**Section 10: Lesson 45**

**Building a Game Window with Python Turtle - Line by Line Analysis**

**Lecturer: Ahmed Sami**

**Introduction**

Good morning, students. Welcome to Lesson 45 of Section 10. Today, we will undertake a detailed exploration of game window initialization in Python, examining each line of code with precision and understanding its significance in the broader context of game development.

This lecture represents a crucial step in your journey from writing simple scripts to creating interactive applications. We will dissect a foundational program that establishes a game window for a Ping Pong game, analyzing every statement to understand not just what it does, but why it is essential.

By the conclusion of this session, you will possess a comprehensive understanding of window creation, configuration, and the game loop mechanism. This knowledge will serve as the cornerstone for all future game development projects you undertake.

**Body**

**Part 1: Module Import and Foundation**

**Line 1: import turtle**

This statement imports the turtle graphics module into our program. The turtle module is part of Python's standard library, meaning it comes pre-installed with Python. This module provides a virtual canvas and drawing tools that we can programmatically control. When we import turtle, we gain access to all its classes and functions, including the Screen class we will use momentarily. This single line establishes our connection to Python's graphics capabilities.

**Part 2: Window Creation and Object Instantiation**

**Line 2: wind = turtle.Screen()**

This line performs object instantiation, creating a new Screen object and assigning it to the variable wind. The Screen() constructor initializes a graphics window that will serve as our game's display surface. The variable name wind (short for window) becomes our reference point for all subsequent window operations. This object encapsulates all the properties and methods needed to manage our game's visual presentation.

**Part 3: Window Configuration and Customization**

**Line 3: wind.title("Ping Pong By Codezilla")**

The title() method sets the text that appears in the window's title bar. This string "Ping Pong By Codezilla" identifies our application to users and appears at the top of the game window. Beyond mere labeling, a descriptive title enhances user experience and provides immediate context about the application's purpose. The title also appears in the operating system's taskbar, making it easier to identify when multiple applications are running.

**Line 4: wind.bgcolor("black")**

The bgcolor() method (background color) establishes the window's background color. By setting it to "black", we create a classic arcade game aesthetic that provides excellent contrast for game elements. The black background reduces eye strain during extended play sessions and ensures that lighter-colored game objects will be clearly visible. The turtle module accepts color names as strings, making it intuitive to specify colors.

**Line 5: wind.setup(width=800, height=600)**

This method call defines the precise dimensions of our game window. The width parameter sets the horizontal span to 800 pixels, while the height parameter establishes a vertical span of 600 pixels. These dimensions create a 4:3 aspect ratio, a traditional format for games that provides ample playing space while remaining manageable on most displays. The use of keyword arguments (width= and height=) makes the code self-documenting and prevents confusion about parameter order.

**Part 4: Performance Optimization**

**Line 6: wind.tracer(0)**

This seemingly simple line implements a crucial optimization technique. The tracer() method controls screen update behavior. By passing 0 as an argument, we completely disable automatic screen refreshing. In the default mode, turtle updates the display after every drawing operation, which can cause flickering and reduce performance. By disabling automatic updates, we gain manual control over when the screen refreshes, allowing us to batch multiple changes and update once per frame, resulting in smoother animation.

**Part 5: The Game Loop Implementation**

**Line 7-8: while True: and wind.update()**

The while True: statement initiates an infinite loop, the heart of our game's execution. This loop will continue indefinitely, processing game logic and rendering frames until the program is explicitly terminated. Within this loop, wind.update() manually refreshes the display to show any changes made since the last update. This manual update approach, combined with the disabled tracer, gives us precise control over our game's frame rate and ensures smooth, flicker-free animation.

The indentation of wind.update() beneath the while statement is critical—it indicates that this statement belongs to the loop body and will execute repeatedly. This continuous cycle of updating forms the game loop, a fundamental pattern in game programming where the program continuously processes input, updates game state, and renders the current frame.

**Conclusion**

Today's comprehensive analysis has revealed the intricate details behind what might initially appear as simple setup code. We have examined how each line contributes to creating a robust foundation for game development, from importing necessary modules to implementing the game loop that drives all interactive applications.

The concepts we have explored—object instantiation, method invocation, parameter passing, and loop construction—are fundamental not only to game development but to all Python programming. The window configuration techniques and performance optimizations you have learned today will prove invaluable as you develop more complex applications.

As you progress in your game development journey, remember that understanding the purpose behind each line of code transforms you from someone who merely copies examples to a developer who can create original solutions. The game window we have constructed today will serve as the canvas for adding paddles, balls, scoring systems, and game logic in subsequent lessons.

I encourage you to experiment with these concepts: modify the window dimensions, try different background colors, and observe how the tracer setting affects performance. Through experimentation and practice, you will develop an intuitive understanding of these fundamental concepts.

**Python Code Tested**

python

Download

Copy code

Wrap

*# Import the turtle module, which provides graphics and drawing capabilities in Python*

*import* turtle

*# Create a window (screen) object using turtle.Screen() to set up the game environment*

wind = turtle.Screen()

*# Set the title of the window to "Ping Pong By Codezilla"*

wind.title("Ping Pong By Codezilla")

*# Set the background color of the window to black*

wind.bgcolor("black")

*# Set up the dimensions of the window: width=800 pixels, height=600 pixels*

wind.setup(*width*=800, *height*=600)

*# Disable automatic screen updates (tracer(0) means updates must be manual for better performance in games)*

wind.tracer(0)

*# Start an infinite loop to keep the game running and updating the screen*

*while* True:

*# Manually update the screen to reflect any changes (since tracer is off)*

wind.update()

*End of Lesson 45*

**Section 10: Lesson 46**

**Adding Game Elements - Paddles and Ball**

**Lecturer: Ahmed Sami**

**Introduction**

Good morning, dear students. Welcome to Lesson 46 of Section 10. Having established our game window in the previous lesson, it is now time to breathe life into our game by adding the essential interactive elements: the paddles and the ball.

