NATIONAL UNIVERSITY OF SCIENCES AND TECHNOLOGY

School of Electrical Engineering and Computer Sciences

Object Oriented Programming (CS-212)
PROJECT REPORT

SUBMITTED TO: *Dr. Ahsan Saadat*

SEMESTER: *Fall-2022*

CLASS + SECTION: *BEE-13A*

PROJECT TITLE: *Football Pygame*

SUBMISSION DATE: *02nd January 2023*

SUBMITTED BY:

1. *Zohaib Irfan* - *372692*
2. *Muhammad Sami Ullah* - *369138*
3. *Usman Ayub* - *368149*

Table of Contents

Abstract **4**
1. Libraries **5**
2. Initializations **5**
3. Classes **7**
3.1 Class ‘Messi’: 7
3.2 Class ‘Glove’: 8
3.3 Class ‘Football’: 9
4. Functions **10**
5. Object Instantiations **11**
6. Main Game Loop **14**
6.1 Background: 14
6.2 Player Movement: 15
6.3 Enemy Movement: 17
6.4 Glove Movement: 18
7. Module of Program in C++ **19**
8. Conclusion **20**
9. Targets Achieved **21**
10. Future Recommendations **21**
11. CEP Attribute Mapping **22**
11.1 WP1: Depth of Knowledge required 22
11.2 WP2: Range of conflicting requirements 22
11.3 WP7: Interdependence 22
11.4 WP8: Consequences 23
Appendix **24**

Table of Figures

Figure 1: Included Libraries 5
Figure 2.1: Games Initials 5
Figure 2.2: Background.png and enemy.png 6
Figure 3.1: Class 'Messi' 7
Figure 3.2: player.png 7
Figure 4.1: Class 'Glove' 8
Figure 4.2: bullet.png 8
Figure 5.1: Class 'Football' 9
Figure 5.2: Game Layout 9
Figure 6.1: Font Display 10
Figure 6.2: Score and Game Over Function 10
Figure 6.3: Display of Score and Game Over 11
Figure 7.1: Object Instantiations 11
Figure 7.2: Firing Glove 12
Figure 7.3: Collision Function 13
Figure 8.1: Main While Loop 14
Figure 8.2: Player Movement Keys 15
Figure 8.3: Movement of Player 16
Figure 8.4: Updating Player Position 16
Figure 9.1: Movement of Football 17
Figure 10.1: Movement of Glove 18

Abstract

Object-oriented programming (OOP) is a programming paradigm that organizes code into "objects" which represent real-world entities and their associated actions. Pygame, a library for creating video games in Python, can be used to develop games using OOP principles.

In a Pygame game developed using OOP, the game's components, such as characters, enemies, and items, can be represented as objects. Each object can have its own attributes (e.g. size, color, position) and methods (e.g. move, attack, interact). The objects can interact with each other and the game environment through methods, and the game's behavior can be controlled by creating relationships between the objects and modifying their attributes.

In this project, we present a shooting football game developed using the Pygame library in Python. The game we have developed is a shooting game in which the player controls a character that can move horizontally across the screen and throw a glove to hit footballs that fall from the top of the screen. The footballs move horizontally and change direction when they reach the edges of the screen. If a football is hit by a glove, it is removed from the game and the player's score increases. The game continues until all the footballs have been hit or a football collides with the player's character, at which point the game is over and the player's score is displayed on the screen.

In the following sections of this report, we will discuss the design and implementation of the game, including the game's mechanics, the use of Pygame's features, and the challenges we faced during development as well as the potential future improvements to the game.

1. Libraries

First of all, we import the `math`, `random`, and `pygame` libraries. The `math` library is a built-in Python library that provides mathematical functions such as trigonometric functions, logarithmic functions, and mathematical constants. The `random` library is a built-in Python library that provides functions for generating random numbers and shuffling sequences randomly. The `pygame` library is a third-party library for creating video games in Python. It provides a set of tools for handling game logic, graphics, sound, and input.

Next, the code imports the `mixer` module from the `pygame` library. The `mixer` module provides functions for playing and controlling audio in Pygame games. The mixer module is imported using from `pygame import mixer`, which allows it to be used without prefixing it with the `pygame` module name

```
import math
import random

import pygame
from pygame import mixer
```

Figure 1: Included Libraries

2. Initializations

To initialize the basic display of the game, we carry out the following operations:

```
# Intialize the pygame
pygame.init()

# create the screen
screen = pygame.display.set_mode((800, 600))

# Background
background = pygame.image.load('background.png')

# Sound
mixer.music.load("background.wav")
mixer.music.play(-1)

# Caption and Icon
pygame.display.set_caption("Space Invader")
icon = pygame.image.load('enemy.png')
pygame.display.set_icon(icon)
```

Figure 2.1: Games Initials

`pygame.init()` initializes the `Pygame` library and prepares it for use. This must be called before any other `Pygame` functions can be used.

`screen = pygame.display.set_mode((800, 600))` creates a window with a width of `800` pixels and a height of `600` pixels. The window is used as the display surface for the game and is where the game's graphics will be drawn.

