zero.

In addition, there are movement and update methods. The movement method sets the position of the player on the screen based on the keyboard input. As previously mentioned, the player can only move vertically using the "A" and "D" keys. The "if" statements in the class check:

1) The playerID to identify which player is setting the input.

2) If the player is within the boundaries where they can be visible on the display.

```
def reload(self):
    pressed_key=pygame.key.get_pressed()
    if(self.playerID==1):
        if pressed_key[K_s]:
            if self.ammo_supplies>1:
                if self.ammo<self.maxammo:
                    self.IsReloading=True
                else:
                    self.IsReloading=False
                if self.IsReloading:
                    self.ammo=5
                    reload_sound.play()
            else:
                print("NO AMMO!")
    elif(self.playerID==2):
        if pressed_key[K_DOWN]:
            if self.ammo_supplies>1:
                if self.ammo<self.maxammo:
                    self.IsReloading=True
                else:
                    self.IsReloading=False
                if self.IsReloading:
                    self.ammo=5
                    reload_sound.play()
            else:
                print("NO AMMO!")

def movement(self):
    if(self.playerID==1):
        pressed_key=pygame.key.get_pressed()
        if pressed_key[K_d]:
            if self.posx<DISPLAY_WIDTH-self.offset:
                self.posx+=self.speed
        if pressed_key[K_a]:
            if self.posx>(self.offset/2-10):
                self.posx-=self.speed
    elif(self.playerID==2):
        pressed_key=pygame.key.get_pressed()
        if pressed_key[pygame.K_RIGHT]:
            if self.posx<DISPLAY_WIDTH-self.offset:
                self.posx+=self.speed
        if pressed_key[pygame.K_LEFT]:
            if self.posx>(self.offset/2-10):
                self.posx-=self.speed
    self.pos=(self.posx,self.posy)
    self.rect.center=self.pos

def update(self):
    self.movement()
    pass
```

Image 2.5: Player Class reload, movement and update methods

Finally, in the update method, the movement method is called and is updated every frame per second in the main game file.

2.1.3 Projectile Class

The Projectile class is used to determine the bullets shot by the player, and are spawned when a player fires. Spawnpoint and color are important parameters that need to be set when creating a Projectile object.

In the _init_ method, the width and height variables are initialized to specify the size of the object. The dimensions of a Projectile are saved in a tuple named size. The body of the Projectile is a rectangle created with the specified size and color. The color of the object is used to set the playerID of the bullet. If the bullet fired is red, it means that it originated from player 1. On the other hand, if the bullet is green, it means that it originated from player 2.

The code sets the values for three variables: rect, position, and speed. The rect variable is created using the get_rect method, which is also used in other classes. It acts as a hitbox to check if the object collides with a target.

```
import pygame
import math
import random

from Var.Constants import LINK_ASSETS_PLAYER,COLOR_RED,COLOR_GREEN,DISPLAY_WINDOW
from Var.Variables import P,P2,Target_spawn

class Projectile(pygame.sprite.Sprite):
    def __init__(self,spawn_point,color):
        super().__init__()
        try:
            self.width=5
            self.height=5
            self.size=(self.width,self.height)
            self.body=pygame.Surface(self.size)
            self.body.fill(color)
            if(color==COLOR_RED):
                self.parentID=1
            elif(color==COLOR_GREEN):
                self.parentID=2
            self.rect=self.body.get_rect()
            self.speed=30
            self.posx,self.posy=spawn_point
            self.pos=(self.posx,self.posy)
            mousex,mousey=pygame.mouse.get_pos()
            self.direction=(self.posx-mousex,self.posy-mousey)
            distance=math.hypot(*self.direction)
            self.direction=(self.direction[0]/distance,self.direction[1]/distance)
        except pygame.error as e:
            print(f"Error in object iniialization : {e}")
```

Image 2.6: Projectile Class Initialization

The Draw method of this class gets the rectangle of the Projectile's body and then draws it in the display, using pygame's blit method.

In the code, the update method is responsible for setting the Object's movement. Specifically, for the Projectile Object, it moves only in a vertical direction. If the Object's position goes beyond the display, it will destroy itself. On the other hand, if the Object collides with a target, it will also get destroyed, and the score of the player who fired the Object will increase.

```
def draw(self):
    self.rect=self.body.get_rect(center=self.pos)
    DISPLAY_WINDOW.blit(self.body,self.rect)

def update(self):
    self.posy-=self.speed
    self.pos=(self.posx,self.posy)
    self.rect.center=self.pos
    if self.posy<-10:
        self.kill()
    if pygame.sprite.spritecollide(self,Target_spawn.group,True):
        self.kill()
        if(self.parentID==1):
            P.score+=P.scoreinc
        if(self.parentID==2):
            P2.score+=P2.scoreinc
```

Image 2.7: Projectile Class draw and update methods

2.1.4 Target Class

The Target class represents the enemies that the player needs to destroy to defend the base. Similar to other classes mentioned earlier, when an object of this class is initialized, it first loads its image and then creates a rectangle based on the image's shape and size. The class also has variables for speed, position, and direction. The position variable is a tuple that contains two other variables: posx and posy, which indicate the object's position on the X and Y axis of the display. Finally, the movement direction of the object is chosen randomly using Pygame's random method.

The methods used by this class are movement, move and update.

The movement method is responsible for updating the position of each target that appears on the screen. In our game, the direction of the target depends on which side of the screen the object is spawned. While the target is on the screen, it moves downwards towards the player. If the target hits the hitbox object or gets hit, it gets destroyed.

Inside the move method, the movement method is called, and it updates the position of the Object. Finally, in the update method, the move method is called and updated every frame per second.

```python
import pygame
import random

from Var.Constants import LINK_ASSETS_TARGET,DISPLAY_WIDTH,DISPLAY_HEIGHT

random.seed()

class Target (pygame.sprite.Sprite):
    def __init__(self):
        super().__init__()
        try:
            self.image = pygame.image.load(LINK_ASSETS_TARGET)
            self.rect=self.image.get_rect()
            self.offset=self.image.get_height()
            self.speed=1
            self.posx=random.randint(40,DISPLAY_WIDTH-40)
            self.posy=-100
            self.pos=(self.posx,self.posy)
            self.rect.center=self.pos
            self.direction=random.choice([-1,1])
            self.damage=0
        except pygame.error as e:
            print(f"Error in object iniialization : {e}")
        self.image=self.image.convert_alpha()

    def movement(self):
        if self.posx<25 or self.posx>DISPLAY_WIDTH-25:
            self.direction*=-1
        if self.posy<DISPLAY_HEIGHT:
            self.posy+=self.speed
            self.posx+=self.speed*self.direction
            self.pos = (self.posx,self.posy)
        else:
            self.kill()
        pass

    def move(self):
        self.movement()
        self.rect.center=self.pos

    def update(self):
        self.move()
```

Image 2.8: Target Class

2.1.5 Target Spawner Class

To create Enemies in our game, we use a class called "Spawner". When we place a Spawner in the game, it will create Enemies at its location. To use the Spawner, we first need to create a group of Enemy Sprites. Then, we can create Enemy objects by setting the spawn time, damage, and speed for each Enemy.

