THE APPLICATION OF STACK AND QUEUE DATA STRUCTURES, MERGE SORT, AND FIBONACCI ALGORITHMS IN THE PIXEL PING PONG GAME

A Final Project
Presented to the
Information Technology and Information Systems Department
College of Science
Adamson University

In partial fulfillment
of the requirements for the subject
Data Structure and Algorithm
under the degree of
Bachelor of Science in Information Technology

By

202310937, Garcia Arabelle Eunice D.
202312648, Sanguenza, Keith Angelo D.

2023-2024-2-29015– Lec - TTh– 7:00-8:00 AM,
2023-2024-2-29016– Lab - Monday- 7:00-10:00 AM

Mrs. Gloria Dela Cruz
Professor

May 19, 2024

TABLE OF CONTENTS

PAGES

PROJECT DESCRIPTION AND ANALYSIS

Ping pong is a game well-enjoyed worldwide, and can be accessible to anyone using computers. Ping pong game has taken many twists and turns in its formation to be the quality, high-graphic game that we now play. However, let us go back to its humble beginnings and recreate the black and white, simple and pixelated video game format that we used to love. The project will use a C# visual studio console and the common objectives are to test the skills, accuracy, speed and strategy of our users. With a ball swinging left to right across the screen, the user can try and play so that the ball will never slip past them.

To develop this game, the creator utilized a variety of data structures and algorithms to ensure its functionality and gameplay dynamics. These include modern techniques, ranging from recursive and iterative algorithms to fundamental data structures like arrays, boolean operations, increment and decrement operations, and conditional statements. The updated version introduces new features such as merge sort and a recursive algorithm. These additions enhance the management of game state, player interactions, and score presentation of the game.

With this project, we hope to recreate the core of the classic ping pong experience while also incorporating components that challenge and engage players. Furthermore, this section seeks to explain the project in great detail in order to provide knowledge and explanation for every aspect that contributed to the game's overall success.

First and foremost, to ensure that the game is neat and organized, initial settings for game elements such as player pad sizes, ball placements, and outcome counters are defined. The scroll bars are removed, the foreground color is selected, and the variables are initialized. These variables contain a variety of game states and settings, including pad sizes, ball positions, player scores, and speed.

```
namespace PingPong
{
    0 references
    class Program
    {
        // Initial settings for game elements
        static int firstPlayerPadSize = 10;
        static int secondPlayerPadSize = 4;
        static int ballPositionX = 0;
        static int ballPositionY = 0;
        static bool ballDirectionUp = true;
        static bool ballDirectionRight = false;
        static int firstPlayerPosition = 0;
        static int secondPlayerPosition = 0;
        static int firstPlayerResult = 0;
        static int secondPlayerResult = 0;
        static int speedFactor = 1; // Speed factor for the ball
        static Random randomGenerator = new Random();
        static Queue<bool> aiMoves = new Queue<bool>();
        static Stack<Tuple<int, int>> ballPositionHistory = new Stack<Tuple<int, int>>();
```

randomGenerator is used to generate random numbers.

aiMoves is a queue to store the moves made by the AI-controlled player.

ballPositionHistory is a stack to record the history of ball positions.

The method seen below guarantees that when the game begins, both players' pads are positioned vertically at the center of the console window, while the ball is placed at the center of the window both horizontally and vertically. This provides a balanced starting point for gameplay.

```
static void SetInitialPositions()
{
    firstPlayerPosition = Console.WindowHeight / 2 - firstPlayerPadSize / 2;
    secondPlayerPosition = Console.WindowHeight / 2 - secondPlayerPadSize / 2;
    ballPositionX = Console.WindowWidth / 2;
    ballPositionY = Console.WindowHeight / 2;
}
```

The HandleUserInput function lets the user control the first player's pad in the game. It checks if the user pressed the up or down arrow keys and adjusts the position of the pad accordingly. If the up arrow key is pressed and the pad isn't at the top edge of the console window, the pad moves up. If the down arrow key is pressed and the pad isn't at the bottom edge of the console window, the pad moves down. This function continuously listens for user input, allowing real-time control of the pad's position within the game.

```
// Handle user input for controlling the first player
1 reference
static void HandleUserInput()
{
    if (Console.KeyAvailable)
    {
        ConsoleKeyInfo keyInfo = Console.ReadKey();
        if (keyInfo.Key == ConsoleKey.UpArrow && firstPlayerPosition > 0)
        {
            firstPlayerPosition--;
        }
        if (keyInfo.Key == ConsoleKey.DownArrow && firstPlayerPosition < Console.WindowHeight - firstPlayerPadSize)
        {
            firstPlayerPosition++;
        }
    }
}
```

After getting an input from the first player, the program calls on the second player. The GreedyMoveSecondPlayer method helps the AI control its pad in the game. It's pretty simple: if the ball is above the middle of the pad and the pad isn't at the top of the window, the pad moves up to catch the ball, and vice versa.

```
1 reference
static void GreedyMoveSecondPlayer()
{
    if (ballPositionY < secondPlayerPosition + secondPlayerPadSize / 2 && secondPlayerPosition > 0)
    {
        secondPlayerPosition--;
    }
    else if (ballPositionY > secondPlayerPosition + secondPlayerPadSize / 2 && secondPlayerPosition < Console.WindowHeight - secondPlayerPadSize)
    {
        secondPlayerPosition++;
    }
}
```