`background = pygame.image.load('background.png')` loads an image file named background.png and stores it in a variable called background. This image will be used as the background for the game.

`mixer.music.load("background.wav")` and `mixer.music.play(-1)` load an audio file named background.wav and play it continuously in the background. The -1 parameter passed to the `play()` function causes the audio to loop indefinitely.

Then we set the caption and icon for the game window. The caption is the text that appears in the title bar of the window and is set using the `set_caption()` function. The icon is the image that appears in the window's title bar and taskbar, and is set using the `set_icon()` function. The caption and icon are set to "`Space Invader`" and the enemy.png image, respectively.

Figure 2.2: Background.png and enemy.png

3. Classes

3.1 Class ‘Messi’:

Following code snippet show the creation of class Messi which is our player

```
# Messi
class Messi:
    messiImg = pygame.image.load('player.png')
    messiX = 370
    messiY = 480
    messiX_change = 0
```

Figure 3.1: Class ‘Messi’

This code defines a class called Messi which represents the player character in the game. A class is a template for creating objects, and an object is an instance of a class.

The Messi class has four attributes: `messiImg`, `messiX`, `messiY`, and `messiX_change`. The `messiImg` attribute is an image of the player character, which is loaded from the file `player.png` using the `pygame.image.load()` function. The `messiX` and `messiY` attributes represent the horizontal and vertical position of the player character on the game screen, respectively. The `messiX_change` attribute represents the change in the player character's horizontal position, which is used to move the player character left or right.

Figure 3.2: player.png

3.2 Class ‘Glove’:

Now, we create the class ‘Glove’ which acts as a weapon to kick the football.

```
# Glove

# Ready - You can't see the glove on the screen
# Throw - The glove is currently moving
class Glove:
    gloveImg = pygame.image.load('bullet.png')
    gloveX = 0
    gloveY = 480
    gloveX_change = 0
    gloveY_change = 2.5
    glove_state = "ready"
```

Figure 4.1: Class ‘Glove’

This code defines a class called **`Glove`** which represents the glove that the player character throws in the game. A class is a template for creating objects, and an object is an instance of a class.

The Glove class has six attributes: **`gloveImg`**, **`gloveX`**, **`gloveY`**, **`gloveX_change`**, **`gloveY_change`**, and **`glove_state`**. The **`gloveImg`** attribute is an image of the glove, which is loaded from the file bullet.png using the **`pygame.image.load()`** function. The **`gloveX`** and **`gloveY`** attributes represent the horizontal and vertical position of the glove on the game screen, respectively. The **`gloveX_change`** and **`gloveY_change`** attributes represent the changes in the glove's horizontal and vertical positions, respectively, which are used to move the glove. The **`glove_state`** attribute is a string representing the state of the glove. It can have two possible values: "**`ready`**" or "**`throw`**". If the value is "**`ready`**", the glove is not currently moving. If the value is "**`throw`**", the glove is currently moving.

Figure 4.2: bullet.png

3.3 Class 'Football':

Following code shows the creation of class 'Football' which acts as an enemy whom we have to hit.

```
# Football
class Football:
    footImg = []
    footX = []
    footY = []
    footX_change = []
    footY_change = []
    num_of_footballs = 6
    for i in range(num_of_footballs):
        footImg.append(pygame.image.load('enemy.png'))
        footX.append(random.randint(0, 736))
        footY.append(random.randint(50, 150))
        footX_change.append(0)
        footY_change.append(0.125)
```

Figure 5.1: Class 'Football'

This code defines a class called **`Football`** which represents the footballs in the game. A class is a template for creating objects, and an object is an instance of a class.

The Football class has six attributes: **`footImg`**, **`footX`**, **`footY`**, **`footX_change`**, **`footY_change`**, and **`num_of_footballs`**. The **`footImg`** attribute is a list of images of the footballs, which are loaded from the file enemy.png using the **`pygame.image.load()`** function. The **`footX`** and **`footY`** attributes are lists of the horizontal and vertical positions of the footballs on the game screen, respectively. The **`footX_change`** and **`footY_change`** attributes are lists of the changes in the footballs' horizontal and vertical positions, respectively, which are used to move the footballs. The **`num_of_footballs`** attribute is an integer representing the number of footballs in the game.

The Football class also has a **`for loop`** which initializes the attributes of the class. The loop iterates **`num_of_footballs`** times and appends an image, a random initial horizontal position between **`0 and 736 pixels`**, a random initial vertical position between **`50`** and **`150`** pixels, a horizontal change of **`0 pixels`**, and a vertical change of **`0.125 pixels`** to the **`footImg`**, **`footX`**, **`footY`**, **`footX_change`**, and **`footY_change`** lists, respectively.

Figure 5.2: Game Layout

4. Functions

Following code show the different function used to

```
# Score

score_value = 0
font = pygame.font.Font('freesansbold.ttf', 32)

textX = 10
testY = 10

# Game Over
over_font = pygame.font.Font('freesansbold.ttf', 64)
```

Figure 6.1: Font Display

This code defines variables and objects related to the score and game over state of the game.

