PCS II PROJECT REPORT

"Networking Handcricket "

SLIDES TO OUR PROJECT:

https://www.canva.com/design/DAGEF_boxw8/nvzbns1mvSW0VsX0aborEA/edit?utm_content=DAGEF_boxw8&utm_campaign=designshare&utm_medium=link2&utm_source=sharebutton

Introduction:

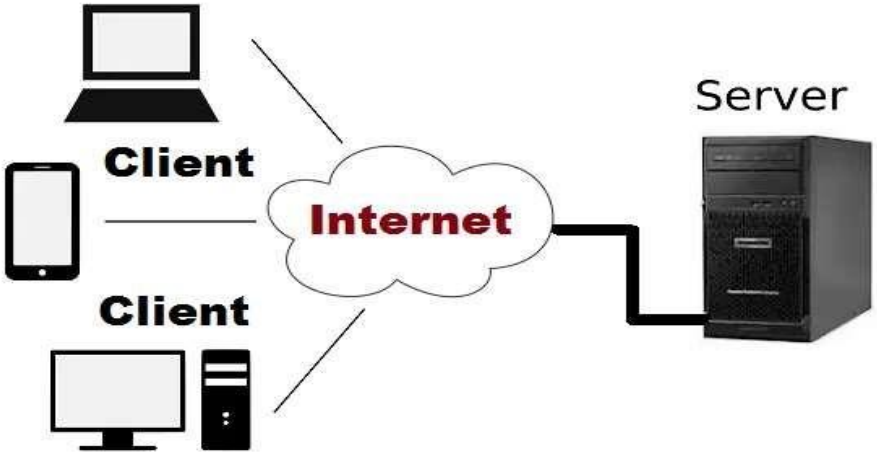
Handcricket is a popular and simple game played by people of all ages. It requires minimal equipment and can be played anytime, anywhere. In this project, we aim to create a digital version of the handcricket game with networking capabilities, allowing players to compete against each other online.

Problem Description:

The goal of this project is to develop a multiplayer handcricket game that enables players to engage in matches over a network. The game should mimic the rules and mechanics of traditional handcricket while providing an intuitive and enjoyable gaming experience for users.

Connectivity of this project to the concepts covered in class:

Client-Server Communication: The game architecture involves a client-server model, where the game client communicates with a central server over a network. Networking protocols such as TCP/IP or UDP/IP are likely used to facilitate this communication, ensuring reliable data transmission between clients and the server.



In the case of this project,who are clients and server??

Client: The client refers to the application running on the player's device (such as a computer or mobile device) that allows them to interact with the handcricket game. Clients send requests to the server, such as requesting to join a match, sending their moves, or retrieving game state updates.

Server: The server is a centralized application responsible for managing game sessions, coordinating communication between clients, and enforcing game rules. It receives requests from clients, processes them, updates the game state accordingly, and sends responses back to the clients.

Where is TCP used in this project??

TCP (Transmission Control Protocol):

TCP is a connection-oriented protocol that provides reliable, ordered, and errorchecked delivery of data packets over a network. In the context of the Handcricket Networking Game, TCP is used for client-server communication to ensure that game data, such as

player moves and game state updates, are reliably transmitted between clients and the server. TCP's built-in mechanisms for error detection, retransmission of lost packets, and flow control are beneficial for maintaining the integrity and consistency of the game data.

Data Synchronization:

In this project,Networking protocols are utilized to synchronize game state and player actions between clients and the server. This ensures that all players have consistent views of the game and that actions performed by one player are correctly reflected for all others participating in the match.

Error Handling and Recovery:

Networking protocols provide mechanisms for error detection, correction, and recovery. In the event of network interruptions or packet loss, the game must handle these issues gracefully to maintain a smooth gaming experience. Protocols such as TCP/IP offer built-in error handling features that help maintain data integrity during transmission.