This class uses several methods including spawn, reset, reset_timer, and update. The spawn method is responsible for creating, setting up, and adding Targets to the group of the class. On the other hand, the reset method simply empties the Sprite group of the Targets. The reset_timer method resets the timer to its default value. The update method is responsible for updating the sprite group every frame. When the timer reaches 0, the Spawner calls the

spawn method and resets the timer. This means that a Target is spawned whenever the timer reaches 0.

```python
import pygame

from Classes.Target import Target

class Spawner():
    def __init__(self,time,damage,speed):
        try:
            print("Spawner Created!")
            self.group=pygame.sprite.Group()
            self.time_set=time
            self.spawn_time=self.time_set
            self.difficulty=[0,1,2]
            self.index=0
            self.targetdmg=damage
            self.targetspeed=speed
        except pygame.error as e:
            print(f"Error in object iniialization : {e}")

    def spawn(self):
        T = Target()
        T.damage=self.targetdmg
        T.speed=self.targetspeed
        self.group.add(T)

    def reset(self):
        self.group.empty()

    def reset_timer(self):
        self.spawn_time=self.time_set

    def update(self):
        self.group.update()
        if self.spawn_time==0:
            self.spawn()
            self.reset_timer()
        self.spawn_time-=1
```

Image 2.9: Target Spawner Class

2.1.6 Hitbox Class

The Hitbox class object represents the base that has to be defended by the player. As presented in other classes of this object, there is the image of the Hitbox Object. In this particular class, the rectangle is set at the bottom of the display screen. This rectangle is used as a hitbox. If the hitbox is hit, it loses HP.

Some other variables used are two Booleans called IsRepairing and dead. The first one checks if the player is repairing the base and the dead variable checks if the base is destroyed.

This class makes use of four methods: repair, check_if_dead, reset, and update.

The repair method is triggered when the player presses the repair button (F) to fix the base. While repairing, if the base's HP is below its maximum, the

IsRepairing Boolean is set to true. During the repair process, the base's HP increases by 1 while the repair time decreases by 1. Once the repair button is released, the repair time is reset.

The check_if_dead method checks if the base is destroyed. If the base's HP is greater than 0 the dead variable is set to false. Else the dead variable is set to true.

The reset method resets all of the methods of this class to its default values.

Finally in the update method, the repair, and check_if_dead methods are called on every frame.

```
import pygame

from pygame.locals import *

from Var.Constants import DISPLAY_WINDOW,COLOR_RED,LINK_ASSETS_BASE

class Hitbox():
    def __init__(self,a,b):
        try:
            self.rect=pygame.Rect(0,a-100,b,100)
            self.image=pygame.image.load(LINK_ASSETS_BASE)
            self.image=self.image.convert_alpha()
            self.base_hp=1000
            self.hp=self.base_hp
            self.IsRepairing=False
            self.dead=False
            self.repair_time=3000
        except pygame.error as e:
            print(f"Error in object initialization : {e}")

    def repair(self):
        pressed_key=pygame.key.get_pressed()
        if self.dead==False:
            if pressed_key[K_f]:
                if self.repair_time>0:
                    if self.hp<self.base_hp:
                        self.IsRepairing=True
                    else:
                        self.IsRepairing=False
                    if self.IsRepairing:
                        self.hp+=1
                        self.repair_time-=1
                else:
                    self.repair_time=3000
            else:
                self.IsRepairing=False

    def check_if_dead(self):
        if self.hp>0:
            self.dead=False
        else:
            self.dead=True

    def reset(self):
        self.base_hp=1000
        self.hp=self.base_hp
        self.IsRepairing=False
        self.dead=False
        self.repair_time=3000

    def update(self):
        self.check_if_dead()
        self.repair()
```

Image 2.10: Hitbox Class

2.1.7 Scope Class

At first, our plan for this class was to have the player aim at the Targets, while the projectile followed the direction given by the player. However, we decided to simplify the gameplay and adopt a more straightforward approach inspired by Space Invaders to make the game more enjoyable.

This class uses only two attributes:

- Image: The sprite of the scope
- Rect: The rectangular created from the image.

The functions used by this class are:

- repair: Changes the scope's sprite to the Repair icon.
- Icon_reset: resets the Scope's sprite to the default one whenever desired.
- Update: this is the default update method for the class, in which the code sets the Scope's position equal to the position of the mouse.

```
import pygame

from Var.Constants import LINK_ASSETS_AIMCURSOR,LINK_ASSETS_REPAIR

class Aim (pygame.sprite.Sprite):
    def __init__(self) :
        super().__init__()
        try:
            self.image=pygame.image.load(LINK_ASSETS_AIMCURSOR)
            self.rect=self.image.get_rect()
            self.Fired=False
        except pygame.error as e:
            print(f"Error in object iniialization : {e}")

    def repair(self):
        self.image=pygame.image.load(LINK_ASSETS_REPAIR)

    def icon_reset(self):
        self.image=pygame.image.load(LINK_ASSETS_AIMCURSOR)

    def update (self):
        self.rect.center=pygame.mouse.get_pos()
```

Image 2.11: Scope Class

2.2 Constants and Variables Files

All the variables of the game are loaded and stored in individual files.