The MoveBall method governs the behavior of the ball within the game. It manages the ball's movement, collision detection, and scoring mechanisms. When the ball collides with the top or bottom walls of the game window, its vertical direction changes, simulating bouncing. If the ball exits the game window on the left or right sides, it resets to the center, updating the score

```
static void MoveBall()
{
    ballPositionHistory.Push(new Tuple<int, int>(ballPositionX, ballPositionY));

    // Bounce the ball if it hits top or bottom walls
    if (ballPositionY == 0 || ballPositionY == Console.WindowHeight - 1)
    {
        ballDirectionUp = !ballDirectionUp;
    }

    // Reset positions and update score if ball goes out of bounds
    if (ballPositionX == Console.WindowWidth - 1)
    {
        SetInitialPositions();
        ballDirectionRight = false;
        ballDirectionUp = true;
        firstPlayerResult++;
        speedFactor++; // Increase speed factor when first player wins
    }

    if (ballPositionX == 0)
    {
        SetInitialPositions();
        ballDirectionRight = true;
        ballDirectionUp = true;
        secondPlayerResult++;
        speedFactor++; // Increase speed factor when second player wins
    }
```

accordingly. Furthermore, the method handles collisions with player paddles, adjusting the ball's direction accordingly. Ultimately, it updates the ball's position based on its direction to facilitate its movement across the game window.

The ball's horizontal position, ballPositionX, serves as the scoring mechanism in the game. When it equals Console.WindowWidth - 1, it indicates that the ball has collided with the right wall, signaling a missed opportunity for the second player. Consequently, the game resets, and the first player earns a point. The same occurs if the ball equals zero and the second player scores a point. From then on, whenever a player scores a point, the speedFactor is incremented, effectively making the ball move faster in subsequent rounds.

```
// Bounce the ball if it hits first player's pad
if (ballPositionX < 3 && ballPositionY >= firstPlayerPosition && ballPositionY < firstPlayerPosition + firstPlayerPadSize)
{
    ballDirectionRight = true;
}

// Bounce the ball if it hits second player's pad
if (ballPositionX >= Console.WindowWidth - 3 - 1 && ballPositionY >= secondPlayerPosition && ballPositionY < secondPlayerPosition + secondPlayerPadSize)
{
    ballDirectionRight = false;
}
```

This is the PrintResult method, which displays the current game results on the console screen. It can be found on the center of the top row of the console window and outputs the scores of both players. The game then checks to see if either player has reached 5 points, indicating the end of the game. It picks the winner and it has a corresponding message for the players.

```
static void PrintResult()
{
    Console.SetCursorPosition(Console.WindowWidth / 2 - 1, 0);
    Console.Write("{0}-{1}", firstPlayerResult, secondPlayerResult);

    // Check if either player has reached 5 points
    if (firstPlayerResult == 5 || secondPlayerResult == 5)
    {
        // Determine the winner and display a message accordingly
        if (firstPlayerResult == 5)
        {
            Console.SetCursorPosition(Console.WindowWidth / 2 - 15, Console.WindowHeight / 2);
            Console.WriteLine("Congratulations! You won!");
        }
        else
        {
            Console.SetCursorPosition(Console.WindowWidth / 2 - 15, Console.WindowHeight / 2);
            Console.WriteLine("You suck! Play better next time!");
        }
```

In addition, it sorts the scores using the merge sort technique and shows the scores. It also computes and displays the 10th Fibonacci number using recursion. Finally, it calls the EndGame function to finish the game.

PROJECT OBJECTIVES

Our goal is to create an engaging Ping Pong game using coding techniques. Key objectives include establishing player controls, developing AI opponents, and to overall guarantee fluid gaming mechanics. Hence, these objectives below are formulated to outline our goals.

1. To Simulate a Ping Pong Game

   The goal is to emulate the fundamental principles of a Ping Pong game in a virtual setting. It involves putting the score system, player paddles, ball movement, and collision detection between the paddles and ball into practice. The focus is on faithfully replicating the mechanics and dynamics of actual Ping Pong gameplay to provide players an authentic and engaging experience.

2. To Handle User Input for Player Movement

   Ensuring smooth and intuitive control over the player's paddle is pivotal for an enjoyable gaming experience. This goal centers on crafting a responsive input mechanism that seamlessly translates player actions into paddle movements via keyboard controls. This entails recognizing and interpreting user keystrokes, translating them into paddle movements, and guaranteeing fluid and accurate paddle positioning.

3. To Implement a Simple AI for the Second Player

   In single-player mode, the game requires an AI opponent to challenge the user. This challenge entails creating an AI algorithm that controls the movement of the second player's paddle. To make clever paddle moves, the AI examines the game state, taking into account parameters such as the position of the ball and the opponent's paddle. The goal is to ensure that the AI provides a demanding but fair gaming experience for the gamer while being balanced and fair in its gameplay behavior.

ORIGINAL GAME OVERVIEW

The Ping Pong game we found uses simple data structures like arrays and boolean flags, as well as methods for random number generation and conditional logic, to produce an engaging gameplay experience on the console. These parts work together perfectly to govern game mechanics, player interactions, and ball movement, resulting in an interesting and challenging gameplay experience. Now, let's recall its basic parts– two paddles on opposite sides, a ball, and two players to play the game.

In this game, he used arrays to create paddles. These arrays store each paddle's vertical position on the console screen. For example, *the DrawFirstPlayer and DrawSecondPlayer methods iterate through these arrays to draw the corresponding paddles on screen.*

```
static void DrawFirstPlayer()
{
    for (int y = firstPlayerPosition; y < firstPlayerPosition + firstPlayerPadSize; y++)
    {
        PrintAtPosition(0, y, '|');
        PrintAtPosition(1, y, '|');
    }
}

static void PrintAtPosition(int x, int y, char symbol)
{
    Console.SetCursorPosition(x, y);
    Console.Write(symbol);
}

static void DrawSecondPlayer()
{
    for (int y = secondPlayerPosition; y < secondPlayerPosition + secondPlayerPadSize; y++)
    {
        PrintAtPosition(Console.WindowWidth - 1, y, '|');
        PrintAtPosition(Console.WindowWidth - 2, y, '|');
    }
}
```

```
private static void MoveBall()
{
    if (ballPositionY == 0)
    {
        ballDirectionUp = false;
    }
    if (ballPositionY == Console.WindowHeight - 1)
    {
        ballDirectionUp = true;
    }
    if (ballPositionX == Console.WindowWidth - 1)
    {
        SetBallAtTheMiddleOfTheGameField();
        ballDirectionRight = false;
        ballDirectionUp = true;
        firstPlayerResult++;
        Console.SetCursorPosition(Console.WindowWidth / 2, Console.WindowHeight / 2);
        Console.WriteLine("First player wins!");
        Console.ReadKey();
    }
```