In this lecture, we will learn how to create, configure, and position game objects on the screen. We will explore important concepts such as creating Turtle objects, customizing their shapes and colors, and determining their precise positions in the game space. These skills will enable you to transform an empty window into a genuine interactive playing environment.

By the end of this lesson, you will have a deep understanding of how to construct and organize visual game elements, and you will be prepared for the next step: adding movement and interaction to these elements.

**Body**

**Part 1: Setting Up the Foundation Environment (Lines 1-7)**

**Line 1: import turtle**

We begin by importing the turtle graphics library. This library provides us with the necessary tools to create visual elements and animations in Python. This single import statement opens the door to all the graphical capabilities we need for our game.

**Line 2: wind = turtle.Screen()**

Creating the main game window that will contain all game elements. This wind object will be the stage upon which all our elements perform. The Screen() constructor initializes a graphical window that serves as our game's display surface.

**Line 3: wind.title("Ping Pong By Codezilla")**

Setting the window title that appears in the title bar, giving our game a clear identity. This title not only labels our application but also appears in the operating system's taskbar, making it easily identifiable among multiple running applications.

**Line 4: wind.bgcolor("black")**

Specifying black as the window's background color, creating perfect contrast with the white and colored elements. The black background reduces eye strain during extended play sessions and ensures maximum visibility for game objects.

**Line 5: wind.setup(width=800, height=600)**

Defining the window dimensions with a width of 800 pixels and height of 600 pixels, providing an appropriate playing area. These dimensions create a 4:3 aspect ratio, a traditional format that offers ample space while remaining manageable on most displays.

**Line 6: wind.tracer(0)**

Disabling automatic screen updates to improve performance and gain manual control over the frame rate. This optimization technique allows us to batch multiple changes and update once per frame, resulting in smoother animation.

**Part 2: Creating the First Paddle - Left Player (Lines 8-14)**

**Line 8: madrab1 = turtle.Turtle()**

Creating a new Turtle object for the first paddle. This object will represent the left player's paddle in the game. The Turtle class provides all the methods needed to control appearance and behavior.

**Line 9: madrab1.speed(0)**

Setting the drawing speed to 0, which is the fastest possible speed. This means the paddle will move instantly without animation delay, ensuring responsive gameplay.

**Line 10: madrab1.shape("square")**

Defining the basic shape of the paddle as a square. We will transform this square into a rectangle in the following line to create the appropriate paddle shape.

**Line 11: madrab1.color("blue")**

Coloring the first paddle blue to distinguish it from the second paddle and add visual aesthetics to the game. Color coding helps players quickly identify their controls.

**Line 12: madrab1.shapesize(stretch\_wid=5, stretch\_len=1)**

Transforming the square into a vertical rectangle by stretching the width 5 times and the length once. This creates the typical paddle shape for a Ping Pong game, providing sufficient surface area for ball contact.

**Line 13: madrab1.penup()**

Lifting the pen to prevent drawing lines when moving the paddle. This is essential because we want to move the paddle without leaving a trail on the screen.

**Line 14: madrab1.goto(-350, 0)**

Positioning the first paddle on the left side of the screen at coordinates (x=-350, y=0), placing it at the middle height on the left edge. This position leaves adequate space from the window edge while maximizing the playing area.

**Part 3: Creating the Second Paddle - Right Player (Lines 16-22)**

**Line 16: madrab2 = turtle.Turtle()**

Creating a second Turtle object for the right paddle, similar to the first paddle but for the second player. This maintains consistency in object creation patterns.

**Line 17: madrab2.speed(0)**

Setting the fastest drawing speed, matching the first paddle to ensure equal responsiveness for both players.

**Line 18: madrab2.shape("square")**

Using the square shape as a base, exactly like the first paddle, maintaining uniformity in the initial shape selection.

**Line 19: madrab2.color("red")**

Coloring the second paddle red to visually distinguish it from the blue paddle, making it easy for players to track their respective paddles during gameplay.

**Line 20: madrab2.shapesize(stretch\_wid=5, stretch\_len=1)**

Applying the same geometric transformation to create the rectangular paddle shape, ensuring both paddles have identical dimensions.

**Line 21: madrab2.penup()**

Lifting the pen to prevent drawing paths during movement, maintaining a clean playing field.

**Line 22: madrab2.goto(350, 0)**

Positioning the second paddle on the right side of the screen at (x=350, y=0), directly opposite the first paddle, creating a symmetrical playing field.

**Part 4: Creating the Game Ball (Lines 24-29)**

**Line 24: madra2 = turtle.Turtle()**

Creating a Turtle object for the ball. Note that the name madra2 might be a typo and could be better named as ball or kora, but we will use the existing name in the code.

**Line 25: madra2.speed(0)**

Setting the fastest speed for the ball to ensure smooth and responsive movement during gameplay.

**Line 26: madra2.shape("square")**

Using the square shape for the ball. At the default size, a small square appears nearly circular, making it suitable for representing a ball.

**Line 27: madra2.color("white")**

Coloring the ball white to provide maximum contrast with the black background and ensure clear visibility throughout the game.

**Line 28: madra2.penup()**

Lifting the pen to prevent drawing the ball's path as it moves across the screen, keeping the display clean and uncluttered.

**Line 29: madra2.goto(0, 0)**

Positioning the ball at the center of the screen at the origin point (x=0, y=0), which is the ideal starting position for the game.