The `score_value` variable is an integer representing the current score of the player. It is initialized to `0.`

The font variable is a `Pygame` font object that will be used to render the score text on the game screen. The font object is created using the `pygame.font.Font()` function, which takes the file name of a `TrueType` font file and the size of the font as arguments. In this case, the font file is `freesansbold.ttf` and the size is `32 pixels`.

The `textX` and `testY` variables are integers representing the horizontal and vertical positions of the score text on the game screen, respectively. They are initialized to `10 pixels.`

The `over_font` variable is a `Pygame` font object that will be used to render the "`game over`" text on the game screen when the game is over. It is created using the `pygame.font.Font()` function, which takes the file name of a TrueType font file and the size of the font as arguments. In this case, the font file is `` `freesansbold.ttf. ``

```
def show_score(x, y):
    score = font.render("Score : " + str(score_value), True, (255, 255, 255))
    screen.blit(score, (x, y))


def game_over_text():
    over_text = over_font.render("GAME OVER", True, (255, 255, 255))
    screen.blit(over_text, (200, 250))
```

Figure 6.2: Score and Game Over Function

The `show_score` function takes two arguments: x and y. These arguments represent the x-coordinate and y-coordinate of a point on the `Pygame` display window (screen), respectively. The function uses the `Pygame` font module to render (draw) the text "`Score :` " followed by the value of the `score_value` variable. The rendered text is drawn on the `Pygame` window at the point specified by the x and y arguments.

The `game_over_text` function displays the text "`GAME OVER`" on the Pygame window using the `over_font` variable. The text is rendered in white and drawn on the window at the point specified by the arguments (`200, 250`).

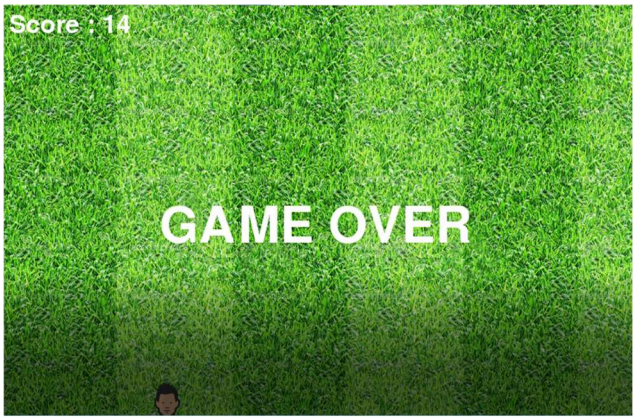

Figure 6.3: Display of Score and Game Over

5. Object Instantiations

Once we are done with the creation of classes and functions, we then test our classes by instantiating their objects.

```
m1 = Messi()
def player(x, y):
    screen.blit(m1.messiImg, (x, y))

f1 = Football()
def enemy(x, y, i):
    screen.blit(f1.footImg[i], (x, y))

g1 = Glove()
def throw_glove(x, y):
    global glove_state
    g1.glove_state = "fire"
    screen.blit(g1.gloveImg, (x + 16, y + 20))
```

Figure 7.1: Object Instantiations

The player function takes two arguments: `x and y`. These arguments represent the x-coordinate and y-coordinate of a point on the `Pygame` display window (screen), respectively. The function uses the `screen.blit` method to draw an image of a soccer player (`m1.messiImg`) on the `Pygame` window at the point specified by the x and y arguments.

The enemy function also takes x and y arguments, which specify the point on the `Pygame` window where the image of a football (`f1.footImg[i])` will be drawn. The function also takes an additional argument i, which is used to select a specific image from the `f1.footImg` list.

The `throw_glove` function takes x and y arguments, which specify the point on the `Pygame` window where the image of a throwing glove (`g1.gloveImg`) will be drawn. The function also sets the value of the `glove_state` variable to "`fire`". The global keyword is used to indicate that the `glove_state` variable is a global variable and can be accessed and modified from anywhere within the program. The image of the glove is drawn on the `Pygame` window at a point offset from the x and y arguments by `16 pixels` in the x-direction and `20 pixels` in the y-direction.