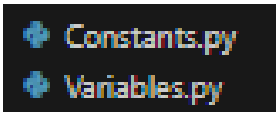


Image 2.12: Constants and Variables files

The Constants.py file loads and stores variables that remain constant throughout the game. These variables include the game title, game exit text, asset links, font, and display variables such as height, width, colors, FPS, and the game clock. The game assets are stored in a distinct folder within the game's directory.

```python
import pygame

pygame.init()

try:
    GAME_TITLE="-------DECK COMMANDER-------"
    GAME_EXIT="Press TAB to exit"


    #+++++LINKS+++++

    LINK_ASSETS_BASE_PC="H:\\My Drive\\Drive fanagiannis\\ΠΜΣ\\ΜΑΘΗΜΑΤΑ\\DECK COMMANDER\\assets"
    LINK_ASSETS_BASE_LAPTOP="G:\\Τα Drive μου\\Drive fanagiannis\\ΠΜΣ\\ΜΑΘΗΜΑΤΑ\\PYTHON\\assets"
    LINK_ASSETS_BASE=LINK_ASSETS_BASE_PC
    LINK_ASSETS_PLAYER="assets\\Player.png"
    LINK_ASSETS_CURSOR="assets\\Aim.png"
    LINK_ASSETS_AIMCURSOR="assets\\AimBig.png"
    LINK_ASSETS_BULLETS="assets\\Bullet.png"
    LINK_ASSETS_TARGET="assets\\Enemy.png"
    LINK_ASSETS_REPAIR="assets\\Repair.png"
    LINK_ASSETS_ROCKET="assets\\Rocket.png"
    LINK_ASSETS_BACKGROUND="assets\\Space.png"
    LINK_ASSETS_SCREEN="assets\\Screen.png"
    LINK_ASSETS_SCREEN2="assets\\Screen2.png"
    LINK_ASSETS_SCREEN3="assets\\Screen3.png"
    LINK_ASSETS_PLANET="assets\\Planet4.png"#Planet_8bit.png"
    LINK_ASSETS_PLANET2="assets\\Planet2_8bit.png"
    LINK_ASSETS_PLANET3="assets\\Planet3_8bit.png"
    LINK_ASSETS_SPACESHIP="assets\\Spaceship.png"
    LINK_ASSETS_SPACESHIP2="assets\\Spaceship2.png"
    LINK_ASSETS_GUN="assets\\Gun.png"
    LINK_ASSETS_BASE="assets\\Base.png"
    #+++++FONT+++++
    FONT_LCD="Fonts\\pixel_lcd_7.ttf"
    FONT_BASIC=pygame.font.Font(FONT_LCD,15)
    FONT_STATS=pygame.font.Font(FONT_LCD,25)
    FONT_GAME_OVER=pygame.font.Font(FONT_LCD,40)
    FONT_MENU_TITLE=pygame.font.Font(FONT_LCD,60)

    #+++++DISPLAY+++++

    DISPLAY_WIDTH=1280
    DISPLAY_HEIGHT=720
    DISPLAY_WINDOW=pygame.display.set_mode((DISPLAY_WIDTH,DISPLAY_HEIGHT))
```

Image 2.13: Constants file A

```
#+++++COLORS+++++

    COLOR_BLACK=pygame.Color(0,0,0)
    COLOR_WHITE=pygame.Color(255,255,255)
    COLOR_GREY=pygame.Color(128,128,128)
    COLOR_RED=pygame.Color(255,0,0)
    COLOR_YELLOW=pygame.Color(255,255,0)
    COLOR_GREEN=pygame.Color(5, 242, 68)
    GAME_CLOCK=pygame.time.Clock()
    FPS_MENU=30
    FPS=60

except pygame.error as e:
    print(f"Error in assets loading : {e}")
```

Image 2.14: Constants file B

In the Variables.py file, all of the variables used in the code are declared.

To begin with, in this code, the Scope object is represented by ADS, while Player1 and Player2 are represented by P and P2 respectively. Next, the positions of the guns for each player are declared - this is where their bullets will be instantiated. Additionally, the game mode and target spawner objects are declared. To allow players to shoot multiple bullets, a sprite group is created for the projectile objects spawned. Three Boolean flags are used in the code: run_main_game, game_over, and multiplayer. These flags determine whether the game is running in singleplayer mode, multiplayer mode, or if the game is over. Finally, the Hitbox object has to be declared inserting the display width and height as mentioned in the explanation of this particular class.

At the end of this file, the UI positions are declared. The code described above is shown in the following images:

```
import pygame

from Var.Constants import DISPLAY_HEIGHT,DISPLAY_WIDTH
from Classes.Player import Player
from Classes.Scope import Aim
from Classes.Spawner_Targets import Spawner
from Classes.Hitbox import Hitbox
from Classes.Gamemode import Gamemode

ADS=Aim()                                                   #SCOPE OBJECT DECLARATION

global P,gm,Target_spawn
P=Player(1)
global P2
P2=Player(2)                                                #PLAYER OBJECT DECLARATION

gunpos=P.pos                                                # GUN POSITION SET
gunpos_2=P2.pos                                             # GUN 2 POSITION SET

gm=Gamemode()                                               #GAMEMODE OBJECT DECLARATION
Target_spawn=Spawner(gm.t_spawn_time,gm.t_Damage,gm.t_speed)    #ENEMY SPAWNER DECLARATION

projectiles_group=pygame.sprite.Group()                     #SET PROJECTILE GROUP
hit=False                                                   #//CUT//

global run_game,run_main_game,run_main_menu,game_over,multiplayer          #BASIC BOOLEANS TO RUN THE GAME
run_main_menu=True
run_main_game=True
run_game=False
game_over=False
multiplayer=False

global score_live,gameround_live
```

Image 2.15: Variables File A

```
#HITBOX

hitbox = Hitbox(DISPLAY_HEIGHT,DISPLAY_WIDTH)                    #ALLY OBJECT DECLARATION (REVERSE!)

#UI

    #UI POSITIONS
ui_left_posx=30
ui_posy=15
ui_right_posx=DISPLAY_WIDTH-220

    #UI LEFT
score_message_pos=(ui_left_posx,ui_posy*2)
gameround_pos=(ui_left_posx,ui_posy*3)
multiplayer_gameround_pos=(ui_left_posx,ui_posy*5)

P2_score_message_pos=(ui_left_posx,ui_posy*4)
P2_energy_message_pos=(ui_right_posx,ui_posy*3)

    #UI RIGHT
hp_message_pos=(ui_right_posx,ui_posy)
energy_message_pos=(ui_right_posx,ui_posy*2)

energy_no_message_pos=(ui_right_posx,100)
energy_no_message_pos2=(ui_right_posx,130)
username_pos=(ui_left_posx,ui_posy)
username_pos2=(ui_left_posx,ui_posy*3)

    #GAME OVER POS
game_over_message_pos=(DISPLAY_WIDTH/2-125,DISPLAY_HEIGHT/2-75)
game_over_message_stats_pos=(DISPLAY_WIDTH/2-275,DISPLAY_HEIGHT/2)
game_over_message_stats_pos2=(DISPLAY_WIDTH/2-275,DISPLAY_HEIGHT/2+100)
game_over_return_mes_pos=(DISPLAY_WIDTH/2-200,ui_posy)

    #SCREEN POS
screen1_pos=(DISPLAY_WIDTH-200,0)
screen2_pos=(0,0)

score_value=100
```

Image 2.16: Variables File B

2.3 Asset Files

The sprites and sound effects are loaded in separate files.

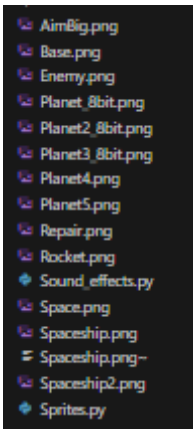


Image 2.17: Assets Files

The sound effects are loaded in the game using the pygame's sound mixer. The links for each sound effect are loaded from the file they are stored.

```python
import pygame

pygame.mixer.init()
reload_sound=pygame.mixer.Sound("Sounds\\recharge.wav")
shooting_sound=pygame.mixer.Sound("Sounds\\Shoot.wav")
explosion_sound=pygame.mixer.Sound("Sounds\\Explosion.wav")
game_over_sound=pygame.mixer.Sound("Sounds\\game_over.wav")
```

Image 2.18: Sounds File

The sprites file is a compilation of methods used to load the desired sprites in the game. The methods used are planets, hp, screeneffect, message, and background.