Implementation of code and explanation in brief:

Four codes are written in order to implement this game namely client.py, server.py, game.py,network.py.Here is the short description of the codes..

server.py:

This code is for the server-side application of the Handcricket Networking Game. It establishes a TCP socket connection, listens for incoming client connections, and manages game sessions between connected clients. The server coordinates gameplay, receives player moves from clients, updates game state, and sends game data back to clients.

client.py:

This code is for the client-side application of the Handcricket Networking Game. It utilizes Pygame for the graphical interface and communicates with the server using the Network class defined in network.py. The client displays the game interface, handles user input, and sends/receives game data to/from the server.

network.py:

This defines the Network class, which handles the client-server communication protocol using TCP sockets. It establishes a connection to the server, sends data, and receives responses. The class also handles encoding and decoding data using pickle for serialization.

game.py:

This contains the definition of the CricketGame class, representing the game logic and state. It defines methods for initializing the game, processing player moves, calculating scores, determining game outcomes (e.g., win, loss, tie), and resetting the game state. These four codes collectively implement the Handcricket Networking Game, enabling multiplayer gameplay over a network where clients connect to a central server to participate in matches.

Here is the brief explanation of each code.....

1)GAME.PY

```
def __init__(self, game_id):
    self.player1_has_played = False
    self.player2_has_played = False
    self.ready_to_start = False
    self.id = game_id
    self.player_moves = [None, None]
    self.player_wins = [0, 0]
    self.ties = 0
    self.scores = [0, 0]
    self.batting_done = [0, 0]
```

- Initializes the game state variables such as whether each player has played, whether the game is ready to start, game ID, player moves, player wins, ties, scores, and batting status.

```
def get_player_move(self, player_index):
    return self.player_moves[player_index]

def get_player_score(self, player_index):
    return self.scores[player_index]
```

- Provides methods to get the move and score of a specific player based on their index.

```
def make_move(self, player_index, move):
    self.player_moves[player_index] = move
    if player_index == 0:
        self.player1_has_played = True
    else:
        self.player2_has_played = True
```

- Updates the player's move in the game state and marks the respective player as having played.

```
def calculate_batsman_score(self, batsman_index, bowler_index, score):
    batsman_move = self.player_moves[batsman_index]
    bowler_move = self.player_moves[bowler_index]

    if batsman_move == bowler_move:
        self.batting_done[batsman_index] = 1
    else:
        score += int(batsman_move)

    return score
```

- Calculates the batsman's score based on their move, the bowler's move, and the current score.

```
def determine_winner(self):
    player1_score = self.scores[0]
    player2_score = self.scores[1]

    if player1_score > player2_score:
        winning_player_index = 0
    elif player1_score < player2_score:
        winning_player_index = 1
    else:
        winning_player_index = -1

    return winning_player_index
```

- Determines the winner based on the scores of both players.

```
def reset_player_moves(self):
    self.player1_has_played = False
    self.player2_has_played = False
```

- Resets the player moves, marking both players as not having played.

2.SERVER.PY

```
import socket
from _thread import *
import pickle
from game import CricketGame
```

- The socket module provides access to networking functions.
- _thread is used for multi-threading support.
- pickle is used for serializing and deserializing Python objects.
- CricketGame is imported from a custom module game, presumably containing the game logic.

```
SERVER_IP = "127.0.0.1"
PORT = 5555
```

- SERVER_IP and PORT specify the IP address and port number the server will bind to.

```
try:
    server_socket.bind((SERVER_IP, PORT))
except socket.error as e:
    print("Failed to bind the server:", str(e))

server_socket.listen()
print("Waiting for a connection... Server is up and running.")
```

- A TCP socket server_socket is created and bound to the specified IP and port.
- The server begins listening for incoming connections.

```
connected_clients = set()
games = {}
game_count = 0
```

- connected_clients: A set to store connected client sockets.
- games: A dictionary to store active game instances.
- game_count: Counter for total game instances initiated.

```
def handle_client_connection(client_conn, player_id, game_id):
```

- This function is responsible for handling client connections and game interactions.

Explanation of while True loop:

- The while True loop ensures continuous communication between the server and the client until the connection is terminated.
- It continuously listens for data sent by the client and responds accordingly.
- The loop breaks if:
- No data is received from the client (if not data).
- The game associated with the client's game ID is not found in the games dictionary.
- An exception occurs during data processing (except block).
- Upon exiting the loop, the function handles cleanup tasks such as removing the game instance from the games dictionary, decrementing the game_count, and closing the client connection.

