Software Engineering

UE23CS341A

5th Semester, Academic Year 2023

Date: 13/10/2025

Name: Roshini Ramesh	SRN: PES1UG23CS488	Section: H

LAB 4: VIBE CODING REPORT — PING PONG GAME

1. **Chat / LLM Usage Record**
   - **Chat Link:** https://chatgpt.com/share/68ee9645-4c18-8003-994d-1d03015ce686
   - **Introduction Message in Chat:**
   - Hi, this is Roshini Ramesh. I'm working on a real-time Ping Pong game using Python and Pygame. I have a partially working project structure. Please help me understand how the logic is organized. Review any code I send to ensure it aligns with the expected behavior. the structure looks like this:
     game
     | - ball.py
     | - game_engine.py
     | - paddle.py
     main.py
   - The chat includes discussions about:
     - Setting up the Pygame environment and initializing the mixer safely.
     - Debugging game engine loops and handling collisions.
     - Improving ball movement, speed, and paddle response.
     - Handling sound initialization errors gracefully (pygame.mixer.init() try-except).
     - Managing screen size, colors, and real-time game updates.
2. **Task-wise Implementation and Observations**

   Task 1: Project Setup and Initialization

   - **What Was Done:**
     - Created the base game structure with main.py and a modular game_engine.py.
     - Initialized Pygame and set up display dimensions (800×600).
     - Added exception handling for mixer initialization.

- **Modified Code Snippet (main.py):**

```python
import pygame
from game.game_engine import GameEngine

# Initialize pygame/Start application
pygame.init()

try:
    pygame.mixer.init()
except Exception as e:
    print("Warning: pygame.mixer failed to initialize:", e)

# Screen dimensions
WIDTH, HEIGHT = 800, 600
SCREEN = pygame.display.set_mode((WIDTH, HEIGHT))
pygame.display.set_caption("Ping Pong - Pygame Version")
```

- **Explanation:**
  - pygame.init() initializes all necessary modules.
  - pygame.mixer.init() was wrapped in a try-except to handle audio driver errors gracefully, preventing crashes.
  - Set screen title and resolution for consistent gameplay visuals.
- **Observation:**
  - The game window now initializes smoothly on all systems, even if the audio backend is missing.
  - The first visual output ("Ping Pong - Pygame Version") appears with a black background.

Task 2: Core Game Engine Integration

- **What Was Done:**
  - Created a GameEngine class inside game/game_engine.py that handles the ball, paddles, and collision logic.
  - Integrated game loop from main.py to the GameEngine class for cleaner modularization.
- **Modified Code Snippet:**

  Main.py

```python
import pygame
from game.game_engine import GameEngine

# Initialize pygame/Start application
pygame.init()

try:
    pygame.mixer.init()
except Exception as e:
    print("Warning: pygame.mixer failed to initialize:", e)

# Screen dimensions
WIDTH, HEIGHT = 800, 600
SCREEN = pygame.display.set_mode((WIDTH, HEIGHT))
pygame.display.set_caption("Ping Pong - Pygame Version")

# Colors
WHITE = (255, 255, 255)
BLACK = (0, 0, 0)

# Clock
clock = pygame.time.Clock()
FPS = 60

# Game loop
engine = GameEngine(WIDTH, HEIGHT)

def main():
    running = True
    while running:
        SCREEN.fill(BLACK)
        for event in pygame.event.get():
            if event.type == pygame.QUIT:
                running = False
            # optional: quit via ESC
            elif event.type == pygame.KEYDOWN and event.key == pygame.K_ESCAPE:
                running = False

        engine.handle_input()
        engine.update()
        engine.render(SCREEN)

        pygame.display.flip()
        clock.tick(FPS)

    pygame.quit()

if __name__ == "__main__":
    main()
```