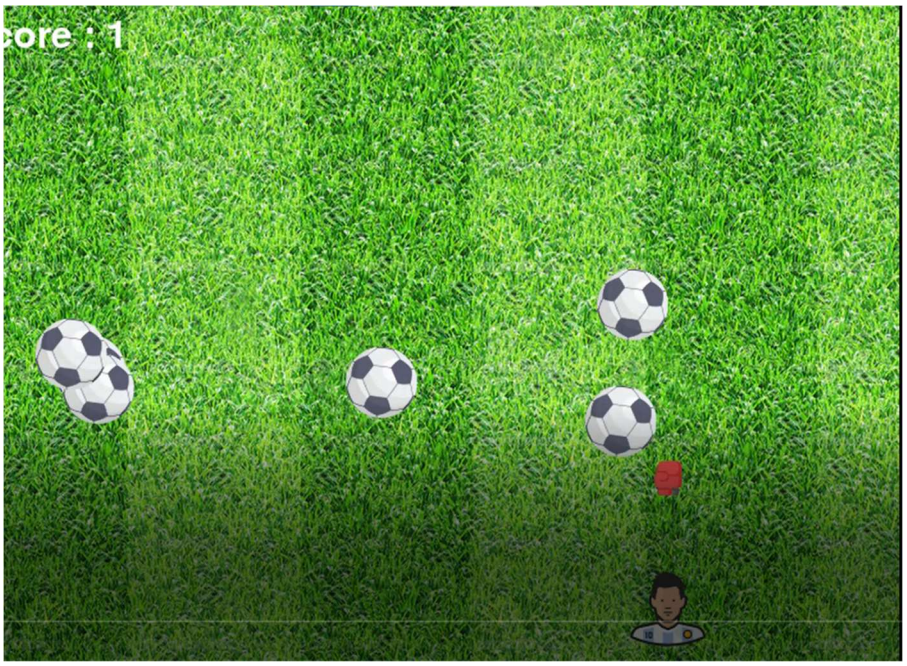

Figure 7.2: Firing Glove

```python
def isCollision(footX, footY, gloveX, gloveY):
    distance = math.sqrt(math.pow(footX - gloveX, 2) + (math.pow(footY - gloveY, 2)))
    if distance < 27:
        return True
    else:
        return False
```

Figure 7.3: Collision Function

This is a Python function named **`isCollision`** that takes four arguments: **`footX`**, **`footY`**, **`gloveX`**, and **`gloveY`**. These arguments represent the x-coordinate and y-coordinate of a football (**`footX`** and **`footY`**) and the x-coordinate and y-coordinate of a throwing glove (**`gloveX`** and **`gloveY`**), respectively.

The function uses the Pythagorean theorem to calculate the distance between the football and the glove. The theorem states that in a right triangle, the square of the hypotenuse (the distance between the two points) is equal to the sum of the squares of the other two sides.

The distance between the two points is calculated as the square root of the sum of the squares of the differences between the x-coordinates and the y-coordinates of the two points. If the distance is less than **`27`**, the function returns **`True`**. Otherwise, it returns **`False`**.

This function can be used to detect whether the football and the glove have collided with each other. If the function returns **`True`**, it means that a collision has occurred. If it returns **`False`**, it means that no collision has occurred.

6. Main Game Loop

The code defines a game loop that will run as long as the value of the running variable is `True`.

6.1 Background:

Inside the loop, the first thing that happens is that the `Pygame` window (screen) is filled with a `solid black` color. This is done using the `screen.fill` method, which takes a tuple of three integers as an argument. The integers represent the amount of red, green, and blue colors to be used, respectively. In this case, `the tuple (0, 0, 0)` specifies that no `red`, `green`, or `blue` colors should be used, resulting in a `black` color.

```
# Game Loop
running = True
while running:

    # RGB = Red, Green, Blue
    screen.fill((0, 0, 0))
    # Background Image
    screen.blit(background, (0, 0))
```

Figure 8.1: Main While Loop

Next, we use the `screen.blit` method to draw an image (background) on the `Pygame` window at the point specified by the arguments (0, 0). The `(0, 0)` point is the top-left corner of the window. As a result, the image will be drawn on the entire window, covering the black color that was previously filled.

This code is essentially setting up the background for the game. The `black` color is first filled on the window, and then the background image is drawn on top of it. This is a common technique in video games to create a scrolling background effect, where the background image is moved across the window to create the illusion of movement.

6.2 Player Movement:

Now we have to handle events that occur during the execution of the game. An event can be any user action (such as pressing a key on the keyboard or clicking the mouse) or an action performed by the program (such as quitting the game).

```
for event in pygame.event.get():
    if event.type == pygame.QUIT:
        running = False

    # if keystroke is pressed check whether its right or left
    if event.type == pygame.KEYDOWN:
        if event.key == pygame.K_LEFT:
            m1.messiX_change = -1.5
        if event.key == pygame.K_RIGHT:
            m1.messiX_change = 1.5
        if event.key == pygame.K_SPACE:
            if g1.glove_state is "ready":
                gloveSound = mixer.Sound("laser.wav")
                gloveSound.play()
                # Get the current x cordinate of the spaceship
                g1.gloveX = m1.messiX
                throw_glove(g1.gloveX, g1.gloveY)

    if event.type == pygame.KEYUP:
        if event.key == pygame.K_LEFT or event.key == pygame.K_RIGHT:
            m1.messiX_change = 0
```

Figure 8.2: Player Movement Keys

The for loop iterates over a list of events returned by **`the pygame.event.get()`** function. This function returns a list of all the events that have occurred since the last frame was drawn.

The if statement inside the loop checks the type of each event. If the event is a **`pygame.QUIT`** event, it means that the user has clicked the close button on the window. In this case, the value of the running variable is set to False, which will cause the game loop to terminate.