This loop is crucial for maintaining the client-server communication throughout the lifespan of the client connection, ensuring smooth gameplay interaction and handling unexpected events gracefully.

```
while True:
    client_conn, client_addr = server_socket.accept()
    print("Connected to client:", client_addr)

    game_count += 1
    player_id = 0
    game_id = (game_count - 1) // 2
    print("Current number of games:", game_count)
    if game_count % 2 == 1:
        games[game_id] = CricketGame(game_id)
        print("A new game is created.")
    else:
        print("Second player connected.")
        games[game_id].ready_to_start = True
        player_id = 1

    start_new_thread(handle_client_connection, (client_conn, player_id, game_id))
```

- The server continuously waits for incoming client connections.
- Upon accepting a new connection, it increments the game count and determines player and game IDs.
- If it's the first player in a new game, it creates a new CricketGame instance.
- If it's the second player, it marks the game as ready to start and assigns the player ID.
- A new thread is spawned to handle the client connection.

3.NETWORK.PY

```python
import socket
import pickle
class Network:
    def __init__(self):
        self.client = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
        self.server = "127.0.0.1"
        self.port = 5555
        self.addr = (self.server, self.port)
        self.p = self.connect()

    def get_player_number(self):
        return self.p

    def connect(self):
        try:
            self.client.connect(self.addr)
            return self.client.recv(2048).decode()
        except:
            pass

    def send(self, data):
        try:
            self.client.send(str.encode(data))
            return pickle.loads(self.client.recv(2048 * 2))
        except socket.error as e:
            print(e)
```

1. __init__(self): This is the constructor method. It initializes the socket object (self.client) and sets up the server address (self.server) and port number (self.port). It also calls the connect() method to establish a connection with the server.
2. get_player_number(self): This method returns the player number received from the server after connecting. However, the actual implementation of this method is missing in the provided code.
3. connect(self): This method attempts to connect to the server using the address and port specified in the constructor. If the connection is successful, it receives data from the server (presumably the player number) and returns it. If an exception occurs during the connection process, it silently handles it without providing any error message.

4. send(self, data): This method sends data to the server. It first encodes the data into bytes using UTF-8 encoding (str.encode()), then sends it over the socket. After sending the data, it attempts to receive a response from the server and deserialize it using the pickle.loads() function. If an error occurs during sending or receiving data, it prints the error message but does not provide detailed information about the error.

4.client.PY

```
import pygame
pygame.init()
from network import Network
import pickle

pygame.font.init()
```

- Pygame Initialization: The script begins by importing the Pygame library and initializing it with pygame.init(), which is necessary to use any of the Pygame functionalities.
- Imports: The script imports a custom module network, presumably for handling network operations, and the pickle module for serializing and deserializing Python object structures.
- Font Initialization: pygame.font.init() initializes the font module, which is required to render text onto the screen in Pygame.

```
window_width = 700
window_height = 700
window = pygame.display.set_mode((window_width, window_height))
pygame.display.set_caption("Cricket Game - Client 1")
background_image = pygame.image.load("Untitled.jpg").convert()
```

- Window Settings: The script sets the game window dimensions (700x700) and creates a game window with these dimensions.
- Window Caption: The caption 'Cricket Game - Client 1' is set for the game window, helpful for identifying the game and the client number during multiclient setups.
- Background Image: Loads and converts an image named Untitled.jpg to be used as the background for the game window.

```
class Button:
    def __init__(self, text, x, y, color):
        self.text = text
        self.x = x
        self.y = y
        self.color = color
        self.width = 120
        self.height = 100

    def draw(self, window):
        pygame.draw.rect(window, self.color, (self.x, self.y, self.width, self.height))
        font = pygame.font.SysFont("comicsans", 40)
        text_surface = font.render(self.text, 1, (0, 0, 0))
        window.blit(text_surface, (self.x + round(self.width / 2) - round(text_surface.get_width() / 2),
                                   self.y + round(self.height / 2) - round(text_surface.get_height() / 2)))

    def click(self, pos):
        x1 = pos[0]
        y1 = pos[1]
        if self.x <= x1 <= self.x + self.width and self.y <= y1 <= self.y + self.height:
            return True
        else:
            return False
```