The planets function is used to load and display the planets' assets on the background. To load these Sprites it is important to import the assets needed from the Constants.py file. The Sprites are loaded using the pygame.image.load method. After that, the sprites are scaled to the desired size using Pygame's scale method. Finally, the Sprites are instantiated on screen using the blit method on the DISPLAY_WINDOW.

The screen effect method is used to fill the display with a specific color for one frame. It is employed when the base is hit by an enemy. When the base gets hit, the screen will temporarily turn red to create a hit effect.

The HP method is used to create a rectangle that corresponds to the HP (health points) of the base. The position of the rectangle is specified in the Variables.py file, and its width is set equal to the HP of the base.

The message function allows you to display a message on the screen in a specified location.

Lastly, the background function is used to create the background of the screen, along with the planets mentioned previously.

```python
import pygame


from Var.Variables import *
from Var.Constants import *

def screens():
    SCREEN=pygame.image.load(LINK_ASSETS_SCREEN3)
    DISPLAY_WINDOW.blit(SCREEN,screen1_pos)
    DISPLAY_WINDOW.blit(SCREEN,screen2_pos)

def planets():
    PLANET=pygame.image.load(LINK_ASSETS_PLANET)
    PLANETSCALED=pygame.transform.scale(PLANET,(200,200))
    PLANET2=pygame.image.load(LINK_ASSETS_PLANET2)
    PLANET2SCALED=pygame.transform.scale(PLANET2,(100,100))
    PLANET3=pygame.image.load(LINK_ASSETS_PLANET3)
    PLANET3SCALED=pygame.transform.scale(PLANET3,(100,100))

    DISPLAY_WINDOW.blit(PLANETSCALED,(-75,DISPLAY_HEIGHT/2))
    DISPLAY_WINDOW.blit(PLANET2SCALED,(DISPLAY_WIDTH-250,100))
    DISPLAY_WINDOW.blit(PLANET3SCALED,(450,0))

def hp():
    energy_supply_left=DISPLAY_WIDTH-180
    energy_supply_top=ui_posy+1
    energy_supply_width=hitbox.hp/7
    energy_supply_height=15
    energy_supply_pos=(energy_supply_left,energy_supply_top,energy_supply_width,energy_supply_height)
    pygame.draw.rect(DISPLAY_WINDOW,COLOR_GREEN,energy_supply_pos)

def screen_effect(color):
    DISPLAY_WINDOW.fill(color)

def message(text,text_color,text_pos,font):
    display_text=font.render(text,True,text_color)
    DISPLAY_WINDOW.blit(display_text,text_pos)
    pass

def background(image):
    background=pygame.image.load(image)
    DISPLAY_WINDOW.blit(background,(0,0))
    planets()
```

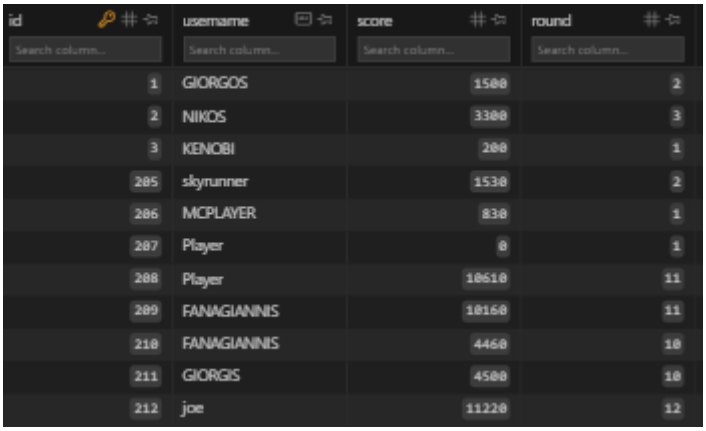
Image 2.19: Sprites File

2.4 Database Files

To have a leaderboard to display the best players in the game, there needs to be a database.

The leaderboard created is a database table with :

1. id (Integer, Primary Key)
2. username (Text)
3. score (Int)
4. round (Int)



id	username	score	round
1	GIORGOS	1500	2
2	NIKOS	3300	3
3	KENOBI	200	1
205	skyrunner	1530	2
206	MCPLAYER	830	1
207	Player	0	1
208	Player	10610	11
209	FANAGIANNIS	10160	11
210	FANAGIANNIS	4460	10
211	GIORGIS	4500	10
212	joe	11220	12

Image 2.20: Leaderboard Table

The id is a unique identification number, that is created automatically whenever a player is inserted in the table.

The username is the name chosen by the player before playing the game.

The score and round are stored whenever the game is over.

The Leaderboard.py method consists of a variety of methods used to manage the game's database. Firstly, the methods used by this class are:

1. **create_connection**

   To establish a connection with the Leaderboard database, the create_connection function is used. This function requires the SQLite3 Python library to be imported. In the main code, the path of the database must be specified to call this function correctly.

2. **query_execution**

   The SQL query is executed on the given connection using the SQLite's execute method. The query is then executed and the database updates using the commit method on the connection.

3. **query_read**

   This function executes a read query on the given connection whenever information from the leaderboard table is needed.

4. **create_leaderboard**

   This method was used to create the game’s leaderboard. Using this code a table, called players, is created. The table is created once by the developer. It is stored in the code as a function for update purposes.

5. **insert_leaderboard**

   Insert a new player on the Leaderboard.

6. **read_leaderboard**

   This function reads a player’s data from the Leaderboard and returns them.

7. **game_over_add_leaderboard**

   This method is called whenever the game ends. With this function, the database stores the player’s data in the Leaderboard when the game is over.

All of the functions described above are shown in the following images.

```python
import pygame
import pygame_menu
import sqlite3

from sqlite3 import Error

def create_connection(path):
    connection=None
    try:
        connection=sqlite3.connect(path)
        print("Connected to Database! ")
    except Error as e:
        print(f"Error '{e}' occured!")
    return connection

def query_execution(connection,query):
    cursor=connection.cursor()
    try:
        cursor.execute(query)
        connection.commit()
        print("Executed query!")
    except Error as e:
        print(f"Error '{e}' occured!")
        pass

def query_read(connection,query):
    cursor=connection.cursor()
    try:
        cursor.execute(query)
        result=cursor.fetchall()
        column_names=[description[0] for description in cursor.description]
        return column_names,result
    except Error as e:
        print(f"Error {e} occured!")
```

Image 2.21: Leaderboard methods A

```python
def create_leaderboard():
    query_create_table="""
    CREATE TABLE IF NOT EXISTS players (
    id INTEGER PRIMARY KEY AUTOINCREMENT,
    username TEXT NOT NULL,
    score INT,
    round INT
    );
    """
    query_execution(connection,query_create_table)

def insert_leaderboard(query):
    players=query_execution(connection,query)

def read_leaderboard(query):
    query_read_players=query
    players=query_read(connection,query_read_players)
    return players

def game_over_add_leaderboard(username,score,round):
    game_over_query="INSERT INTO players(username,score,round) VALUES ('"+ str(username) +"',"+str(score)+","+str(round)+")"
    insert_leaderboard(game_over_query)

connection=create_connection("Database\\deckcommander_db.sql")
```

Image 2.22: Leaderboard methods B

2.5 Main File

All of the files, functions/methods, and variables mentioned previously, are called in the game's main file, Game.py, to create a complete game.