**Part 5: Main Game Loop (Lines 31-32)**

**Line 31: while True:**

Initiating the infinite loop that keeps the game running continuously. This loop will execute indefinitely, processing game logic and rendering frames until the program is explicitly terminated.

**Line 32: wind.update()**

Manually updating the screen to display any changes in object positions. This manual update works in conjunction with tracer(0) to provide precise control over the frame rate and ensure smooth, flicker-free animation.

**Conclusion**

In this lesson, we have taken a significant step forward in building our Ping Pong game. We have learned how to create and configure the essential game objects: two paddles for the players and a ball for gameplay. Each object has been carefully customized in terms of shape, color, size, and position.

The concepts we covered today—creating multiple Turtle objects, customizing their properties, and controlling their positions—form the foundation for any two-dimensional game. We have seen how a few lines of code can transform an empty window into a playing field ready for action.

In the next lesson, we will bring these static elements to life by implementing movement functions and keyboard controls, enabling players to control the paddles and initiate ball movement. Continue practicing and experimenting, and remember that every great game started with simple elements like those we created today.

I encourage you to experiment with changing colors, sizes, and positions to gain a deeper understanding of how each parameter affects the game's appearance. Practice and experimentation are the keys to mastery in game programming.

**Python Code Tested**

python

Download

Copy code

Wrap

*import* turtle *# Import turtle graphics library for creating visual elements and animations*

wind = turtle.Screen() *# Create the main game window where all game objects will be displayed*

wind.title("Ping Pong By Codezilla") *# Set the text that appears in the window's title bar*

wind.bgcolor("black") *# Set the window's background color to black for a classic game look*

wind.setup(*width*=800, *height*=600) *# Define window dimensions: 800 pixels wide by 600 pixels tall*

wind.tracer(0) *# Turn off automatic screen updates for manual control and better performance*

*# First paddle (left player)*

madrab1 = turtle.Turtle() *# Create a turtle object that will become the left paddle*

madrab1.speed(0) *# Set animation speed to fastest (0 = no animation delay)*

madrab1.shape("square") *# Use square as the base shape for the paddle*

madrab1.color("blue") *# Color the left paddle blue*

madrab1.shapesize(*stretch\_wid*=5, *stretch\_len*=1) *# Transform square into a 5x1 rectangle (paddle shape)*

madrab1.penup() *# Lift the pen to prevent drawing lines when moving*

madrab1.goto(-350, 0) *# Position paddle on the left side of screen at x=-350, y=0*

*# Second paddle (right player)*

madrab2 = turtle.Turtle() *# Create a turtle object that will become the right paddle*

madrab2.speed(0) *# Set animation speed to fastest (0 = no animation delay)*

madrab2.shape("square") *# Use square as the base shape for the paddle*

madrab2.color("red") *# Color the right paddle red*

madrab2.shapesize(*stretch\_wid*=5, *stretch\_len*=1) *# Transform square into a 5x1 rectangle (paddle shape)*

madrab2.penup() *# Lift the pen to prevent drawing lines when moving*

madrab2.goto(350, 0) *# Position paddle on the right side of screen at x=350, y=0*

*# Game ball*

madra2 = turtle.Turtle() *# Create a turtle object that will be the game ball*

madra2.speed(0) *# Set animation speed to fastest for smooth ball movement*

madra2.shape("square") *# Use square shape (appears circular at default size)*

madra2.color("white") *# Color the ball white for visibility against black background*

madra2.penup() *# Lift the pen to prevent drawing lines when moving*

madra2.goto(0, 0) *# Start the ball at the center of the screen (x=0, y=0)*

*while* True: *# Main game loop - runs forever to keep the game active*

wind.update() *# Refresh the screen to show any changes in object positions*

*End of Lesson 46*

**Section 10: Lesson 47**

**Implementing Paddle Movement and Keyboard Controls**

**Lecturer: Ahmed Sami**

**Introduction**

Good morning, dear students. Welcome to Lesson 47 of Section 10. Today marks a pivotal moment in our game development journey as we transform our static game elements into interactive, responsive components that react to player input.

In this lecture, we will explore the fundamental concepts of user interaction in game programming. We will learn how to create movement functions, capture keyboard input, and bind specific keys to game actions. These skills represent the bridge between a static display and an interactive gaming experience.

By the end of this lesson, you will understand how to implement controlled movement for game objects and establish a responsive control system that allows players to interact with your game in real-time. This knowledge is essential for creating any interactive application, not just games.

**Body**

**Part 1: Foundation and Window Setup (Lines 1-17)**

**Lines 1-17: Basic Setup**

These lines remain identical to our previous lesson, establishing the game window and its properties. We import the turtle module, create the game window with dimensions of 800x600 pixels, set a black background, and disable automatic screen updates for optimal performance. This foundation provides the canvas upon which our interactive elements will operate.

**Part 2: Creating Game Objects (Lines 18-40)**

**Lines 18-24: First Paddle Creation**

We create the first paddle (madrab1) with the same properties as before: blue color, rectangular shape (5x1 stretch), and positioned at x=-350. The consistency in object creation ensures predictable behavior across our game elements.

**Lines 26-32: Second Paddle Creation**

The second paddle (madrab2) mirrors the first but with red coloring and positioned at x=350. The symmetrical setup creates a balanced playing field for both players.

**Lines 34-40: Ball Creation with Improved Naming**

Notice the significant improvement in naming convention. The ball object is now properly named ball instead of the ambiguous madra2 from our previous lesson. This semantic naming makes our code more readable and maintainable. The ball retains its white color and central position at (0, 0).