It also uses conditional expressions to control ball movement and collisions. The MoveBall method *assesses the ball's position and modifies its direction in response to collisions with the console window's edges and player paddles.* If the ball reaches the top or bottom edge, the ballDirectionUp flag is toggled to reverse the vertical direction.

The MoveBall() method is called to update the position and movement of the ball based on its current trajectory and any collisions with the player paddles or edges of the console window. Within each iteration of the loop, the program checks if there's any input available from

the player using Console.KeyAvailable. If there is input, it reads the key pressed using Console.ReadKey() and checks if it corresponds to the up or down arrow keys. If the up arrow key is pressed, the MoveFirstPlayerUp() method is called to move the first player's paddle up. If the down arrow key is pressed, the MoveFirstPlayerDown() method is called to move the first player's paddle down.

After processing player input, the program calls the SecondPlayerAIMove() method to simulate movement for the second player's paddle based on the current state of the game.

```
static void SecondPlayerAIMove()
{
    int randomNumber = randomGenerator.Next(1, 101);
    //if (randomNumber == 0)
    //{
    //    MoveSecondPlayerUp();
    //}
    //if (randomNumber == 1)
    //{
    //    MoveSecondPlayerDown();
    //}
    if (randomNumber <= 70)
    {
        if (ballDirectionUp == true)
        {
            MoveSecondPlayerUp();
        }
        else
        {
            MoveSecondPlayerDown();
        }
    }
}
```

The game uses the.NET framework's Random class to create random numbers for simulating AI movement. The randomGenerator object generates random integers, which are then utilized by the SecondPlayerAIMove function *to determine whether the second player's (which is an AI) paddle moves up or down.*

Boolean flags (ballDirectionUp and ballDirectionRight) are used to track the direction of the ball's movement. These flags control the vertical and horizontal movement of the ball, determining whether it moves up, down, left, or right.

```
private static void MoveBall()
{
    if (ballPositionY == 0)
    {
        ballDirectionUp = false;
    }
    if (ballPositionY == Console.WindowHeight - 1)
    {
        ballDirectionUp = true;
    }
    if (ballPositionX == Console.WindowWidth - 1)
    {
        SetBallAtTheMiddleOfTheGameField();
        ballDirectionRight = false;
        ballDirectionUp = true;
        firstPlayerResult++;
        Console.SetCursorPosition(Console.WindowWidth / 2, Console.WindowHeight / 2);
        Console.WriteLine("First player wins!");
        Console.ReadKey();
    }
    if (ballPositionX == 0)
    {
        SetBallAtTheMiddleOfTheGameField();
        ballDirectionRight = true;
        ballDirectionUp = true;
```

After updating the game state, the console screen is cleared using Console.Clear() to prepare for the next frame. The game elements (player paddles, ball, and score) are then redrawn on the screen using the DrawFirstPlayer(), DrawSecondPlayer(), DrawBall(), and PrintResult() methods.

The primary game loop, implemented using a while loop, continuously updates the game state and renders the screen.

RE-CODING PROCESS

The re-coding of the Ping Pong game entailed a comprehensive strategy of enhancements and refactoring aimed at boosting functionality, performance, and maintainability. Key priorities included optimizing the game loop, refining player and ball movement mechanics, and integrating novel features to augment complexity and entertainment value. This iterative process began with a meticulous analysis of the existing codebase to pinpoint areas ripe for improvement, paving the way for a systematic implementation of tailored adjustments.

One of the first steps in the re-coding process was to simplify and consolidate the game initialization and setup procedures. This included setting initial positions for the players and the ball, as well as configuring the console settings to remove scroll bars and set the foreground color. By centralizing these setup tasks, the code became easier to read and maintain.

To handle user input more efficiently, the re-coded version uses a non-blocking approach that checks for key presses without pausing the game loop. This ensures that the game remains responsive to player actions while maintaining a consistent frame rate. The logic for moving the players was refined to prevent the paddles from moving outside the console window, enhancing the game's playability.

A significant addition to the re-coded version is the introduction of a speed factor for the ball, which increases as the game progresses. This adds a layer of challenge, as players must adapt to the increasing speed of the ball. The implementation of this feature required modifications to the ball movement logic and the game loop timing.

The AI for the second player was also enhanced. Instead of making random movements, the AI now queues movements based on the ball's position, making it more competitive. This was achieved by using a queue to store the AI's intended movements, which are then executed in sequence. This approach simulates a more strategic behavior pattern for the AI player. **(Refer to the screenshot below)**

```
1 reference
static void MoveBall()
{
    ballPositionHistory.Push(new Tuple<int, int>(ballPositionX, ballPositionY));

    if (ballPositionY == 0 || ballPositionY == Console.WindowHeight - 1)
    {
        ballDirectionUp = !ballDirectionUp;
    }

    if (ballPositionX == Console.WindowWidth - 1)
    {
        SetInitialPositions();
        ballDirectionRight = false;
        ballDirectionUp = true;
        firstPlayerResult++;
        speedFactor++; // Increase speed factor when first player wins
    }

    if (ballPositionX == 0)
    {
        SetInitialPositions();
        ballDirectionRight = true;
        ballDirectionUp = true;
        secondPlayerResult++;
        speedFactor++; // Increase speed factor when second player wins
    }

    if (ballPositionX < 3 && ballPositionY >= firstPlayerPosition && ballPositionY < firstPlayerPosition + firstPlayerPadSize)
    {
        ballDirectionRight = true;
    }

    if (ballPositionX >= Console.WindowWidth - 3 - 1 && ballPositionY >= secondPlayerPosition && ballPositionY < secondPlayerPosition + secondPlayerPadSize)
    {
        ballDirectionRight = false;
    }

    ballPositionX += ballDirectionRight ? 1 : -1;
    ballPositionY += ballDirectionUp ? -1 : 1;
}
```