Firstly, the necessary libraries have to be imported into the main file.

```
import pygame
import pygame_menu
import sys
import time

from pygame.locals import *
from pygame.sprite import *

from Classes.Scope import Aim
from Classes.Player import Player
from Classes.Target import Target
from Classes.Spawner_Targets import Spawner
from Classes.Hitbox import Hitbox
from Classes.Projectiles import Projectile
from Classes.Gamemode import Gamemode

from Database.Leaderboard import *
from Var.Variables import *
from Var.Constants import *
from assets.Sound_effects import shooting_sound,explosion_sound,game_over_sound
from assets.Sprites import *
```

Image 2.23: Main File Imports

To develop a game using Python, the developer can import the pygame library. This library provides the necessary methods and commands required to create the desired game.

In addition, the pygame_menu library can be used to create an interactive menu where a player can choose between the single-player or multiplayer mode of the game.

Next, the classes required for the game should be imported.

Finally, all the functions and variables, that were mentioned previously, should be imported as well and will be used by the main program.

Before running the main game, the pygame library has to be initialized. The game's caption and icon are set and the mouse visibility has to be set to false.

```
pygame.init()
pygame.display.set_caption("DECK COMMANDER V1.0")
icon=pygame.image.load(LINK_ASSETS_SPACESHIP)
icon=pygame.transform.scale(icon,(30,30))
pygame.display.set_icon(icon)
pygame.mouse.set_visible(False)
```

Image 2.24: Pygame Initialization

Moving on, below are the functions that help the code run the main game. Many of these functions are duplicates of each other for the multiplayer mode of the game.

Image 2.25: Main File Functions

The game runs using the game function. This function consists of five functions:

- maingame_solo: The single-player mode of the game.
- maingame_multiplayer: The multiplayer mode of the game.
- leaderboards: Setting the display of the leaderboard in the main menu.
- mainmenu: runs the main menu loop of the game.
- exit: The exit method of the game.

Starting the game, the menu style has to be set using the pygame_menu's methods. Initially, the menu's theme, background, title and font are set. Next, the menu that was set is stored in a variable used menu, using the pygame_menu.Menu method. Finally, the main menu function is called and the main menu loop is initialized. If the main menu's loop breaks, the player quits the game and the leaderboard is updated.

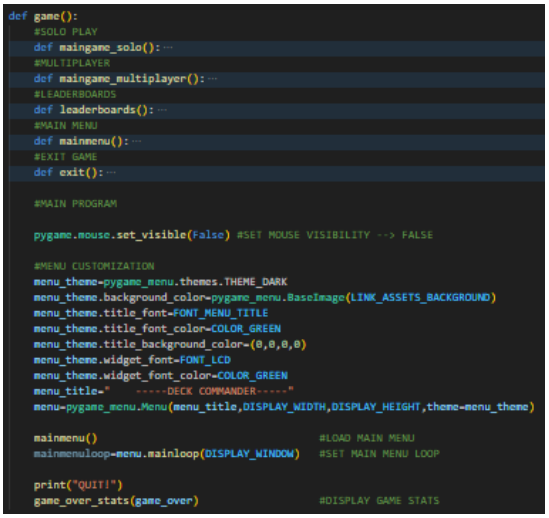


Image 2.26: Main Game Loop

The exit function is composed of two commands that allow the program to quit pygame and exit the game.

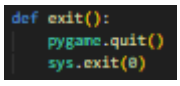


Image 2.27: Exit function

The leaderboards function creates the leaderboard table with the top 10 players. The leaderboard is displayed by adding a table on the menu using pygame_menu's add.table method and populating it with the players' statistics. After setting the borders and row names of the table, using the read_leaderboard method, the leaderboard can be populated with the players' stats. The stats of each player in the players' database are added after that.

The purpose of the main menu function is to set the style of the main menu of the game. Initially, a widget is created which contains two text inputs and three buttons. The two text inputs will allow the user to enter usernames for player1 and player2 respectively. On the other hand, the three buttons will work as selectors. The singleplayer button will initiate the single player mode of the game by calling the maingame_solo function, the multiplayer button will initiate the multiplayer mode of the game by calling the maingame_multiplayer function, and finally, the Quit button will call the exit function.

```
def mainmenu():
    menu_theme.widget_margin=(-600,0)
    global button_username,button_username2
    if gm.game_over:
        game_over_stats(game_over)
    button_username=menu.add.text_input(" Enter Username : ",default="Player",maxchar=12)
    button_username2=menu.add.text_input(" Enter Username 2: ",default="Player2",maxchar=12)
    leaderboards()
    button_startgame_solo=menu.add.button(" Singleplayer ",maingame_solo)
    button_startgame_multi=menu.add.button(" Multiplayer ",maingame_multiplayer)
    menu.add.button(" Quit ",exit)
```

Image 2.28: Main Menu function

The function maingame_solo sets the loop for the single player mode of the game. First, it calls a function called reset_game (which will be covered later on). Then, it creates a variable called "username", which is set by the player. Once this is done, the game loop begins.

The game loop consists of a while loop that breaks when the game has ended and two for loops. The first for loop acts as an event manager. In this loop, the code checks whether the game has ended or if there are any input events from the player. For example, if the player clicks the left mouse button and the hitbox is not dead, then the "fire" function will be called. Additionally, if the player presses the "LEFT_TAB" key on the keyboard, the game will exit and the leaderboard will be updated.

The second loop updates the projectile group whenever the player fires a bullet. Finally, the set_game_solo function is called, and the game clock and display are updated.

```
def maingame_solo():

    reset_game()
    pygame.mouse.set_visible(False)
    global username
    username=button_username.get_value()
    run_main_game=True
    while run_main_game:
        for event in pygame.event.get():
            if event.type==QUIT :
                    pygame.quit()
                    sys.exit(0)
            if event.type == MOUSEBUTTONUP:
                if event.button == 1:       #LEFT CLICK
                    if hitbox.dead==False:
                        fire()
            if event.type == KEYDOWN:
                pressed_key=pygame.key.get_pressed()
                if pressed_key[K_TAB]:
                    print("GAME EXITED!")
                    if gm.game_over:
                        game_over_stats()
                    run_main_game=False
        for Projectile in projectiles_group:
            Projectile.update()
        set_game_solo()
        pygame.display.flip()
        GAME_CLOCK.tick(FPS)
```