**Part 3: Movement Functions (Lines 42-57)**

**Lines 42-45: madrab1\_up() Function**

python

Download

Copy code

Wrap

def madrab1\_up():

y = madrab1.ycor()

y += 20

madrab1.sety(y)

This function implements upward movement for the first paddle. Let's analyze each line:

* Line 43: ycor() retrieves the current y-coordinate of madrab1
* Line 44: Increases the y-coordinate by 20 pixels
* Line 45: sety() updates the paddle's position to the new y-coordinate

The increment of 20 pixels provides smooth yet responsive movement—not too slow to be frustrating, nor too fast to lose control.

**Lines 47-50: madrab1\_down() Function**

python

Download

Copy code

Wrap

def madrab1\_down():

y = madrab1.ycor()

y -= 20

madrab1.sety(y)

This function mirrors the upward movement but decreases the y-coordinate by 20 pixels, moving the paddle downward. The symmetrical movement distance ensures consistent gameplay feel.

**Lines 52-55: madrab2\_up() Function**

python

Download

Copy code

Wrap

def madrab2\_up():

y = madrab2.ycor()

y += 20

madrab2.sety(y)

Identical logic applies to the second paddle's upward movement. Maintaining consistent movement patterns between paddles ensures fair gameplay for both players.

**Lines 57-60: madrab2\_down() Function**

python

Download

Copy code

Wrap

def madrab2\_down():

y = madrab2.ycor()

y -= 20

madrab2.sety(y)

Completes the movement functions by providing downward movement for the second paddle. The parallel structure in all four functions demonstrates good programming practice through consistency and predictability.

**Part 4: Keyboard Event Binding (Lines 62-67)**

**Line 62: wind.listen()**

This crucial command activates the window's ability to receive keyboard input. Without this line, all our keyboard bindings would be ineffective. It essentially tells the program to start paying attention to keyboard events.

**Line 63: wind.onkeypress(madrab1\_up, "w")**

This line creates an event binding that connects the 'w' key to the madrab1\_up function. When a player presses 'w', the function executes, moving the left paddle upward. The choice of 'w' follows the standard WASD control scheme familiar to gamers.

**Line 64: wind.onkeypress(madrab1\_down, "s")**

Binds the 's' key to downward movement for the left paddle. Together with 'w', this creates intuitive vertical control for the left player.

**Line 65: wind.onkeypress(madrab2\_up, "Up")**

Assigns the Up arrow key to move the right paddle upward. Using arrow keys for the second player is a classic control scheme that allows two players to share a keyboard comfortably.

**Line 66: wind.onkeypress(madrab2\_down, "Down")**

Completes the control scheme by binding the Down arrow key to the right paddle's downward movement. The arrow keys provide intuitive control that matches the visual direction of movement.

**Part 5: Main Game Loop (Lines 69-71)**

**Lines 69-71: The Eternal Loop**

python

Download

Copy code

Wrap

*while* True:

wind.update()

The infinite loop continues to run, now with added significance. Each iteration not only updates the display but also processes any keyboard events that have occurred. The wind.update() call refreshes the screen to show paddle movements resulting from player input, creating the responsive gameplay experience.

**Conclusion**

Today's lesson has transformed our static game display into an interactive experience. We have successfully implemented movement functions that modify object positions and created a keyboard control system that responds to player input in real-time.

The concepts we explored—function definition, coordinate manipulation, event-driven programming, and input binding—are fundamental to all interactive software development. These patterns appear not just in games but in any application that requires user interaction.

Key takeaways from this lesson include:

1. Functions encapsulate repeated behaviors, making code modular and maintainable
2. The coordinate system allows precise control over object positioning
3. Event-driven programming enables responsive user interfaces
4. Consistent movement increments ensure smooth, predictable gameplay

In our next lesson, we will add ball movement and collision detection, bringing us closer to a fully functional Ping Pong game. The paddle controls we implemented today will then serve their true purpose—intercepting and redirecting the ball.

I encourage you to experiment with different movement speeds, try alternative key bindings, and observe how these changes affect gameplay feel. Remember that game development is as much about user experience as it is about technical implementation.

**Python Code Tested**

python

Download

Copy code

Wrap

*# Import the turtle module, which provides graphics and drawing capabilities in Python*

*import* turtle

*# Create a window (screen) object using turtle.Screen() to set up the game environment*

wind = turtle.Screen()

*# Set the title of the window to "Ping Pong By Codezilla"*

wind.title("Ping Pong By Codezilla")

*# Set the background color of the window to black*

wind.bgcolor("black")

*# Set up the dimensions of the window: width=800 pixels, height=600 pixels*

wind.setup(*width*=800, *height*=600)

*# Disable automatic screen updates (tracer(0) means updates must be manual for better performance in games)*

wind.tracer(0)

*# madrab1*

madrab1 = turtle.Turtle() *# Create a turtle object for paddle 1*

madrab1.speed(0) *# Set the animation speed to maximum (0 = fastest)*

madrab1.shape("square") *# Set the shape to square*

madrab1.color("blue") *# Set the color to blue*

madrab1.shapesize(*stretch\_wid*=5, *stretch\_len*=1) *# Stretch the square vertically by 5 and horizontally by 1*

madrab1.penup() *# Lift the pen up to prevent drawing lines when moving*

madrab1.goto(-350, 0) *# Position the paddle at x=-350, y=0*

*# madrab2*

madrab2 = turtle.Turtle() *# Create a turtle object for paddle 2*

madrab2.speed(0) *# Set the animation speed to maximum (0 = fastest)*

madrab2.shape("square") *# Set the shape to square*

madrab2.color("red") *# Set the color to red*

madrab2.shapesize(*stretch\_wid*=5, *stretch\_len*=1) *# Stretch the square vertically by 5 and horizontally by 1*