For the ball movement, a history of positions was introduced using a stack. This allows for potential future features, such as undoing movements or analyzing the ball's trajectory. The movement logic was carefully adjusted to ensure that the ball bounces off the walls and paddles correctly, with conditions to check for collisions and to update the ball's direction accordingly. **(Refer to the screenshot below)**

```
1 reference
static void MoveBall()
{
    ballPositionHistory.Push(new Tuple<int, int>(ballPositionX, ballPositionY));

    if (ballPositionY == 0 || ballPositionY == Console.WindowHeight - 1)
    {
        ballDirectionUp = !ballDirectionUp;
    }
```

A notable addition to the code was the divide and conquer sorting, like the Merge Sort algorithm, efficiently sorts lists by breaking them into smaller sublists, recursively sorting each, and merging them. This method minimizes comparisons, achieving O(n log n) time complexity. Meanwhile, the recursive Fibonacci algorithm computes Fibonacci numbers by breaking the

problem into smaller instances until reaching a base case, though it may be less efficient for large n due to recursion overhead. Despite potential inefficiencies, both approaches offer elegant solutions to their respective problems, showcasing the versatility of these computational techniques.

Merge Sort divides lists, sorts them, then merges them, achieving O(n log n) time complexity. The recursive Fibonacci algorithm computes Fibonacci numbers by breaking the problem into smaller instances until reaching a base case. While recursion offers an elegant solution, it may suffer inefficiencies for large n. Nonetheless, both techniques exemplify powerful computational approaches. **(Refer to the screenshot below)**

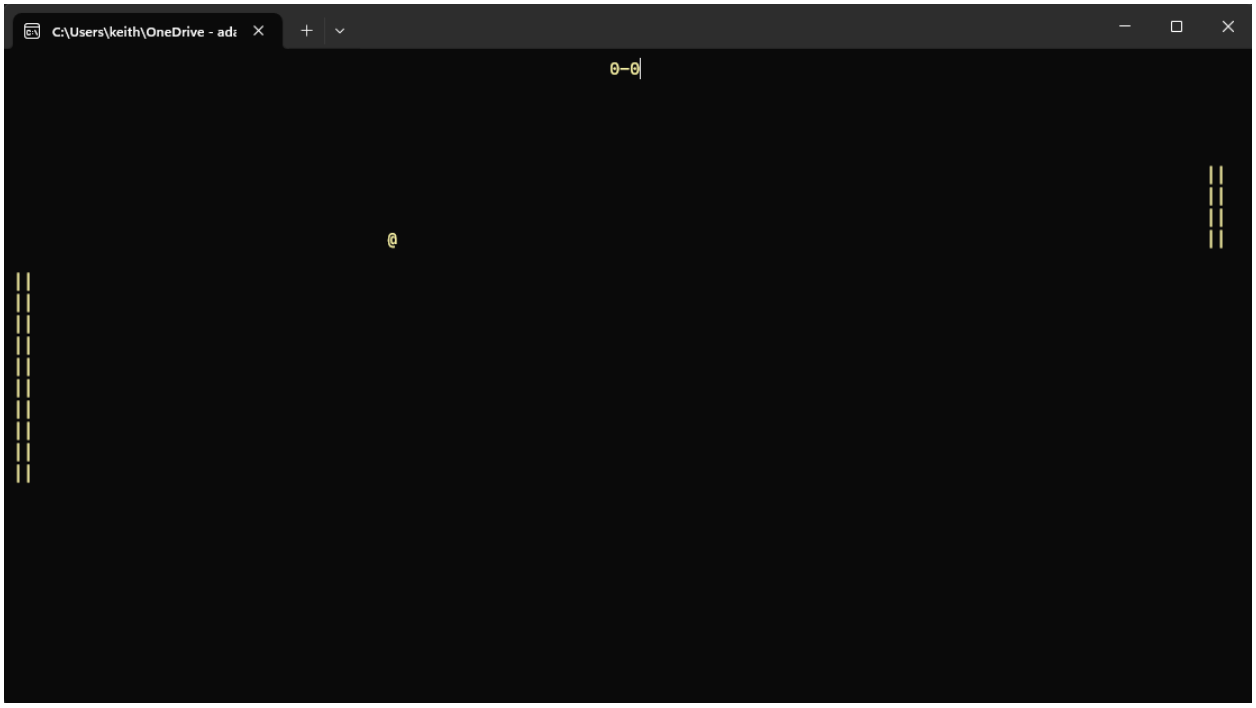
```
// Divide and Conquer: Merge Sort
3 references
static void MergeSort(int[] array, int left, int right)
{
    if (left < right)
    {
        int mid = (left + right) / 2;
        MergeSort(array, left, mid);
        MergeSort(array, mid + 1, right);
        Merge(array, left, mid, right);
    }
}

1 reference
static void Merge(int[] array, int left, int mid, int right)
{
    int n1 = mid - left + 1;
    int n2 = right - mid;
    int[] leftArray = new int[n1];
    int[] rightArray = new int[n2];
    Array.Copy(array, left, leftArray, 0, n1);
    Array.Copy(array, mid + 1, rightArray, 0, n2);
    int i = 0, j = 0, k = left;
    while (i < n1 && j < n2)
    {
        if (leftArray[i] <= rightArray[j])
        {
            array[k] = leftArray[i];
            i++;
        }
        else
        {
            array[k] = rightArray[j];
            j++;
        }
        k++;
    }
    while (i < n1)
    {
        array[k] = leftArray[i];
        i++;
        k++;
    }
    while (j < n2)
    {
        array[k] = rightArray[j];
        j++;
        k++;
    }
```