Image 2.29: Single Player mode Loop

As previously mentioned, the main_game_multiplayer function is triggered when the player selects the multiplayer mode. This function is essentially the same as the main_game_solo function, which was described earlier, with the only difference being the inclusion of functions for multiplayer. For instance, in the event detection loop, the code detects if the second player has fired a bullet via the fire2 function. Moreover, this loop utilizes the game_over_stats_multiplayer and set_game_multiplayer functions instead of the game_over_stats and set_game_solo functions, respectively.

```
def maingame_multiplayer():
    reset_game()
    pygame.mouse.set_visible(False)
    global username,username2
    username=button_username.get_value()
    username2=button_username2.get_value()
    run_main_game=True
    while run_main_game:
        for event in pygame.event.get():
            if event.type==QUIT :
                    pygame.quit()
                    sys.exit(0)
            if event.type == KEYUP:
                if event.key == K_w:
                    if hitbox.dead==False:
                        fire()
                if event.key == K_UP:
                    if hitbox.dead==False:
                        fire_2()
                if event.key == K_TAB:
                    print("GAME EXITED!")
                    if gm.game_over:
                        game_over_stats_multiplayer()
                    run_main_game=False
        for Projectile in projectiles_group:
            Projectile.update()
        set_game_multiplayer()
        pygame.display.flip()
        GAME_CLOCK.tick(FPS)
```

Image 2.30: Multiplayer mode loop

Here is the result of the code that was explained above :

Image 2 31: Main Menu

The set_game functions are used to set the basic display of the game. In these functions, the desired functions are called to set the scene on the display of the game, depending on the mode the player selects. All of the functions that will be mentioned are described previously.

To start the main_game_solo function, the code needs to call the "background", hp and messages functions. This will fill the display's background with the desired image from the assets that are imported from the Constants.py file. It will also display the HP bar of the base and set the desired messages, using the messages function. Then, the code checks if the base is destroyed. If so, event_game_over function is called. Otherwise, it spawns targets using the Target Spawn object and displays the base sprite.

By calling the message function, the desired messages can be displayed on the window. These messages give feedback to the player about the base's hp, the bullets remaining, the username, the score, etc. If the player has no ammo, then the No ammo message is displayed on the screen.

The event functions must be called, like the event_ally_hit and the event_repair. These functions are used to check if the base is hit and initialize the repair actions respectively.

Finally, the update functions for the objects of each class are called, and the rounds function. Then the pygame draws the objects that may be instantiated in the projectiles group.

```
def set_game_solo():
    background(LINK_ASSETS_BACKGROUND)          #BACKGROUND SET
    hp()                                        #POWER FEEDBACK
    messages()                                  #MESSAGES

    if hitbox.dead==False:
        Target_spawn.group.draw(DISPLAY_WINDOW)  #SET TARGET SPAWNER SPAWN
        DISPLAY_WINDOW.blit(hitbox.image,hitbox.rect)
    else:
        event_game_over()                        #IF GAME OVER --> DISABLE SPAWNER

    #SPAWN MESSAGES

    message(score_message_text,COLOR_GREEN,score_message_pos,FONT_BASIC)
    message(hp_message_text,COLOR_GREEN,hp_message_pos,FONT_BASIC)
    message(energy_message_text,COLOR_GREEN,energy_message_pos,FONT_BASIC)
    message(username,COLOR_GREEN,username_pos,FONT_BASIC)
    message(gameround_message_text,COLOR_GREEN,gameround_pos,FONT_BASIC)
    message(exit_game_message_text,COLOR_GREEN,game_over_return_mes_pos,FONT_BASIC)

    if P.ammo<1:
        message(energy_no_message_text,COLOR_GREEN,energy_no_message_pos,FONT_BASIC)   #SPAWN NO AMMO MESSAGE (IF NO AMMO)

    #EVENTS

    event_ally_hit()                            #ALLY HIT
    event_repair()                              #REPAIR

    DISPLAY_WINDOW.blit(P.image_rotated,P.rect_rotated) #PLAYER SPAWN

    #UPDATES
    P.reload()              #UPDATE PLAYER
    ADS.update()            #UPDATE AIM
    Target_spawn.update()   #UPDATE TARGET SPAWNER
    hitbox.update()         #UPDATE HITBOX

    #ROUND MANIPULATION
    rounds()

    for projectile in projectiles_group:    #DRAW PROJECTILES
        projectile.draw()
```

Image 2.32: The set game solo function

The set_game_multiplayer function is an exact copy of the set_game_solo, with the addition of some desired commands.

First, in addition to the messages function, the code calls the messages_multiplayer function, to spawn the desired messages for the second player of the game. Other things that are added to this function are :

- A blit method for the second player
- An update method for the second player

The event_game_over and rounds functions are replaced with the event_game_over_multiplayer and rounds_multiplayer functions respectively.

```
def set_game_multiplayer():
    background(LINK_ASSETS_BACKGROUND)          #BACKGROUND SET
    hp()                                        #POWER FEEDBACK
    messages()                                  #MESSAGES
    messages_multiplayer()                      #MULTIPLAYER MESSAGES

    if hitbox.dead==False:
        Target_spawn.group.draw(DISPLAY_WINDOW)  #SET TARGET SPAWNER SPAWN
    else:
        event_game_over_multiplayer()                       #IF GAME OVER --> DISABLE SPAWNER

    #SPAWN MESSAGES

    message(score_message_text,COLOR_GREEN,score_message_pos,FONT_BASIC)         #P1 SCORE MESSAGE
    message(p2_score_message_text,COLOR_GREEN,P2_score_message_pos,FONT_BASIC)      #P2 SCORE MESSAGE
    message(hp_message_text,COLOR_GREEN,hp_message_pos,FONT_BASIC)
    message(energy_message_text,COLOR_GREEN,energy_message_pos,FONT_BASIC)       #P1 AMMO
    message(p2_energy_message_text,COLOR_GREEN,P2_energy_message_pos,FONT_BASIC)    #P2 AMMO
    message(username,COLOR_GREEN,username_pos,FONT_BASIC)                    #HOST USERNAME
    message(username2,COLOR_GREEN,username_pos2,FONT_BASIC)                 #PLAYER2 USERNAME
    message(gameround_message_text,COLOR_GREEN,multiplayer_gameround_pos,FONT_BASIC)    #GANEROUND MESSAGE
    message(exit_game_message_text,COLOR_GREEN,game_over_return_mes_pos,FONT_BASIC)     #GAME OVER MESSAGE

    if P.ammo<1:
        message(energy_no_message_text,COLOR_GREEN,energy_no_message_pos,FONT_BASIC)   #SPAWN NO AMMO MESSAGE (IF NO AMMO)
    if P2.ammo<1:
        message(energy_no_message_text2,COLOR_GREEN,energy_no_message_pos2,FONT_BASIC)

    #EVENTS

    event_ally_hit()                            #ALLY HIT
    event_repair()                              #REPAIR
```