If the event is a **`pygame.KEYDOWN`** event, it means that a key on the keyboard has been pressed. The if statement checks the value of the **`event.key`** attribute to determine which key has been pressed. If the left arrow key (**`pygame.K_LEFT`**) has been pressed, the **`messiX_change`** attribute of the m1 object is set to **`-1.5`**. If the right arrow key (**`pygame.K_RIGHT`**) has been pressed, the **`messiX_change`** attribute is set to **`1.5.`**

If the space bar (**`pygame.K_SPACE`**) has been pressed, the program checks the value of the **`glove_state`** attribute of the **`g1 object`**. If the value is "**`ready`**", a glove sound is played and the **`gloveX`** attribute of the g1 object is set to the value of the **`messiX`** attribute of the **`m1 object`**. The **`throw_glove`** function is then called, with the **`gloveX`** and **`gloveY`** attributes of the g1 object as arguments.

If the event is a **`pygame.KEYUP`** event, it means that a key on the keyboard has been released. The if statement checks the value of the **`event.key`** attribute to determine which key has been

released. If the **left** arrow key or the **right** arrow key has been released, the **messiX_change** attribute of the **m1 object** is set to 0. This will stop the movement of the soccer player in the x-direction.

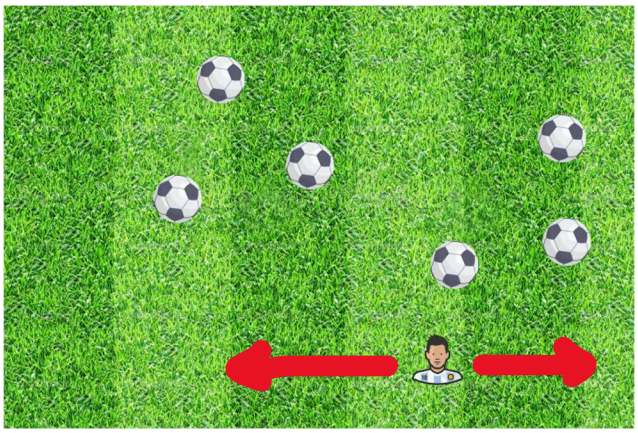

Figure 8.3: Movement of Player

Now we have to update the position of the soccer player (m1). The soccer player has an attribute called **messiX** that represents the x-coordinate of the player's position on the **Pygame** window. The player also has an attribute called **messiX_change**, which represents the amount by which the player's x-coordinate should change on each frame.

```
m1.messiX += m1.messiX_change
if m1.messiX <= 0:
    m1.messiX = 0
elif m1.messiX >= 736:
    m1.messiX = 736
```

Figure 8.4: Updating Player Position

The value of the **messiX** attribute by adding the value of the **messiX_change** attribute to it. This causes the player's position to be updated on the x-axis.

The following if statement checks whether the player's x-coordinate has become less than or equal to 0. If it has, the value of the **messiX** attribute is set to 0. This prevents the player from moving off the left side of the window.

The following elif statement checks whether the player's x-coordinate has become greater than or equal to 736. If it has, the value of the **messiX** attribute is set to 736. This prevents the player from moving off the right side of the window.

This code is essentially implementing boundary checking for the player's movement on the x-axis. It ensures that the player stays within the boundaries of the **Pygame** window and does not move off the screen.

6.3 Enemy Movement:

This is the part of the program in which a player controls a glove to catch falling footballs. The code appears to be responsible for controlling the movement and behavior of the footballs, which are referred to as enemies.

```
# Enemy Movement
for i in range(f1.num_of_footballs):
    # Game Over
    if f1.footY[i] > 490:
        for j in range(f1.num_of_footballs):
            f1.footY[j] = 2000
        game_over_text()
        break
    f1.footY[i] += f1.footY_change[i]
    if f1.footY[i] <= 0:
        f1.footY = 0

    # Collision
    collision = isCollision(f1.footX[i], f1.footY[i], g1.gloveX, g1.gloveY)
    if collision:
        explosionSound = mixer.Sound("explosion.wav")
        explosionSound.play()
        g1.gloveY = 480
        g1.glove_state = "ready"
        score_value += 1
        f1.footX[i] = random.randint(0, 736)
        f1.footY[i] = random.randint(50, 150)

    enemy(f1.footX[i], f1.footY[i], i)
```

Figure 9.1: Movement of Football

The program first sets up a loop that iterates through all of the footballs by using the range function with the variable f1.num_of_footballs. This indicates that f1 is an object representing the footballs, and **`num_of_footballs`** is a property of this object that holds the number of footballs in the game.

Inside the loop, the program checks if the vertical position of the current football (**`f1.footY[i])`** is greater than **`490.`** If it is, then the game is over. The code sets the vertical position of all of the footballs to **`2000`** and calls a function named **`game_over_text().`** The break statement then exits the loop.

If the game is not over, the program updates the vertical position of the current football by adding the value of **`f1.footY_change[i]`** to it. This suggests that **`footY_change`** is another property of the **`f1 object`** that holds the amount that the vertical position of each football should change on each iteration of the loop. The program then checks if the vertical position of the football has gone below 0 and, if it has, sets the position to 0.