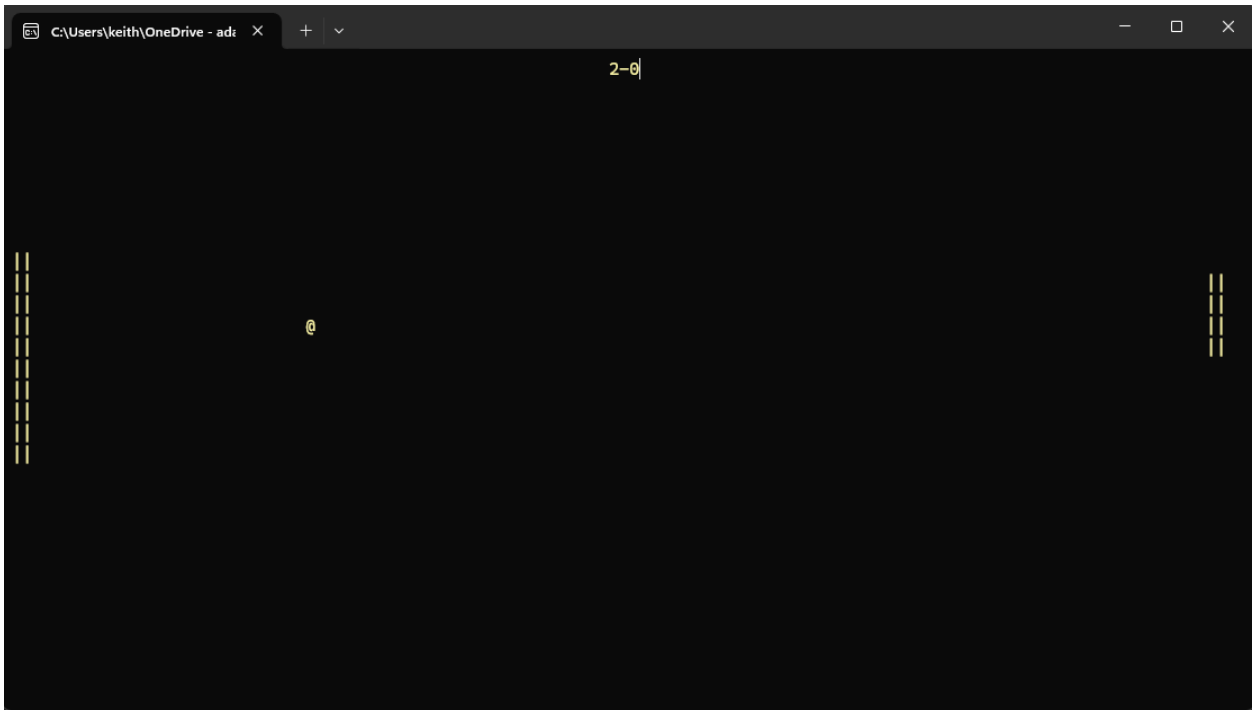
```
// Recursive Algorithm: Fibonacci Sequence
3 references
static int Fibonacci(int n)
{
    if (n <= 1) return n;
    return Fibonacci(n - 1) + Fibonacci(n - 2);
}
```

Throughout the re-coding process, emphasis was placed on writing clean, well-organized code. This involved using descriptive variable names, removing commented-out code that was no longer needed, and organizing the code into logical sections. These practices not only make the code more readable but also facilitate easier updates and maintenance in the future.

In essence, the re-coding approach for the Ping Pong game prioritized enhancing functionality, refining code organization, and introducing compelling new features to elevate user engagement. Through the application of contemporary coding methodologies and meticulous refinement of game mechanics, the revamped version delivers an enriched user experience, all while preserving the timeless essence of the classic Ping Pong game.

SCREEN SHOTS OF THE GAME (IN GAME)