game_engine.py

```
import pygame
from .paddle import Paddle
from .ball import Ball

WHITE = (255, 255, 255)
BLACK = (0, 0, 0)

class GameEngine:
    def __init__(self, width, height):
        self.width = width
        self.height = height
        self.paddle_width = 10
        self.paddle_height = 100

        self.player = Paddle(10, height // 2 - 50, self.paddle_width,
self.paddle_height)
        self.ai = Paddle(width - 20, height // 2 - 50, self.paddle_width,
self.paddle_height)
        self.ball = Ball(width // 2, height // 2, 7, 7, width, height)

        self.player_score = 0
        self.ai_score = 0
        self.font = pygame.font.SysFont("Arial", 30)

        # winning score target (default)
        self.winning_score = 5

        # --- sound setup ---
        # expected folder: sounds/
        self.snd_paddle = None
        self.snd_wall = None
        self.snd_score = None
        try:
            self.snd_paddle = pygame.mixer.Sound("sounds/paddle.mp3")
        except Exception as e:
            print("Warning: could not load paddle.mp3:", e)
        try:
            self.snd_wall = pygame.mixer.Sound("sounds/wall.mp3")
        except Exception as e:
            print("Warning: could not load wall.mp3:", e)
        try:
            self.snd_score = pygame.mixer.Sound("sounds/score.mp3")
        except Exception as e:
            print("Warning: could not load score.mp3:", e)

    def handle_input(self):
        keys = pygame.key.get_pressed()
        if keys[pygame.K_w]:
            self.player.move(-self.player.speed, self.height)
        if keys[pygame.K_s]:
            self.player.move(self.player.speed, self.height)

    def reset_positions(self):
        """Reset paddles to center and ball to center (but keep winning_score)."""
        self.player.y = self.height // 2 - self.paddle_height // 2
        self.ai.y = self.height // 2 - self.paddle_height // 2
        self.player_score = 0
        self.ai_score = 0
        self.ball.reset()

    def update(self):
        # Record previous velocities so we can detect changes and play sounds
        prev_vx = self.ball.velocity_x
        prev_vy = self.ball.velocity_y

        # Move ball
        self.ball.move()

        # Immediately check for paddle collisions after movement
        # (these will change velocity_x when needed)
        scoring_happened = False
        if self.ball.rect().colliderect(self.player.rect()):
            self.ball.x = self.player.x + self.player.width
            self.ball.velocity_x *= -1
        elif self.ball.rect().colliderect(self.ai.rect()):
            self.ball.x = self.ai.x - self.ball.width
            self.ball.velocity_x *= -1

        # Scoring (consider ball width)
        if self.ball.x + self.ball.width < 0:
            # AI scored (left side)
            self.ai_score += 1
            scoring_happened = True
            # Play score sound immediately (don't rely on vx change)
            if self.snd_score:
                self.snd_score.play()
            self.ball.reset()
        elif self.ball.x > self.width:
            # Player scored (right side)
            self.player_score += 1
            scoring_happened = True
            if self.snd_score:
                self.snd_score.play()
            self.ball.reset()

        # AI movement
        self.ai.auto_track(self.ball, self.height)

        # After all movement/collisions/resets, compare velocities and play sounds
        curr_vx = self.ball.velocity_x
        curr_vy = self.ball.velocity_y

        # If vx changed: paddle hit or scoring (score takes precedence)
        if curr_vx != prev_vx:
            if not scoring_happened:
                if self.snd_paddle:
                    self.snd_paddle.play()
            else:
                if self.snd_paddle:
                    self.snd_paddle.play()

        # If vy changed: play wall sound only when the ball actually bounced off
top/bottom
        if curr_vy != prev_vy:
            # detect an actual top/bottom bounce by checking the ball's position
            bounced_top = self.ball.y <= 0