Next, the program calls a function named **isCollision** and passes it the horizontal and vertical positions of the current football and the glove as arguments. This function appears to be responsible for detecting whether the football and glove have collided. If a collision is detected, the code plays a sound effect, resets the vertical position of the glove to **480,** sets the state of the glove to "**ready**", increments the **score** by **1,** and assigns new random horizontal and vertical positions to the football.

Finally, the program calls a function named enemy and passes it the horizontal and vertical positions of the current football and the loop variable i as arguments. It's not clear what this function does, but it's likely that it is responsible for drawing or rendering the football on the screen.

6.4 Glove Movement:

Since the player controls a character named **Messi** who can throw a glove to catch falling footballs, the code is responsible for controlling the movement and behavior of the glove.

```
# Bullet Movement
if g1.gloveY <= 0:
    g1.gloveY = 480
    g1.glove_state = "ready"

if g1.glove_state == "fire":
    throw_glove(g1.gloveX, g1.gloveY)
    g1.gloveY -= g1.gloveY_change

player(m1.messiX, m1.messiY)
show_score(textX, testY)
pygame.display.update()
```

Figure 10.1: Movement of Glove

The program begins by checking if the vertical position of the glove (**g1.gloveY**) is less than or equal to 0. If it is, the code sets the vertical position of the glove to 480 and the state of the glove to "**ready**". This suggests that the glove can be thrown upward, and that its state changes to "**fire**" when it is being thrown. When the glove reaches the top of the screen or goes off the screen, its state is reset to "**ready**" and it appears back at its starting position.

If the state of the glove is "**fire**", the code calls a function named **throw_glove** and passes it the horizontal and vertical positions of the glove as arguments. It's not clear what this function does, but it's likely that it is responsible for drawing or rendering the glove on the screen. The code then updates the vertical position of the glove by subtracting the value of **g1.gloveY**_change from it. This suggests that **gloveY_change** is a property of the g1 object that holds the amount that the vertical position of the glove should change on each iteration of the loop.

Finally, the program calls three more functions: player, **show_score**, and **pygame.display.update**. The player function is likely responsible for drawing or rendering the

player character on the screen. The `show_score` function is likely responsible for displaying the player's score on the screen. The `pygame.display.update` function is a `Pygame` function that updates the contents of the display window. It's likely that these functions are called after the movement of the glove and footballs has been updated so that the changes are reflected on the screen.

7. Module of Program in C++

Now let us have a look how a program will look like if the above Python program is coded in C++ language. The following code represents the creation of classes implemented above in C++.

```
#include <cmath>
#include <cstdlib>
#include <ctime>
#include <iostream>
#include <SDL2/SDL.h>
#include <SDL2/SDL_image.h>
#include <SDL2/SDL_mixer.h>

const int SCREEN_WIDTH = 800;
const int SCREEN_HEIGHT = 600;

class Messi {
public:
SDL_Texture* messiImg;
int messiX;
int messiY;
int messiX_change;

Messi() {
messiImg = NULL;
messiX = 370;
messiY = 480;
messiX_change = 0;
}};

class Football {
public:
SDL_Texture* footImg[6];
int footX[6];
int footY[6];
int footX_change[6];
int footY_change[6];
int num_of_footballs;

Football() {
```

```
for (int i = 0; i < num_of_footballs; i++) {
footImg[i] = NULL;
footX[i] = rand() % 736;
footY[i] = rand() % 100 + 50;
footX_change[i] = 0;
footY_change[i] = 0.125;}
num_of_footballs = 6;
}};

class Glove {
public:
SDL_Texture* gloveImg;
int gloveX;
int gloveY;
int gloveX_change;
int gloveY_change;
std::string glove_state;

Glove() {
gloveImg = NULL;
gloveX = 0;
gloveY = 480;
gloveX_change = 0;
gloveY_change = 2.5;
glove_state = "ready";
}};
```

8. Conclusion

This is a **Pygame** program for a game in which the player controls a character named Messi to catch falling footballs. The game has a background image and plays background music, and the window has a caption and icon. The game has three main classes: Messi, Football, and Glove.

The **Messi** class represents the player character and has properties for the character's image, horizontal and vertical positions, and horizontal position change. The **Football** class represents the enemy footballs and has properties for the football images, horizontal and vertical positions, horizontal and vertical position changes, and the number of footballs. The **Glove** class represents the glove that the player character throws to catch the footballs, and has properties for the glove image, horizontal and vertical positions, horizontal and vertical position changes, and state.

The program also has functions for displaying the score, displaying game over text, drawing the player character, drawing the footballs, throwing the glove, detecting collisions between the footballs and glove, and handling input and game updates. There is a main game loop that runs until the game is quit, and within this loop the script handles events, updates the positions of the player character and footballs, and renders the updated game state on the screen.

9. Targets Achieved

In general, the Pygame library is often used to create simple games or interactive programs that involve graphics, sound, and user input. The code may be attempting to achieve a number of goals, such as creating a functional game loop, rendering game elements on the screen, handling user input, updating the game state, detecting collisions, displaying a score, and displaying game over text.