madrab2.penup() *# Lift the pen up to prevent drawing lines when moving*

madrab2.goto(350, 0) *# Position the paddle at x=350, y=0*

*# ball*

ball = turtle.Turtle() *# Create a turtle object for the ball*

ball.speed(0) *# Set the animation speed to maximum (0 = fastest)*

ball.shape("square") *# Set the shape to square*

ball.color("white") *# Set the color to white*

ball.penup() *# Lift the pen up to prevent drawing lines when moving*

ball.goto(0, 0) *# Position the ball at the center (x=0, y=0)*

*# functions*

def madrab1\_up(): *# Define function to move paddle 1 up*

y = madrab1.ycor() *# Get the current y coordinate of paddle 1*

y += 20 *# Increase y by 20*

madrab1.sety(y) *# Set the new y coordinate*

def madrab1\_down(): *# Define function to move paddle 1 down*

y = madrab1.ycor() *# Get the current y coordinate of paddle 1*

y -= 20 *# Decrease y by 20*

madrab1.sety(y) *# Set the new y coordinate*

def madrab2\_up(): *# Define function to move paddle 2 up*

y = madrab2.ycor() *# Get the current y coordinate of paddle 2*

y += 20 *# Increase y by 20*

madrab2.sety(y) *# Set the new y coordinate*

def madrab2\_down(): *# Define function to move paddle 2 down*

y = madrab2.ycor() *# Get the current y coordinate of paddle 2*

y -= 20 *# Decrease y by 20*

madrab2.sety(y) *# Set the new y coordinate*

*# keyboard bindings*

wind.listen() *# Tell the window to listen for keyboard input*

wind.onkeypress(madrab1\_up, "w") *# When 'w' key is pressed, call madrab1\_up function*

wind.onkeypress(madrab1\_down, "s") *# When 's' key is pressed, call madrab1\_down function*

wind.onkeypress(madrab2\_up, "Up") *# When 'Up' arrow key is pressed, call madrab2\_up function*

wind.onkeypress(madrab2\_down, "Down") *# When 'Down' arrow key is pressed, call madrab2\_down function*

*# main game loop*

*while* True: *# Start an infinite loop to keep the game running*

wind.update() *# Update the screen to reflect any changes (since tracer is off)*

*End of Lesson 47*

**Section 10: Lesson 48**

**Implementing Ball Movement and Boundary Collision**

**Lecturer: Ahmed Sami**

**Introduction**

Good morning, dear students. Welcome to Lesson 48 of Section 10. Today, we advance our Ping Pong game by introducing dynamic ball movement and implementing collision detection with the game boundaries. This lesson marks the transition from static gameplay elements to a truly animated gaming experience.

In this lecture, we will explore velocity-based movement, frame rate control, and boundary collision detection. These concepts are fundamental to any physics-based game and will provide you with the knowledge to create smooth, realistic motion in your applications.

By the end of this lesson, you will understand how to implement continuous object movement, detect collisions with screen boundaries, and create realistic bouncing effects. These skills form the core mechanics of countless games and simulations.

**Body**

**Part 1: Module Import and Time Control (Lines 1-2)**

**Line 1: import turtle**

As in previous lessons, we import the turtle graphics library to access our drawing and animation capabilities.

**Line 2: import time**

This new import introduces the time module, which provides various time-related functions. We will use this module to control the speed of our game loop, ensuring consistent gameplay across different computer speeds. The time module allows us to introduce precise delays between frame updates.

**Part 2: Window and Game Objects Setup (Lines 3-32)**

**Lines 3-7: Window Configuration**

These lines establish our game window with the familiar black background, 800x600 dimensions, and disabled automatic updates. The consistency in setup ensures a stable foundation for our enhanced gameplay.

**Lines 9-32: Paddle and Ball Creation**

The paddles are created identically to our previous lesson. However, notice the crucial additions to the ball object at lines 31-32:

**Line 31: ball.dx = 1.5**

This line introduces a new attribute dx (delta x) to the ball object, representing the horizontal velocity. The value 1.5 means the ball will move 1.5 pixels horizontally per frame update.

**Line 32: ball.dy = 1.5**

Similarly, dy (delta y) represents the vertical velocity. Together, dx and dy create diagonal movement at a 45-degree angle when both values are equal.

**Part 3: Movement Functions and Keyboard Bindings (Lines 34-60)**

These sections remain unchanged from our previous lesson, providing paddle control through the W/S keys for player one and arrow keys for player two. The functions move paddles by 20 pixels per key press, ensuring responsive control.

**Part 4: Enhanced Game Loop with Ball Movement (Lines 62-65)**

**Line 62: while True:**

The infinite loop continues to drive our game.

**Line 63: wind.update()**

Updates the screen to display all changes.

**Line 64: time.sleep(0.01)**

This critical addition introduces a 10-millisecond delay between each frame. This pause serves multiple purposes:

* Controls the game speed to approximately 100 frames per second
* Prevents the CPU from running at maximum capacity
* Ensures consistent gameplay speed across different computers
* Reduces system resource consumption

**Part 5: Ball Movement Implementation (Lines 67-69)**

**Line 68: ball.setx(ball.xcor() + ball.dx)**

This line implements horizontal movement:

* ball.xcor() retrieves the current x-coordinate
* Adds ball.dx to move horizontally
* setx() updates the ball's position

**Line 69: ball.sety(ball.ycor() + ball.dy)**

Similarly implements vertical movement:

* ball.ycor() retrieves the current y-coordinate
* Adds ball.dy to move vertically
* sety() updates the position

Together, these lines create smooth diagonal movement across the screen.