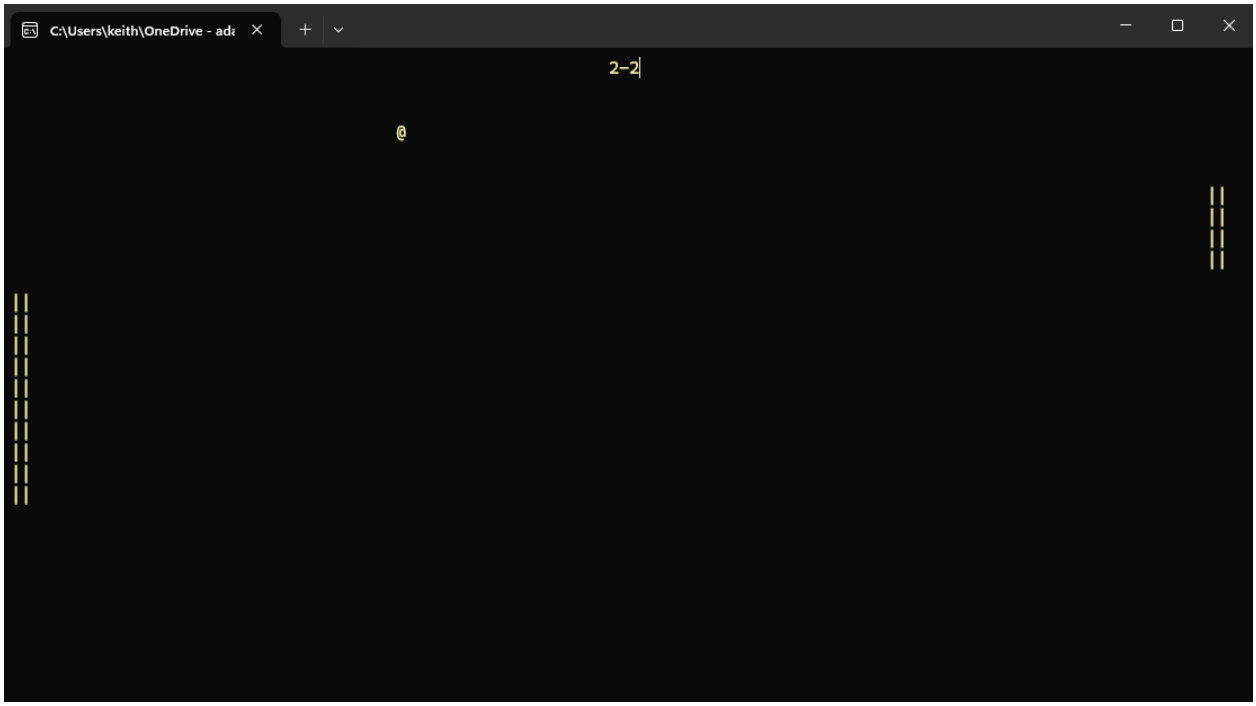


Figure 1. In game

Figure 2. Player 1 wins the game

Figure 3. When player 1 loses the game

SOURCE CODE LISTING

```
// Initial settings for game elements
static int firstPlayerPadSize = 10;
static int secondPlayerPadSize = 4;
static int ballPositionX = 0;
static int ballPositionY = 0;
static bool ballDirectionUp = true;
static bool ballDirectionRight = false;
static int firstPlayerPosition = 0;
static int secondPlayerPosition = 0;
static int firstPlayerResult = 0;
static int secondPlayerResult = 0;
static int speedFactor = 1; // Speed factor for the ball
static Random randomGenerator = new Random();
static Queue<bool> aiMoves = new Queue<bool>();
static Stack<Tuple<int, int>> ballPositionHistory = new Stack<Tuple<int, int>>();

0 references
static void Main(string[] args)
{
    // Set console settings
    Console.ForegroundColor = ConsoleColor.Yellow;
    Console.BufferHeight = Console.WindowHeight;
    Console.BufferWidth = Console.WindowWidth;

    // Set initial positions of game elements
    SetInitialPositions();

    // Main game loop
    while (true)
    {
        // Handle user input
        HandleUserInput();

        // Move the AI-controlled second player
        GreedyMoveSecondPlayer();

        // Move the ball
        MoveBall();

        // Draw the game on the console
        DrawGame();

        // Print current game results
        PrintResult();
```

Figure 4. The game variables

```
        // Pause for a short time to control game speed
        Thread.Sleep(60 / speedFactor);
    }
}

// Set initial positions of players and the ball
3 references
static void SetInitialPositions()
{
    firstPlayerPosition = Console.WindowHeight / 2 - firstPlayerPadSize / 2;
    secondPlayerPosition = Console.WindowHeight / 2 - secondPlayerPadSize / 2;
    ballPositionX = Console.WindowWidth / 2;
    ballPositionY = Console.WindowHeight / 2;
}

// Handle user input for controlling the first player
1 reference
static void HandleUserInput()
{
    if (Console.KeyAvailable)
    {
        ConsoleKeyInfo keyInfo = Console.ReadKey();
        if (keyInfo.Key == ConsoleKey.UpArrow && firstPlayerPosition > 0)
        {
            firstPlayerPosition--;
        }
        if (keyInfo.Key == ConsoleKey.DownArrow && firstPlayerPosition < Console.WindowHeight - firstPlayerPadSize)
        {
            firstPlayerPosition++;
        }
    }
}
```

Figure 5. The initial positions of the ball and the method for user input.

```
// Greedy algorithm to move the AI-controlled second player
1 reference
static void GreedyMoveSecondPlayer()
{
    if (ballPositionY < secondPlayerPosition + secondPlayerPadSize / 2 && secondPlayerPosition > 0)
    {
        secondPlayerPosition--;
    }
    else if (ballPositionY > secondPlayerPosition + secondPlayerPadSize / 2 && secondPlayerPosition < Console.WindowHeight - secondPlayerPadSize)
    {
        secondPlayerPosition++;
    }
}

// Move the ball and handle collisions
1 reference
static void MoveBall()
{
    ballPositionHistory.Push(new Tuple<int, int>(ballPositionX, ballPositionY));

    // Bounce the ball if it hits top or bottom walls
    if (ballPositionY == 0 || ballPositionY == Console.WindowHeight - 1)
    {
        ballDirectionUp = !ballDirectionUp;
    }

    // Reset positions and update score if ball goes out of bounds
    if (ballPositionX == Console.WindowWidth - 1)
    {
        SetInitialPositions();
        ballDirectionRight = false;
        ballDirectionUp = true;
        firstPlayerResult++;
        speedFactor++; // Increase speed factor when first player wins
    }

    if (ballPositionX == 0)
    {
        SetInitialPositions();
        ballDirectionRight = true;
        ballDirectionUp = true;
        secondPlayerResult++;
        speedFactor++; // Increase speed factor when second player wins
    }
```