- Class Definition: Button class is created to handle interactive buttons in the game. It includes initialization and methods for rendering the button and detecting clicks.
- Initialization: Stores button parameters such as position, size, text, and color.
- Drawing the Button: Renders the button on the game window, drawing a rectangle and text centered on the button.
- Click Detection: Determines if a mouse click is within the button's boundaries, enabling interaction within the game.

```
def redraw_window(window, game, player_number):
    global text1, text2
    window.fill((255, 255, 0))
    background_image = pygame.image.load("game-backimg.jpeg").convert()
    background_image = pygame.transform.scale(background_image, (window.get_width(), window.get_height()))
    window.blit(background_image, (0, 0))
```

- Fills the window with a yellow color using RGB values.
- Loads the background image from the file "game-backimg.jpeg" and converts it to a format compatible with Pygame.
- Resizes the background image to fit the window size using pygame.transform.scale().
- Blits the background image onto the window surface at coordinates (0, 0).

```
if not game.are_both_players_ready():
    waiting_image = pygame.image.load("wait.jpg").convert()
    waiting_image = pygame.transform.scale(waiting_image, (window_width, window_height))
    window.blit(waiting_image, (0, 0))

    font = pygame.font.Font(None, 36)
    text = font.render("Waiting for another player...", True, (0, 0, 0))
    text_rect = text.get_rect(center=(window_width // 2, window_height // 4 - 50))
    window.blit(text, text_rect)
```

- Checks if both players are not ready. If so, it displays a waiting image and text.
- Loads the "wait.jpg" image and scales it to fit the window size.
- Blits the waiting image onto the window at coordinates (0, 0).
- Renders text to display "Waiting for another player..." at the center of the window.

Else block:

1. Rendering Player Roles and Texts:
    - It renders text to display player roles ("You Are" and "Opponent is") at specific positions on the window.
2. Determining Player Roles:
    - Based on the player_number and game state (game.batting_done), it determines whether the player is a batsman or a bowler and renders corresponding text.
3. Rendering Player Moves or Status:
    - It retrieves player moves and renders them as text on the window. If both players have played their moves, it renders the actual moves. Otherwise, it renders either "Locked In" or "Waiting..." based on whether the player has made their move or not.
4. Rendering Player Scores:
    - It retrieves player scores from the game object and renders them as text on the window.
5. Handling Special Cases (Player Outs):
    - It checks for special cases where a player is out (game.get_player_score(1) == 0) and renders corresponding messages.

6. Rendering Buttons:
    - It iterates over a list of buttons and draws them on the window using the draw method of each button object.
7. Updating Display:
- Finally, it updates the Pygame display to reflect all the changes made in the window.

Main function:

```
def main():
    run = True
    clock = pygame.time.Clock()
    network = Network()
    player_number = int(network.get_player_number())
    print("You are player", player_number)
    while run:
        clock.tick(60)
        try:
            game = network.send("get")
        except:
            run = False
            print("Couldn't get game")
            break

        if game.have_both_players_played():
            redraw_window(window, game, player_number)
            pygame.time.delay(500)
            try:
                game = network.send("score")
            except:
                run = False
                print("Couldn't get game for score")
                break

        if game.batting_done[0] and game.batting_done[1]:
            redraw_window(window, game, player_number)
            pygame.time.delay(500)
            try:
                game = network.send("reset")
            except:
                run = False
                print("Couldn't get game")
                break

            font = pygame.font.SysFont("comicsans", 90)
```

- initialization: The main function initializes essential game components like the window, networking connection, and player identification.
- Game Loop: It enters a continuous loop to maintain the game's interactive nature.
- Clock Management: Manages the game's frame rate using Pygame's clock module.
- Network Communication: Communicates with the server to retrieve the current game state and send player moves.
- Game State Update: Regularly updates the game state by fetching information from the server through the network connection.
- Event Handling: Handles events such as player inputs (mouse clicks) and window updates to ensure responsive gameplay.