**Part 6: Boundary Collision Detection (Lines 71-86)**

**Lines 72-74: Top Border Collision**

python

Download

Copy code

Wrap

*if* ball.ycor() > 290:

ball.sety(290)

ball.dy \*= -1

This code detects when the ball hits the top boundary:

* Line 72: Checks if the ball's y-coordinate exceeds 290 (10 pixels from the 300-pixel top edge)
* Line 73: Prevents the ball from going beyond the boundary by setting it exactly at 290
* Line 74: Reverses the vertical direction by multiplying dy by -1, creating a bounce effect

**Lines 76-78: Bottom Border Collision**

python

Download

Copy code

Wrap

*if* ball.ycor() < -290:

ball.sety(-290)

ball.dy \*= -1

Implements identical logic for the bottom boundary:

* Detects collision at y < -290
* Constrains the ball position
* Reverses vertical direction for bouncing

**Lines 80-82: Right Border Collision**

python

Download

Copy code

Wrap

*if* ball.xcor() > 390:

ball.goto(0, 0)

ball.dx \*= -1

Handles the ball passing the right boundary:

* Line 80: Detects when the ball exceeds x=390 (missed by player 2)
* Line 81: Resets the ball to center position (0, 0)
* Line 82: Reverses horizontal direction for the next serve

**Lines 84-86: Left Border Collision**

python

Download

Copy code

Wrap

*if* ball.xcor() < -390:

ball.goto(0, 0)

ball.dx \*= -1

Handles the ball passing the left boundary:

* Detects when the ball goes beyond x=-390 (missed by player 1)
* Resets to center
* Reverses horizontal direction

The 10-pixel buffer (290 instead of 300, 390 instead of 400) accounts for the ball's size, preventing it from appearing to overlap the edges.

**Conclusion**

Today's lesson has transformed our Ping Pong game from a static display into a dynamic, animated experience. We have successfully implemented continuous ball movement using velocity vectors and created realistic boundary interactions through collision detection and directional reversal.

The concepts covered today—velocity-based movement, frame rate control, collision detection, and physics simulation—are fundamental to game development and interactive applications. These techniques apply not only to simple games but also to complex simulations and professional software.

Key achievements from this lesson include:

1. Implementation of smooth, continuous movement using velocity components
2. Frame rate control for consistent gameplay across different systems
3. Boundary collision detection with appropriate response behaviors
4. Ball reset mechanism simulating scoring events

In our next lesson, we will implement paddle-ball collisions, allowing players to actually play the game by hitting the ball back and forth. This will complete the core gameplay mechanics of our Ping Pong game.

I encourage you to experiment with different velocity values, boundary positions, and sleep durations to understand their effects on gameplay. Consider how you might add features like accelerating ball speed or varying bounce angles to enhance the gaming experience.

**Python Code Tested**

python

Download

Copy code

Wrap

*import* turtle *# Import the turtle graphics library*

*import* time *# Import the time module to pause the game loop*

wind = turtle.Screen() *# Create a window/screen object for the game*

wind.title("Ping Pong By Codezilla") *# Set the window title*

wind.bgcolor("black") *# Set the background color to black*

wind.setup(*width*=800, *height*=600) *# Set window dimensions: 800 pixels wide, 600 pixels high*

wind.tracer(0) *# Turn off automatic screen updates for better performance*

*# madrab1*

madrab1 = turtle.Turtle() *# Create a turtle object for the first paddle*

madrab1.speed(0) *# Set animation speed to maximum (0 = fastest)*

madrab1.shape("square") *# Set the shape to square*

madrab1.color("blue") *# Set the color to blue*

madrab1.shapesize(*stretch\_wid*=5, *stretch\_len*=1) *# Stretch the square: 5x height, 1x width*

madrab1.penup() *# Lift the pen to avoid drawing lines when moving*

madrab1.goto(-350, 0) *# Position the paddle at x=-350, y=0 (left side)*

*# madrab2*

madrab2 = turtle.Turtle() *# Create a turtle object for the second paddle*

madrab2.speed(0) *# Set animation speed to maximum*

madrab2.shape("square") *# Set the shape to square*

madrab2.color("red") *# Set the color to red*

madrab2.shapesize(*stretch\_wid*=5, *stretch\_len*=1) *# Stretch the square: 5x height, 1x width*

madrab2.penup() *# Lift the pen to avoid drawing lines*

madrab2.goto(350, 0) *# Position the paddle at x=350, y=0 (right side)*

*# ball*

ball = turtle.Turtle() *# Create a turtle object for the ball*

ball.speed(0) *# Set animation speed to maximum*

ball.shape("square") *# Set the shape to square*

ball.color("white") *# Set the color to white*

ball.penup() *# Lift the pen to avoid drawing lines*

ball.goto(0, 0) *# Position the ball at center (x=0, y=0)*

ball.dx = 1.5 *# Set ball's horizontal speed (pixels per frame)*

ball.dy = 1.5 *# Set ball's vertical speed (pixels per frame)*

*# functions*

def madrab1\_up(): *# Define function to move madrab1 up*

y = madrab1.ycor() *# Get the current y coordinate of madrab1*

y += 20 *# Increase y by 20 pixels*

madrab1.sety(y) *# Set madrab1's new y position*

def madrab1\_down(): *# Define function to move madrab1 down*

y = madrab1.ycor() *# Get the current y coordinate of madrab1*

y -= 20 *# Decrease y by 20 pixels*

madrab1.sety(y) *# Set madrab1's new y position*

def madrab2\_up(): *# Define function to move madrab2 up*

y = madrab2.ycor() *# Get the current y coordinate of madrab2*

y += 20 *# Increase y by 20 pixels*

madrab2.sety(y) *# Set madrab2's new y position*

def madrab2\_down(): *# Define function to move madrab2 down*

y = madrab2.ycor() *# Get the current y coordinate of madrab2*

y -= 20 *# Decrease y by 20 pixels*

madrab2.sety(y) *# Set madrab2's new y position*

*# keyboard bindings*

wind.listen() *# Tell the window to listen for keyboard input*

wind.onkeypress(madrab1\_up, "w") *# When 'w' key is pressed, call madrab1\_up function*