            bounced_bottom = (self.ball.y + self.ball.height) >= self.height

            if bounced_top or bounced_bottom:
                if self.snd_wall:
                    self.snd_wall.play()

        # Check for game over and replay menu if needed
        self.check_game_over(pygame.display.get_surface())

    def render(self, screen):
        # Draw paddles and ball
        pygame.draw.rect(screen, WHITE, self.player.rect())
        pygame.draw.rect(screen, WHITE, self.ai.rect())
        pygame.draw.ellipse(screen, WHITE, self.ball.rect())
        pygame.draw.aaline(screen, WHITE, (self.width // 2, 0), (self.width // 2,
self.height))
```

```
        # Draw score (centered-ish)
        player_text = self.font.render(str(self.player_score), True, WHITE)
        ai_text = self.font.render(str(self.ai_score), True, WHITE)
        screen.blit(player_text, (self.width // 4 - player_text.get_width() // 2, 20))
        screen.blit(ai_text, (self.width * 3 // 4 - ai_text.get_width() // 2, 20))

    # --- game over and replay menu methods (keep the ones you already added) ---
    def check_game_over(self, screen):
        if self.player_score >= self.winning_score or self.ai_score >=
self.winning_score:
            # Show winner message
            big_font = pygame.font.SysFont("Arial", 60)
            if self.player_score >= self.winning_score:
                msg = "Player Wins!"
            else:
                msg = "AI Wins!"

            # Clear screen and draw winner text
            screen.fill(BLACK)
            text = big_font.render(msg, True, WHITE)
            screen.blit(text, (self.width // 2 - text.get_width() // 2, self.height //
3 - text.get_height() // 2))
            pygame.display.flip()

            # small visible pause
            pygame.time.delay(1000)

            # Show replay menu (blocks until user chooses)
            self._show_replay_menu(screen)

    def _show_replay_menu(self, screen):
        clock = pygame.time.Clock()
        menu_font = pygame.font.SysFont("Arial", 28)
        title_font = pygame.font.SysFont("Arial", 44)

        selecting = True
        while selecting:
            for event in pygame.event.get():
                if event.type == pygame.QUIT:
                    pygame.quit()
                    exit()
                elif event.type == pygame.KEYDOWN:
                    if event.key == pygame.K_3:
                        self.winning_score = 3
                        self.reset_positions()
                        selecting = False
                        break
                    elif event.key == pygame.K_5:
                        self.winning_score = 5
                        self.reset_positions()
                        selecting = False
                        break
                    elif event.key == pygame.K_7:
                        self.winning_score = 7
                        self.reset_positions()
                        selecting = False
                        break
                    elif event.key == pygame.K_ESCAPE:
                        pygame.quit()
                        exit()

            # Draw the menu UI
            screen.fill(BLACK)
            title = title_font.render("Play Again?", True, WHITE)
            screen.blit(title, (self.width // 2 - title.get_width() // 2, 60))

            opt1 = menu_font.render("Press 3 : Best of 3", True, WHITE)
            opt2 = menu_font.render("Press 5 : Best of 5", True, WHITE)
            opt3 = menu_font.render("Press 7 : Best of 7", True, WHITE)
            opt4 = menu_font.render("Press ESC : Exit", True, WHITE)

            start_y = self.height // 2 - 40
            spacing = 40
            screen.blit(opt1, (self.width // 2 - opt1.get_width() // 2, start_y))
            screen.blit(opt2, (self.width // 2 - opt2.get_width() // 2, start_y + 
spacing))
            screen.blit(opt3, (self.width // 2 - opt3.get_width() // 2, start_y + 
spacing * 2))
            screen.blit(opt4, (self.width // 2 - opt4.get_width() // 2, start_y + 
spacing * 3))

            pygame.display.flip()
            clock.tick(30)
```