Image 2.33:The set game multiplayer function A

```
DISPLAY_WINDOW.blit(P.image_rotated,P.rect_rotated) #PLAYER SPAWN
DISPLAY_WINDOW.blit(P2.image_rotated,P2.rect_rotated) #PLAYER 2 SPAWN

#UPDATES
P.reload()              #UPDATE PLAYER
P2.reload()             #UPDATE PLAYER 2
P2.update()
ADS.update()            #UPDATE AIM
Target_spawn.update()   #UPDATE TARGET SPAWNER
hitbox.update()         #UPDATE HITBOX

#ROUND MANIPULATION
rounds_multiplayer()

for projectile in projectiles_group:    #DRAW PROJECTILES
    projectile.draw()
```

Image 2.34: The set game multiplayer function B

As previously mentioned, the messages and messages_multiplayer are used to set the desired feedback for the player to play the game.

```
def messages():
    global hp_live,score_message_text,hp_message_text,game_over_message_text,game_over_message_stats,energy_message_text,energy_no_message_te
    #STATS MESSAGES
    score_live="%06d" % P.score                 #SCORE MESSAGE

    #score2_live="%06d" % P2.score

    ammo_live="%02d" % int(P.ammo)              #PLAYER AMMO MESSAGE
    hp_live="%04d" % int(hitbox.hp)             #PLAYER HP MESSAGE
    gameround_live="%02d" % gm.round            #GAME ROUND MESSAGE
    #STATIC MESSAGES
    score_message_text = f"Score : {score_live}" #STATIC SCORE MESSAGE

    #score2_message_text=f"Score : {score2_live}"

    hp_message_text = "HP "                     #STATIC HP MESSAGE
    game_over_message_text = "GAME OVER ! "     #STATIC GAME OVER MESSAGE
    game_over_message_stats =f" Round :"f"{gameround_live}"+" "+f"Player 1 Score : "f"{score_live}"  #STATIC GAME OVER MESSAGE WITH STATS
    energy_message_text = f"Energy : {ammo_live}"       #STATIC ENERGY MESSAGE
    energy_no_message_text = "OUT OF AMMO! "            #STATIC NO ENERGY MESSAGE
    gameround_message_text=f"Round : {gameround_live}"  #STATIC ROUND MESSAGE
    exit_game_message_text="PRESS TAB TO RETURN TO THE MAIN MENU"     #EXIT GAME MESSAGE
```

Image 2.35: The messages function

```
def messages_multiplayer():
    global p2_score_live,p2_ammo_live,p2_energy_message_text,p2_score_message_text,energy_no_message_text,energy_no_message_text2,game_over_message_stats2
    #STATS MESSAGES
    p2_score_live="%06d" % P2.score                     #SCORE MESSAGE
    p2_ammo_live="%02d" % int(P2.ammo)                  #PLAYER AMMO MESSAGE
    p2_energy_message_text = f"Energy 2: {p2_ammo_live}"     #STATIC ENERGY MESSAGE
    p2_score_message_text = f"Score 2: {p2_score_live}"      #STATIC SCORE MESSAGE
    energy_no_message_text = "PLAYER 1 OUT OF AMMO! "           #STATIC NO ENERGY MESSAGE
    energy_no_message_text2 = "PLAYER 2 OUT OF AMMO! "          #STATIC NO ENERGY MESSAGE
    game_over_message_stats2 =f" Round :"f"{gameround_live}"+" "+f"Player 2 Score : "f"{p2_score_live}"  #STATIC GAME OVER MESSAGE WITH STATS
```

Image 2.36: The messages multiplayer function

The functions rounds and rounds_multiplayer are responsible for managing the rounds of the game. These functions use variables from the objects of the player class, the target spawner class, and the game mode class. If a player's score reaches a certain number of points, then the round is increased. As the rounds increase, the targets become more difficult to hit. In multiplayer mode, the rounds change once the total score of both players reaches a certain threshold.

```
def rounds():
    if P.score>=gm.game_round_change_score:         #IF PLAYER SCORE EQUALS SCORE-CHANGING ROUND
        gm.round_inc()                              #CALL FUNCTION FOR ROUND INCREASE FROM GAMEMODE CLASS
    if gm.round>=gm.round_change:                   #IF ROUND REACHES A THRESHHOLD INCREASE DIFFICULTY
        gm.round_difficulty_inc()                   #CALL FUNCTION FOR DIFFICULTY INCREASE FROM GAMEMODE CLASS (INCREASES TARGET DAMAGE,SPEED AND DECREASES SPAWNTIME)
        Target_spawn.time_set=gm.t_spawn_time       #SET NEW TARGET SPAWNTIME PARAMETER
        Target_spawn.targetdmg=gm.t_Damage          #SET NEW TARGET DAMAGE PARAMETER
        Target_spawn.targetspeed=gm.t_speed         #SET NEW TARGET SPEED PARAMETER
def rounds_multiplayer():
    if P.score+P2.score>=gm.game_round_change_score:        #IF PLAYER SCORE EQUALS SCORE-CHANGING ROUND
        gm.round_inc()                              #CALL FUNCTION FOR ROUND INCREASE FROM GAMEMODE CLASS
    if gm.round>=gm.round_change:                   #IF ROUND REACHES A THRESHHOLD INCREASE DIFFICULTY
        gm.round_difficulty_inc()                   #CALL FUNCTION FOR DIFFICULTY INCREASE FROM GAMEMODE CLASS (INCREASES TARGET DAMAGE,SPEED AND DECREASES SPAWNTIME)
        Target_spawn.time_set=gm.t_spawn_time       #SET NEW TARGET SPAWNTIME PARAMETER
        Target_spawn.targetdmg=gm.t_Damage          #SET NEW TARGET DAMAGE PARAMETER
        Target_spawn.targetspeed=gm.t_speed         #SET NEW TARGET SPEED PARAMETER
```

Image 2.37: The rounds and rounds multiplayer functions

The fire and fire2 functions are used when a player fires a bullet both of these functions are the same. First, the code checks if a player is repairing. If he is repairing then the player can’t shoot. If he is not repairing then the code check if the player has ammo to shoot, and if the base is alive. If all the conditions are

met then the shooting sound plays, the player loses one ammo and the projectile is added to its group.

```
def fire():
    if hitbox.IsRepairing:              #IF PLAYER IS REPAIRING THEN FIRE IS DISABLED
        print("REPAIRING...")           #CONSOLE REPAIRING MESSAGE
    else:
        if P.ammo>=1:                   #IF PLAYER IS AMMO BIGGER THAN 1 THEN CHECK IF HITBOX IS ALIVE
            if hitbox.hp>0:             #CHECK IF HITBOX IS ALIVE
                shooting_sound.play()                                       #PLAY FIRE SOUND
                P.ammo-=1                                                   #DECREASE AMMO PER FIRE
                projectiles_group.add(Projectile(P.pos,COLOR_RED))                  #ADD PROJECTILE TO PROJECTILE GROUP (PROJECTILE CLASS)
def fire_2():
    if hitbox.IsRepairing:              #IF PLAYER IS REPAIRING THEN FIRE IS DISABLED
        print("REPAIRING...")           #CONSOLE REPAIRING MESSAGE
    else:
        if P2.ammo>=1:                   #IF PLAYER IS AMMO BIGGER THAN 1 THEN CHECK IF HITBOX IS ALIVE
            if hitbox.hp>0:             #CHECK IF HITBOX IS ALIVE
                shooting_sound.play()                                       #PLAY FIRE SOUND
                P2.ammo-=1                                                   #DECREASE AMMO PER FIRE
                projectiles_group.add(Projectile(P2.pos,COLOR_GREEN))                  #ADD PROJECTILE TO PROJECTILE GROUP (PROJECTILE CLASS)
```