wind.onkeypress(madrab1\_down, "s") *# When 's' key is pressed, call madrab1\_down function*

wind.onkeypress(madrab2\_up, "Up") *# When 'Up' arrow key is pressed, call madrab2\_up function*

wind.onkeypress(madrab2\_down, "Down") *# When 'Down' arrow key is pressed, call madrab2\_down function*

*# main game loop*

*while* True: *# Infinite loop to keep the game running*

wind.update() *# Update the screen with any changes*

time.sleep(0.01) *# Pause the loop for a short time to control the game speed*

*# move the ball*

ball.setx(ball.xcor() + ball.dx) *# Move ball horizontally: current x position + dx*

ball.sety(ball.ycor() + ball.dy) *# Move ball vertically: current y position + dy*

*# border check*

*if* ball.ycor() > 290: *# Check if ball hit the top border (y > 290)*

ball.sety(290) *# Keep ball at the border position*

ball.dy \*= -1 *# Reverse vertical direction (bounce)*

*if* ball.ycor() < -290: *# Check if ball hit the bottom border (y < -290)*

ball.sety(-290) *# Keep ball at the border position*

ball.dy \*= -1 *# Reverse vertical direction (bounce)*

*if* ball.xcor() > 390: *# Check if ball passed the right border (missed by player 2)*

ball.goto(0, 0) *# Reset ball to center position*

ball.dx \*= -1 *# Reverse horizontal direction*

*if* ball.xcor() < -390: *# Check if ball passed the left border (missed by player 1)*

ball.goto(0, 0) *# Reset ball to center position*

ball.dx \*= -1 *# Reverse horizontal direction*

*End of Lesson 48*

**Section 10: Lesson 49**

**Implementing Paddle-Ball Collision Detection**

**Lecturer: Ahmed Sami**

**Introduction**

Good morning, dear students. Welcome to Lesson 49 of Section 10. Today marks a crucial milestone in our Ping Pong game development as we implement paddle-ball collision detection. This functionality transforms our game from a simple animation into a fully playable, interactive experience where players can actually hit and redirect the ball.

In this lecture, we will explore the mathematics and logic behind collision detection between rectangular objects. We will learn how to determine when two game objects intersect and how to respond appropriately to create realistic gameplay mechanics. These collision detection techniques are fundamental to virtually every video game ever created.

By the end of this lesson, you will understand how to detect collisions between moving objects, calculate collision boundaries, and implement appropriate physical responses. This knowledge represents the final piece needed to complete our functional Ping Pong game.

**Body**

**Part 1: Foundation Code Review (Lines 1-91)**

The first 91 lines of our code remain unchanged from the previous lesson. We have our window setup, paddle creation, ball initialization with velocity components, movement functions, keyboard bindings, and boundary collision detection. This existing foundation provides the framework upon which we'll add our paddle collision system.

**Part 2: Understanding Collision Detection Theory**

Before examining the code, let's understand the collision detection concept. For a ball to collide with a paddle, two conditions must be met simultaneously:

1. **Horizontal Overlap**: The ball must be at the same horizontal position as the paddle
2. **Vertical Overlap**: The ball must be within the vertical range of the paddle

This is known as Axis-Aligned Bounding Box (AABB) collision detection, one of the most efficient methods for detecting collisions between rectangular objects.

**Part 3: Left Paddle Collision Detection (Lines 93-96)**

**Line 93-94: Collision Condition Check**

python

Download

Copy code

Wrap

*if* (ball.xcor() < -340 and ball.xcor() > -350) and (ball.ycor() < madrab1.ycor() + 40 and ball.ycor() > madrab1.ycor() - 40):

This complex conditional statement checks for collision with the left paddle (madrab1). Let's break it down:

**Horizontal Check: (ball.xcor() < -340 and ball.xcor() > -350)**

* The left paddle is positioned at x=-350
* We check if the ball is between x=-350 and x=-340
* This 10-pixel range represents the paddle's hitting zone
* The ball must be approaching from the right (x > -350) but not past the paddle (x < -340)

**Vertical Check: (ball.ycor() < madrab1.ycor() + 40 and ball.ycor() > madrab1.ycor() - 40)**

* madrab1.ycor() gives the paddle's center y-coordinate
* We add and subtract 40 pixels to create an 80-pixel vertical hit zone
* This accounts for the paddle's height (stretched to 5 times the default 20-pixel square = 100 pixels)
* The 40-pixel range on each side provides a reasonable collision area

**Line 95: ball.setx(-340)**

When collision is detected, we immediately position the ball at x=-340. This prevents the ball from penetrating through the paddle or getting stuck inside it. This technique, called "position correction," is essential for stable collision physics.

**Line 96: ball.dx \*= -1**

We reverse the ball's horizontal velocity, causing it to bounce back toward the opposite side. The multiplication by -1 maintains the speed magnitude while reversing the direction.