To achieve these goals, the code may be solving a number of complex engineering problems, such as implementing object-oriented programming principles, using Pygame functions and methods correctly, organizing code into logical and readable structures, and handling events and game updates efficiently. The code may also be solving problems related to game design, such as creating engaging gameplay, designing appropriate game mechanics, and balancing difficulty.

10.Future Recommendations

There are many possible ways to modify or extend our game. Here are a few examples:

- **Add additional gameplay elements:** You could add additional gameplay elements to the game, such as power-ups, boss battles, or level progression.
- **Improve the graphics:** You could improve the graphics of the game by using higher quality images, adding animation, or adding particle effects.
- **Add sound effects:** You could add sound effects to the game to enhance the gameplay experience, such as sound effects for player actions, enemy actions, and collisions.
- **Add more enemies:** You could add more enemy types to the game, each with their own unique behaviors and characteristics.
- **Add a high score system:** You could add a high score system to the game to track the player's best scores and allow them to compete with other players.
- **Add multi-player support:** You could add multi-player support to the game, allowing multiple players to play together online or on the same device.
- **Add mobile support:** You could port the game to mobile platforms, such as Android or iOS, to allow players to play on their phones or tablets.
- **Add VR support:** You could add VR support to the game, allowing players to play in virtual reality using a headset.

These are just a few examples of the many ways in which the game could be modified or extended. The specific changes will depend on the goals and design of the game.

11. CEP Attribute Mapping

11.1 WP1: Depth of Knowledge required

To create the program for our game, a beginner-intermediate level of knowledge in Python programming is required. This includes understanding of basic Python concepts such as variables, loops, conditional statements, functions, and object-oriented programming. Familiarity with Pygame library and its functions for creating games is also necessary to write the given program. Additionally, understanding of basic concepts in game development such as game loops and handling user input would be helpful in writing the program. It would also be helpful to have some understanding of concepts such as image loading, displaying images on screen, and playing sound in Pygame.

11.2 WP2: Range of conflicting requirements

Our program for the game demonstrates diversity in a number of ways. For example:

- The code uses different modules and libraries, such as the math, random, pygame, and mixer modules, to perform a variety of tasks. This demonstrates the use of different libraries and modules to achieve different goals.
- The code uses different data types and structures, such as lists, dictionaries, and classes, to store and manipulate data. This demonstrates the use of different data types and structures to represent different kinds of data and to implement different algorithms.
- The code uses different control structures, such as loops and conditional statements, to control the flow of execution. This demonstrates the use of different control structures to implement different kinds of logic and decision-making in the game.
- The code uses different algorithms and techniques, such as image loading, image rendering, collision detection, and sound playback, to implement various gameplay elements. This demonstrates the use of different algorithms and techniques to solve different problems and to achieve different goals.

Overall, the code demonstrates a diverse set of concepts and techniques that are used in different contexts and situations to create a complex and interactive game.

11.3 WP7: Interdependence

In our program, the different modules (`math`, `random`, `pygame`, and `mixer`) are used to perform specific tasks. The math module is used to perform mathematical operations such as square root and power, which are used in the `isCollision` function to calculate the distance between two objects. The random module is used to generate random numbers, which are used to randomly position the football objects on the screen. The `pygame` module is used to create the game, handle user input, and display graphics on the screen. The mixer module, which is a submodule of pygame, is used to play sound in the game.

The different classes (`Messi`, `Football`, and `Glove`) are used to define objects that are used in the game. The `Messi` class represents the player character, while the Football class represents the enemy objects. The `Glove` class represents the bullets that the player can throw to defeat the enemies. These classes are interdependent because they are all used in the main game loop and interact with each other. For example, the player object moves based on user input and can throw gloves, while the football objects move on their own and can be defeated by the player's gloves.

The functions in the program are also interdependent because they are all called and used in the main game loop. For example, the player function is used to display the player object on the screen, the enemy function is used to display the football objects on the screen, and the `throw_glove` function is used to display the gloves on the screen. The `isCollision` function is used to check if a collision has occurred between the player's gloves and the football objects, and if a collision is detected, the score is updated and the football object is removed from the screen. The `show_score` and `game_over_text` functions are used to display the score and game over message on the screen, respectively.

Hence different modules and sub-tasks have been integrated to achieve our final product.

11.4 WP8: Consequences

Our project can be justified as a learning exercise because it involves several concepts that are commonly used in computer science and software engineering. These concepts include object-oriented programming, event-driven programming, and game development.

In terms of object-oriented programming, the program makes use of classes to define objects and their associated data and behavior. This helps to organize the code and make it more modular and reusable. The use of inheritance and polymorphism is also demonstrated through the use of the pygame library, which is an object-oriented library for game development.

In terms of event-driven programming, the code makes use of a main game loop that continuously waits for and handles user input and other events. This allows the game to respond to user actions and make updates to the game state accordingly.

In terms of game development, the code demonstrates how to create a simple game using the pygame library. It shows how to load and display graphics, handle user input, and play sound. The code also shows how to create and update game objects, and how to detect and handle collisions.