Figure 6. The code construct for the AI second player and the ball's path across the screen

```
    // Bounce the ball if it hits first player's pad
    if (ballPositionX < 3 && ballPositionY >= firstPlayerPosition && ballPositionY < firstPlayerPosition + firstPlayerPadSize)
    {
        ballDirectionRight = true;
    }

    // Bounce the ball if it hits second player's pad
    if (ballPositionX >= Console.WindowWidth - 3 - 1 && ballPositionY >= secondPlayerPosition && ballPositionY < secondPlayerPosition + secondPlayerPadSize)
    {
        ballDirectionRight = false;
    }

    // Update ball position
    ballPositionX += ballDirectionRight ? 1 : -1;
    ballPositionY += ballDirectionUp ? -1 : 1;
}

// Draw the game elements on the console
```

```
    // Draw the game elements on the console
    1 reference
    static void DrawGame()
    {
        Console.Clear();
        for (int y = firstPlayerPosition; y < firstPlayerPosition + firstPlayerPadSize; y++)
        {
            PrintAtPosition(0, y, '|');
            PrintAtPosition(1, y, '|');
        }
        for (int y = secondPlayerPosition; y < secondPlayerPosition + secondPlayerPadSize; y++)
        {
            PrintAtPosition(Console.WindowWidth - 1, y, '|');
            PrintAtPosition(Console.WindowWidth - 2, y, '|');
        }
        PrintAtPosition(ballPositionX, ballPositionY, '@');
    }
```

Figure 7. The paddles from opposite sides of the screen.

```
// Print the current game results
1 reference
static void PrintResult()
{
    Console.SetCursorPosition(Console.WindowWidth / 2 - 1, 0);
    Console.Write("{0}-{1}", firstPlayerResult, secondPlayerResult);

    // Check if either player has reached 5 points
    if (firstPlayerResult == 5 || secondPlayerResult == 5)
    {
        // Determine the winner and display a message accordingly
        if (firstPlayerResult == 5)
        {
            Console.SetCursorPosition(Console.WindowWidth / 2 - 15, Console.WindowHeight / 2);
            Console.WriteLine("Congratulations! You won!");
        }
        else
        {
            Console.SetCursorPosition(Console.WindowWidth / 2 - 15, Console.WindowHeight / 2);
            Console.WriteLine("You suck! Play better next time!");
        }
```

Figure 8. The scores and the corresponding message for the winner/loser.

```
        // Display sorted scores using divide and conquer (merge sort)
        int[] scores = { firstPlayerResult, secondPlayerResult };
        MergeSort(scores, 0, scores.Length - 1);
        Console.WriteLine("Sorted Scores: {0}, {1}", scores[0], scores[1]);

        // Display the 10th Fibonacci number using recursion
        int fibNumber = 10;
        Console.WriteLine("Fibonacci (Recursive): {0}", Fibonacci(fibNumber));

        // End the game
        EndGame();
    }
}
```

Figure 9. Display of sorted scores

```
// Method to end the game after displaying the final message
1 reference
static void EndGame()
{
    Console.SetCursorPosition(Console.WindowWidth / 2 - 15, Console.WindowHeight / 2 + 1);
    Console.WriteLine("Press any key to exit...");
    Console.ReadKey();
    Environment.Exit(0); // Exit the game
}

// Print a character at a specified position on the console
5 references
static void PrintAtPosition(int x, int y, char symbol)
{
    Console.SetCursorPosition(x, y);
    Console.Write(symbol);
}

// Divide and Conquer: Merge Sort
3 references
static void MergeSort(int[] array, int left, int right)
{
    if (left < right)
    {
        int mid = (left + right) / 2;
        MergeSort(array, left, mid);
        MergeSort(array, mid + 1, right);
        Merge(array, left, mid, right);
    }
}
```

Figure 10. Method to close the game

```
// Divide and Conquer: Merge Sort
3 references
static void MergeSort(int[] array, int left, int right)
{
    if (left < right)
    {
        int mid = (left + right) / 2;
        MergeSort(array, left, mid);
        MergeSort(array, mid + 1, right);
        Merge(array, left, mid, right);
    }
}

1 reference
static void Merge(int[] array, int left, int mid, int right)
{
    int n1 = mid - left + 1;
    int n2 = right - mid;
    int[] leftArray = new int[n1];
    int[] rightArray = new int[n2];
    Array.Copy(array, left, leftArray, 0, n1);
    Array.Copy(array, mid + 1, rightArray, 0, n2);
    int i = 0, j = 0, k = left;
    while (i < n1 && j < n2)
    {
        if (leftArray[i] <= rightArray[j])
        {
            array[k] = leftArray[i];
            i++;
        }
        else
        {
            array[k] = rightArray[j];
            j++;
        }
        k++;
    }
    while (i < n1)
    {
        array[k] = leftArray[i];
        i++;
        k++;
    }
```

Figure 11. The Merge Sort Algorithm

```
        while (j < n2)
        {
            array[k] = rightArray[j];
            j++;
            k++;
        }
    }

    // Recursive Algorithm: Fibonacci Sequence
    3 references
    static int Fibonacci(int n)
    {
        if (n <= 1) return n;
        return Fibonacci(n - 1) + Fibonacci(n - 2);
    }
```