- Player Moves: Sends player moves to the server for processing.
- Move Evaluation: When both players have made their moves, it sends the updated game state back to the server for evaluation.
- Result Display: Displays the game result, such as a win, loss, or tie, on the game window.
- End Game Handling: Handles the end of the game, waiting for player input to start a new game or quit.

```
        font = pygame.font.SysFont("comicsans", 90)
        if game.determine_winner() == player_number:
            text = font.render("You Won!", 1, (0,255, 0))
        elif game.determine_winner() == -1:
            text = font.render("Tie Game!", 1, (255, 0, 75))
        else:
            text = font.render("You Lost...", 1, (255, 0, 0))

        window.blit(text, (220, 25))
        pygame.display.update()
        pygame.time.delay(3000)
        run = False

    for event in pygame.event.get():
        if event.type == pygame.QUIT:
            run = False
            pygame.quit()

        if event.type == pygame.MOUSEBUTTONDOWN:
            pos = pygame.mouse.get_pos()
            for button in buttons:
                if button.click(pos) and game.are_both_players_ready():
                    if player_number == 0:
                        if not game.player1_has_played:
                            network.send(button.text)
                            print("Button", button.text, "clicked")
                    else:
                        if not game.player2_has_played:
                            network.send(button.text)
                            print("Button", button.text, "clicked")

    redraw_window(window, game, player_number)
```

- Flow Control: Orchestrates the flow of the game, ensuring smooth transitions between game states.
- Error Handling: Includes error handling mechanisms to manage exceptions and unexpected situations.
- User Interaction: Facilitates user interaction by handling mouse clicks and displaying relevant information on the game window.
- Graphics Rendering: Utilizes Pygame's graphics capabilities to render game elements and provide a visually appealing interface.

- Gameplay Logic: Implements the game's core logic, such as determining winners and managing player turns.
- Networking Protocol: Adheres to the networking protocol established by the server for seamless communication.
- Performance Optimization: Optimizes performance by efficiently managing resources and minimizing unnecessary computations.
- Feedback Mechanism: Provides feedback to the user through graphical elements and text rendering.
- Modularity: Encapsulates different aspects of the game's functionality into separate functions for maintainability and readability.
- Overall Coordination: Coordinates various game components, including networking, graphics, and user interaction, to deliver a cohesive gaming experience.

Menu_screen function:

```
def menu_screen():
    run = True
    clock = pygame.time.Clock()

    while run:
        clock.tick(5)
        window.fill((255, 255, 255))
        font = pygame.font.SysFont("comicsans", 40)
        text = font.render("Click to Play!", 1, (0, 0, 0))
        window.blit(text, (220, 10))
        pygame.display.update()

        for event in pygame.event.get():
            if event.type == pygame.QUIT:
                pygame.quit()
                run = False
            if event.type == pygame.MOUSEBUTTONDOWN:
                run = False
        window.blit(background_image, [70, 100])

        pygame.display.update()

    main()


while True:
    menu_screen()
```

- Initialization: Initializes the game loop (run variable) and a clock to control the frame rate.
- Game Loop:
- Enters a while loop that continues as long as run is True.
- Uses the clock to limit the loop's speed to 5 iterations per second.

- Rendering Menu:
- Fills the window with a white background.
- Renders text ("Click to Play!") using the "comicsans" font with a font size of 40 and black color.
- Blits the rendered text onto the window at coordinates (220, 10).
- Updates the display to show the rendered text.
- Event Handling:
- Checks for user input events.
- If the user clicks the mouse button, it exits the menu screen loop by setting run to False.
- If the user closes the window, it quits Pygame and exits the program.

2. Outer Loop:
    - Enters an infinite loop that continuously calls the menu_screen function.
    - The menu_screen function is repeatedly called, allowing the user to interact with the menu and start the game by clicking the mouse button.

WORKING OF THE CODE:

Working of the code in the slides as well as demo video..

CONCLUSION:

In conclusion, the development of the Cricket Game project involved the creation of multiple Python scripts that collectively form a functional game system. The server-side code facilitates the game's logic, including player moves, scoring, and determining winners. On the client side, the main script manages the game's graphical interface, player interaction, and network communication with the server. Through the utilization of Pygame for graphics rendering and networking modules, the project demonstrates the integration of various programming concepts to create an interactive multiplayer game. Additionally, the implementation of object-oriented programming principles enhances code modularity and maintainability. Overall, the Cricket Game project showcases the application of Python programming skills to develop a comprehensive gaming experience.

REFERENCES:

https://realpython.com/python-sockets/?authuser=0

https://docs.python.org/3/howto/sockets.html

TEAM MEMBERS:

1.KARAN REDDY(B22AI023)

2.ABHIJAN THEJA(B22AI025)