Overall, the knowledge gained from studying the program can be applied to various engineering contexts, such as creating interactive applications, developing games, and building software systems that involve object-oriented design and event-driven programming

Appendix

```
import math
import random

import pygame
from pygame import mixer

# Intialize the pygame
pygame.init()

# create the screen
screen = pygame.display.set_mode((800, 600))

# Background
background = pygame.image.load('background.png')

# Sound
mixer.music.load("background.wav")
mixer.music.play(-1)

# Caption and Icon
pygame.display.set_caption("Space Invader")
icon = pygame.image.load('enemy.png')
pygame.display.set_icon(icon)

# Messi
class Messi:
    messiImg = pygame.image.load('player.png')
    messiX = 370
    messiY = 480
    messiX_change = 0

# Football
class Football:
    footImg = []
    footX = []
    footY = []
    footX_change = []
    footY_change = []
    num_of_footballs = 6
    for i in range(num_of_footballs):
        footImg.append(pygame.image.load('enemy.png'))
        footX.append(random.randint(0, 736))
        footY.append(random.randint(50, 150))
        footX_change.append(0)
```

```
        footY_change.append(0.125)

# Glove

# Ready - You can't see the glove on the screen
# Throw - The glove is currently moving
class Glove:
    gloveImg = pygame.image.load('bullet.png')
    gloveX = 0
    gloveY = 480
    gloveX_change = 0
    gloveY_change = 2.5
    glove_state = "ready"

# Score

score_value = 0
font = pygame.font.Font('freesansbold.ttf', 32)

textX = 10
testY = 10

# Game Over
over_font = pygame.font.Font('freesansbold.ttf', 64)


def show_score(x, y):
    score = font.render("Score : " + str(score_value), True, (255, 255, 255))
    screen.blit(score, (x, y))


def game_over_text():
    over_text = over_font.render("GAME OVER", True, (255, 255, 255))
    screen.blit(over_text, (200, 250))

m1 = Messi()
def player(x, y):
    screen.blit(m1.messiImg, (x, y))

f1 = Football()
def enemy(x, y, i):
    screen.blit(f1.footImg[i], (x, y))

g1 = Glove()
def throw_glove(x, y):
```

```
    global glove_state
    g1.glove_state = "fire"
    screen.blit(g1.gloveImg, (x + 16, y + 20))


def isCollision(footX, footY, gloveX, gloveY):
    distance = math.sqrt(math.pow(footX - gloveX, 2) + (math.pow(footY - gloveY,
2)))
    if distance < 27:
        return True
    else:
        return False


# Game Loop
running = True
while running:

    # RGB = Red, Green, Blue
    screen.fill((0, 0, 0))
    # Background Image
    screen.blit(background, (0, 0))
    for event in pygame.event.get():
        if event.type == pygame.QUIT:
            running = False

        # if keystroke is pressed check whether its right or left
        if event.type == pygame.KEYDOWN:
            if event.key == pygame.K_LEFT:
                m1.messiX_change = -1.5
            if event.key == pygame.K_RIGHT:
                m1.messiX_change = 1.5
            if event.key == pygame.K_SPACE:
                if g1.glove_state is "ready":
                    gloveSound = mixer.Sound("laser.wav")
                    gloveSound.play()
                    # Get the current x cordinate of the spaceship
                    g1.gloveX = m1.messiX
                    throw_glove(g1.gloveX, g1.gloveY)

        if event.type == pygame.KEYUP:
            if event.key == pygame.K_LEFT or event.key == pygame.K_RIGHT:
                m1.messiX_change = 0

    # 5 = 5 + -0.1 -> 5 = 5 - 0.1
```

```
    # 5 = 5 + 0.1

    m1.messiX += m1.messiX_change
    if m1.messiX <= 0:
        m1.messiX = 0
    elif m1.messiX >= 736:
        m1.messiX = 736

    # Enemy Movement
    for i in range(f1.num_of_footballs):
        # Game Over
        if f1.footY[i] > 490:
            for j in range(f1.num_of_footballs):
                f1.footY[j] = 2000
            game_over_text()
            break
        f1.footY[i] += f1.footY_change[i]
        if f1.footY[i] <= 0:
            f1.footY = 0

        # Collision
        collision = isCollision(f1.footX[i], f1.footY[i], g1.gloveX, g1.gloveY)
        if collision:
            explosionSound = mixer.Sound("explosion.wav")
            explosionSound.play()
            g1.gloveY = 480
            g1.glove_state = "ready"
            score_value += 1
            f1.footX[i] = random.randint(0, 736)
            f1.footY[i] = random.randint(50, 150)

        enemy(f1.footX[i], f1.footY[i], i)

    # Bullet Movement
    if g1.gloveY <= 0:
        g1.gloveY = 480
        g1.glove_state = "ready"

    if g1.glove_state == "fire":
        throw_glove(g1.gloveX, g1.gloveY)
        g1.gloveY -= g1.gloveY_change

    player(m1.messiX, m1.messiY)
    show_score(textX, testY)
    pygame.display.update()
```