Figure 12. The Fibonacci Sequence

FLOWCHART

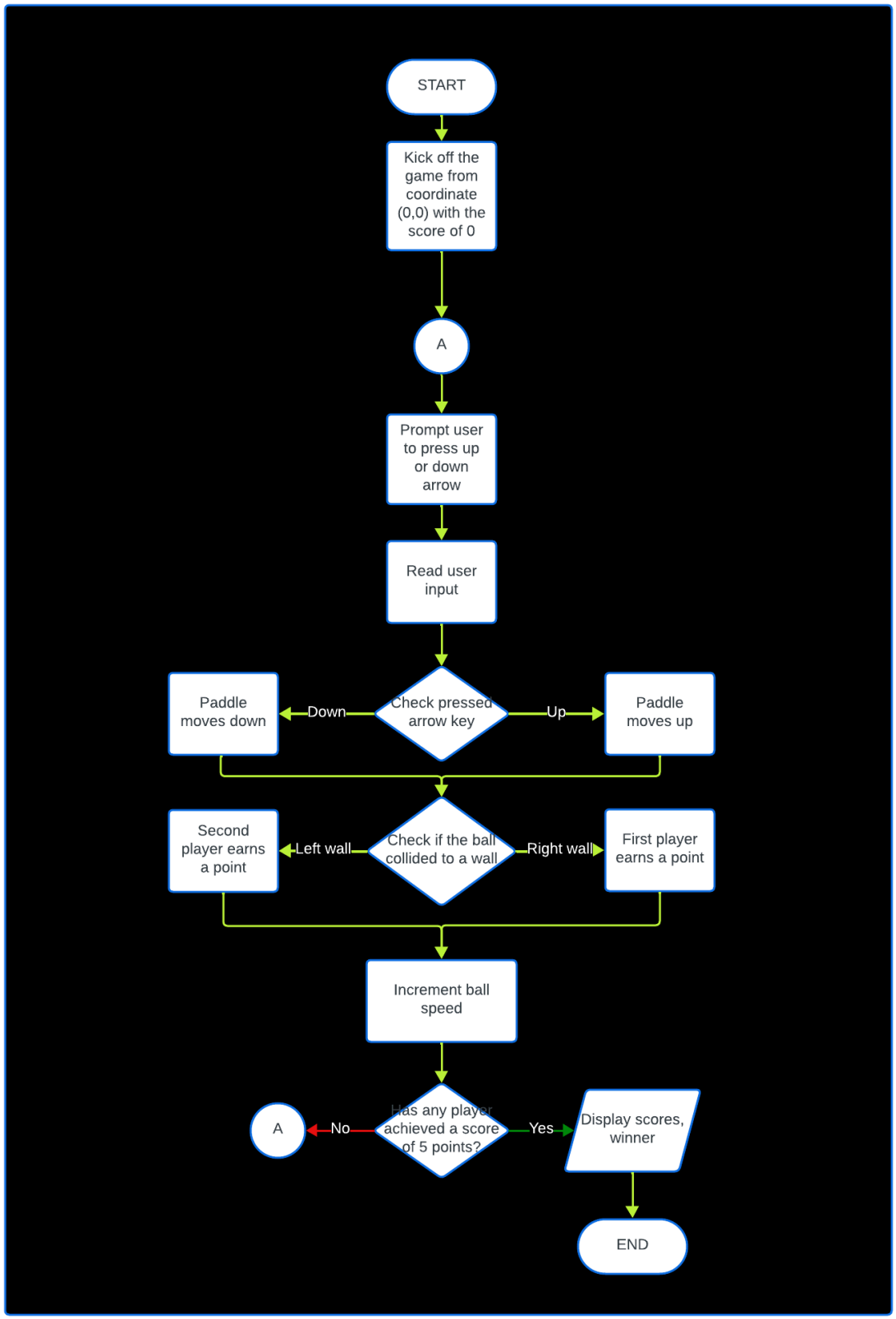

START
Kick off the game from coordinate (0,0) with the score of 0
A
Prompt user to press up or down arrow
Read user input
Check pressed arrow key
Down
Up
Paddle moves down
Paddle moves up
Check if the ball collided to a wall
Left wall
Right wall
Second player earns a point
First player earns a point
Increment ball speed
Has any player achieved a score of 5 points?
No
Yes
A
Display scores, winner
END

Figure 6. The program flowchart

LEARNINGS

During the course of re coding and analysis, several crucial learnings that are integral to the field of IT have emerged. This PingPong game code teaches us a lot about how to make a game using computer programming, especially using a language called C#.

First, the code shows us that setting up the game is really important. This includes deciding things like how big the player's paddles are, where the ball starts, and how fast everything moves. These settings make the game work and can be changed to make the game harder or easier.

Second, the code uses loops and "if" statements to control how the game works. There's a main loop that keeps the game going until someone wins. Inside this loop, there are different tasks like checking what buttons the player is pressing, moving the computer player, moving the ball, drawing everything on the screen, detecting collisions with the sides and the ball, and indicating who's winning. "If" statements are used to decide things like which way the ball should bounce.

Third, the code uses a simple computer player, or AI, to play against the human player. The AI is programmed to move towards the ball, which makes the game more fun and challenging.

Fourth, the code uses things called Queue and Stacks to remember what's happened in the game. The AI's moves are stored in a Queue, and the places the ball has been are stored in a Stack. These can be used to do things like take back moves or play the game again.

Finally, the code uses some more advanced programming ideas to control how fast the game goes and to do some math calculations. These aren't directly related to the game, but they

show how flexible C# is and how it can be used for different things in programming.

In short, this PingPong game code is a great way to learn about making a game in C#, including setting up the game, controlling how it works, using AI, remembering what's happened, and using advanced programming ideas. It's a really useful tool for anyone who wants to learn about making games or programming in C#.

RESOURCES

CSharpConsoleGames/PingPong/Program.cs at master · NikolayIT/CSharpConsoleGames. (2024). GitHub. https://github.com/NikolayIT/CSharpConsoleGames/blob/master/PingPong/Program.cs

Code, B. (2021). Learn Merge Sort in 13 minutes 🔪 [YouTube Video]. In *YouTube*. https://www.youtube.com/watch?v=3j0SWDX4AtU

Halverson, S. (2022). How To Code The Fibonacci Sequence In C# | Programming Tutorial For Beginners [YouTube Video]. In YouTube. https://www.youtube.com/watch?v=m8BAZ0L_cYU