- **Explanation:**
    - The GameEngine class encapsulates all the game logic — paddle movement, ball motion, and score updates.

    - engine.run() manages the main game loop, handling events, updates, and rendering each frame.
- **Observation:**
    - The codebase became cleaner and easier to debug.
    - The gameplay ran at a stable frame rate (~60 FPS).
    - Input lag was reduced after separating rendering and logic updates.

Task 3: Ball and Paddle Collision Handling

- **What Was Done:**
    - Added boundary checks and collision responses in game_engine.py.
    - Implemented paddle reflection logic that modifies ball direction based on where it hits the paddle.
- **Modified Code Snippet:**

```
if self.ball.rect().colliderect(self.player.rect()):
            self.ball.x = self.player.x + self.player.width
            self.ball.velocity_x *= -1
        elif self.ball.rect().colliderect(self.ai.rect()):
            self.ball.x = self.ai.x - self.ball.width
            self.ball.velocity_x *= -1
```

- **Explanation:**
    - colliderect() checks for intersection between the ball and paddle.
    - When collision occurs, the x-velocity inverts (ball_speed_x *= -1) to simulate a bounce.
- **Observation:**
    - Ball properly bounces off both paddles.
    - Increased game responsiveness and interactivity.
    - Added randomness in ball speed slightly for more challenging gameplay.

Task 4: Scoring and Game Reset

- **What Was Done:**
    - Implemented score counters for both players.
    - Reset ball position when a player misses.
    - Displayed real-time scores on screen.
- **Modified Code Snippet:**

```
        if self.ball.x + self.ball.width < 0:
            # AI scored (left side)
            self.ai_score += 1
            scoring_happened = True
            # Play score sound immediately (don't rely on vx change)
            if self.snd_score:
                self.snd_score.play()
            self.ball.reset()
        elif self.ball.x > self.width:
            # Player scored (right side)
            self.player_score += 1
            scoring_happened = True
            if self.snd_score:
                self.snd_score.play()
            self.ball.reset()
```

- **Explanation:**
  - When the ball crosses the screen edges, a point is awarded to the opposing player.
  - reset_ball() re-centers the ball to restart play.
- **Observation:**
  - Smooth transitions between rounds.
  - Score displayed correctly and updated dynamically.
  - The gameplay loop continued seamlessly after each point.

3. **Reflection on the Hardest Task**
   - **Hardest Task:** Debugging and Ensuring Stable Game Performance
   - **Why It Was Difficult:**
     - The most time-consuming challenge was debugging unexpected behavior during gameplay — such as lag spikes, input unresponsiveness, or objects not updating as expected.
     - Pygame doesn't provide detailed runtime error messages for logic-related issues, so tracing the source of problems often required trial and error.
     - Small mistakes, like incorrect indentation or misplaced update calls, could cause major gameplay disruptions.
   - **How I Overcame It:**
     - Used systematic debugging by adding print statements and isolating problem areas.
     - Ran multiple short test sessions after each change instead of large code edits to pinpoint bugs faster.
     - Consulted Pygame documentation and online community discussions to understand typical timing and rendering pitfalls.
     - Gradually stabilized the game through iteration — focusing on making each test run smoother than the last.
4. **Proposed Enhancement (Beyond the Given Tasks)**
   - **Suggested Feature:** Add Multiplayer Mode with Two Separate Player Controls or Online Connectivity.
     - **Idea:**
       - Extend the current local two-player setup to include:
       - **Local Multiplayer:** Two players on the same device using different key bindings (e.g., W/S for Player 1, UP/DOWN for Player 2).
       - **Online Multiplayer:** Use sockets to allow players to connect over LAN or the internet.
   - **Expected Benefit:**
     - Promotes collaboration and competitive gameplay.

        - Expands the project’s learning scope by introducing networking and event synchronization.
        - Enhances replay value and overall fun factor.

5. **Conclusion**

   The Ping Pong game was successfully built using Pygame with a structured modular approach.

    - Through this lab:
        - I learned to manage event loops, rendering, and input handling.
        - Debugged sound and frame synchronization issues.
        - Understood the importance of modular code for scalability.
        - The game now runs smoothly with scoring, paddle movement, and collisions. Future work includes adding multiplayer functionality for both local and online modes.