Image 2.38: The fire and fire2 functions

The game_over_stats and game_over_stats_multiplayer functions call the game_over_add_leaderboard function from the Leaderboard file and updates the leaderboard with the stats of the active players.

```
def game_over_stats():
    game_over_add_leaderboard(username,P.score,gm.round)                    #ADDS STATS TO LEADERBOARD AFTER GAME OVER
    print("GAME OVER")
def game_over_stats_multiplayer():
    game_over_add_leaderboard(username,P.score,gm.round)                    #ADDS STATS TO LEADERBOARD AFTER GAME OVER
    game_over_add_leaderboard(username2,P2.score,gm.round)                  #ADDS STATS TO LEADERBOARD AFTER GAME OVER
    print("GAME OVER")
```

Image 2.39: The game over stats functions

The reset_game function calls every reset method for the objects of each class used and empties the projectiles in the projectiles group.

```
def reset_game():
    P.reset()                           #RESET PLAYER TO DEFAULT
    P2.reset()
    Target_spawn.reset()                #RESET ENEMY SPAWNER TO DEFAULT
    gm.reset()                          #RESET GAMEMODE TO DEFAULT
    hitbox.reset()                      #RESET ALLY TO DEFAULT
    projectiles_group.empty()           #RESET PROJECTILES TO DEFAULT
```

Image 2.40: The reset game function

Finally, the event functions are called whenever an event occurs.

The event_ally_hit function is called whenever the base is hit. When this occurs, the screen_effect function is called, the hit sound is played and the base takes damage. Also, the player loses 10 points from his score.

```
def event_ally_hit():
    if hitbox.dead==False:                                          #CHECKS IF ALLY IS DEAD (FUNCTION RUNS ONLY IF ALLY IS ALIVE)
        Ally_Hit=pygame.sprite.spritecollide(hitbox,Target_spawn.group,True)    #BOOL Ally_Hit CHECKS COLLISION OF ENEMY WITH ALLY
        if Ally_Hit:                                                #IF THE ALLY IS HIT THEN ...
            screen_effect(COLOR_RED)                                #HIT SCREEN EFFECT
            explosion_sound.play()                                  #HIT SOUND EFFECT
            hitbox.hp-=gm.t_Damage                                  #ALLY TAKE DAMAGE
            if P.score>0:                                           #IF PLAYER SCORE IS GREATER THAN 0
                P.score-=10                                         #THEN SUBTRACT 10 FROM SCORE AFTER ALLY HIT
            if hitbox.hp<0:                                         #DISPLAY HP ONLY GREATER THAN 0
                hitbox.hp=0
```

Image 2.41: The event ally hit function

The event_repair method is called whenever a player presses the repair key (F). In this function, the Scope's icon_reset method is called, if the base is not repairing, and the Player's movement is updated. On the contrary, if the player is repairing, then the Scope's image is set to the repair icon.

```
def event_repair():
    if hitbox.IsRepairing==False:                   #CHECKS IF ALLY IS NOT REPAIRING
        ADS.icon_reset()                            #SET AIM ICON TO INVISIBLE
        if hitbox.dead==False:                      #CHECKS IF ALLY IS ALIVE
            P.update()                              #IF ALLY IS ALIVE THEN KEEP UPDATING THE PLAYER
    else:                                           #IF ALLY IS REPAIRING
        DISPLAY_WINDOW.blit(ADS.image,ADS.rect)     #SET REPAIR ICON TO VISIBLE
        ADS.repair()                                #RUNS REPAIR FUNCTION FROM AIM CLASS
```

Image 2.42: The event repair function

Finally, the event_game_over and event_game_over_multiplayer functions are called whenever the game is over. Both of these methods, empty the targets from the target group and display the stats of the active players.

```
def event_game_over():
    Target_spawn.group.empty()                                                  #DELETE ALL ENEMY INSTANCES FROM ENEMY GROUP
    #game_over_sound.play(0)                                                    #//CUT//GAME OVER SOUND
    message(game_over_message_text,COLOR_GREEN,game_over_message_pos,FONT_GAME_OVER)    #DISPLAY GAME OVER MESSAGES
    message(game_over_message_stats,COLOR_GREEN,game_over_message_stats_pos,FONT_STATS)
    run_main_game=False                                                         #SET RUN GAME BOOLEAN TO FALSE
    gm.game_over=True                                                           #SET GAME OVER BOOLEAN TO TRUE
def event_game_over_multiplayer():
    Target_spawn.group.empty()                                                  #DELETE ALL ENEMY INSTANCES FROM ENEMY GROUP
    #game_over_sound.play(0)                                                    #//CUT//GAME OVER SOUND
    message(game_over_message_text,COLOR_GREEN,game_over_message_pos,FONT_GAME_OVER)    #DISPLAY GAME OVER MESSAGES
    message(game_over_message_stats,COLOR_GREEN,game_over_message_stats_pos,FONT_STATS)
    message(game_over_message_stats2,COLOR_GREEN,game_over_message_stats_pos2,FONT_STATS)
    run_main_game=False                                                         #SET RUN GAME BOOLEAN TO FALSE
    gm.game_over=True
```

Image 2.43: The event game-over functions

Sources

Space Invaders: https://en.wikipedia.org/wiki/Space_Invaders

Galaga: https://en.wikipedia.org/wiki/Galaga

Duck Hunt: https://en.wikipedia.org/wiki/Duck_Hunt

Gamefragger: https://gamefragger.com/

W3schools: https://www.w3schools.com/

Pygame Documentation: https://www.pygame.org/docs/

Convert_alpha(): https://stackoverflow.com/questions/32576648/what-do-subsurface-convert-alpha-and-do

Freesound: https://freesound.org/

Tutorials:

https://www.youtube.com/watch?v=y9VG3Pztok8

https://www.youtube.com/watch?v=KR2zP6yuWAs

https://www.youtube.com/watch?v=4TfZjhw0J-8

https://www.youtube.com/watch?v=OUOI6iCrmCk

Python playlist:

https://www.youtube.com/watch?v=C32EzXbYY2g&list=PLCjvBIkXlmeJL75a9aokT4HKWGOVODya4

Images

Table Of Contents