**Part 4: Right Paddle Collision Detection (Lines 98-101)**

**Line 98-99: Collision Condition Check**

python

Download

Copy code

Wrap

*if* (ball.xcor() > 340 and ball.xcor() < 350) and (ball.ycor() < madrab2.ycor() + 40 and ball.ycor() > madrab2.ycor() - 40):

This mirrors the left paddle collision but with adjusted coordinates:

**Horizontal Check: (ball.xcor() > 340 and ball.xcor() < 350)**

* The right paddle is positioned at x=350
* We check if the ball is between x=340 and x=350
* The ball must be approaching from the left (x < 350) but not past the paddle (x > 340)

**Vertical Check: (ball.ycor() < madrab2.ycor() + 40 and ball.ycor() > madrab2.ycor() - 40)**

* Identical logic to the left paddle
* Creates an 80-pixel vertical hit zone centered on the paddle

**Line 100: ball.setx(340)**

Positions the ball at x=340 upon collision, preventing penetration and ensuring clean bounces.

**Line 101: ball.dx \*= -1**

Reverses the horizontal velocity to send the ball back toward the left side.

**Part 5: Collision Detection Optimization Considerations**

The current implementation uses fixed collision boundaries, which works well for our simple game. However, there are several important considerations:

1. **Paddle Size Accuracy**: The 40-pixel vertical range is slightly less than the actual paddle height (50 pixels from center), providing a more realistic hit zone that requires precision.
2. **Frame Rate Dependency**: At very high speeds, the ball might pass through the paddle between frames. Our current speed settings prevent this issue.
3. **Edge Cases**: The position correction (setx) ensures the ball doesn't get stuck inside the paddle even if collision detection occurs late.

**Conclusion**

Today's lesson has completed the core gameplay mechanics of our Ping Pong game. We have successfully implemented paddle-ball collision detection, enabling players to interact with the ball and play an actual game. The collision detection system we built demonstrates fundamental concepts applicable to countless game development scenarios.

The key concepts we mastered today include:

1. Axis-Aligned Bounding Box (AABB) collision detection
2. Compound conditional logic for simultaneous condition checking
3. Position correction to prevent object penetration
4. Velocity reversal for realistic bounce physics

Our Ping Pong game now features:

* Responsive paddle controls
* Smooth ball movement
* Boundary collision with bouncing
* Paddle collision detection with ball redirection

In future lessons, we could enhance this game by adding:

* Score tracking and display
* Increasing ball speed over time
* Sound effects for collisions
* Paddle edge effects for varied ball angles
* AI opponents

I encourage you to experiment with the collision boundaries, try different paddle sizes, and observe how these changes affect gameplay. Understanding collision detection is crucial for any aspiring game developer, as it forms the foundation of interactive gaming experiences.

**Python Code Tested**

python

Download

Copy code

Wrap

*import* turtle *# Import the turtle graphics library*

*import* time *# Import the time module to pause the game loop*

wind = turtle.Screen() *# Create a window/screen object for the game*

wind.title("Ping Pong By Codezilla") *# Set the window title*

wind.bgcolor("black") *# Set the background color to black*

wind.setup(*width*=800, *height*=600) *# Set window dimensions: 800 pixels wide, 600 pixels high*

wind.tracer(0) *# Turn off automatic screen updates for better performance*

*# madrab1*

madrab1 = turtle.Turtle() *# Create a turtle object for the first paddle*

madrab1.speed(0) *# Set animation speed to maximum (0 = fastest)*

madrab1.shape("square") *# Set the shape to square*

madrab1.color("blue") *# Set the color to blue*

madrab1.shapesize(*stretch\_wid*=5, *stretch\_len*=1) *# Stretch the square: 5x height, 1x width*

madrab1.penup() *# Lift the pen to avoid drawing lines when moving*

madrab1.goto(-350, 0) *# Position the paddle at x=-350, y=0 (left side)*

*# madrab2*

madrab2 = turtle.Turtle() *# Create a turtle object for the second paddle*

madrab2.speed(0) *# Set animation speed to maximum*

madrab2.shape("square") *# Set the shape to square*

madrab2.color("red") *# Set the color to red*

madrab2.shapesize(*stretch\_wid*=5, *stretch\_len*=1) *# Stretch the square: 5x height, 1x width*

madrab2.penup() *# Lift the pen to avoid drawing lines*

madrab2.goto(350, 0) *# Position the paddle at x=350, y=0 (right side)*

*# ball*

ball = turtle.Turtle() *# Create a turtle object for the ball*

ball.speed(0) *# Set animation speed to maximum*

ball.shape("square") *# Set the shape to square*

ball.color("white") *# Set the color to white*

ball.penup() *# Lift the pen to avoid drawing lines*

ball.goto(0, 0) *# Position the ball at center (x=0, y=0)*

ball.dx = 1.5 *# Set ball's horizontal speed (pixels per frame)*

ball.dy = 1.5 *# Set ball's vertical speed (pixels per frame)*

*# functions*

def madrab1\_up(): *# Define function to move madrab1 up*

y = madrab1.ycor() *# Get the current y coordinate of madrab1*

y += 20 *# Increase y by 20 pixels*

madrab1.sety(y) *# Set madrab1's new y position*

def madrab1\_down():*# Define function to move madrab1 down*

y = madrab1.ycor() *# Get the current y coordinate of madrab1*

y -= 20 *# Decrease y by 20 pixels*

madrab1.sety(y) *# Set madrab1's new y position*

def madrab2\_up(): *# Define function to move madrab2 up*

y = madrab2.ycor() *# Get the current y coordinate of madrab2*

y += 20 *# Increase y by 20 pixels*

madrab2.sety(y) *# Set madrab2's new y position*

def madrab2\_down():*# Define function to move madrab2 down*

y = madrab2.ycor() *# Get the current y coordinate of madrab2*

y -= 20 *# Decrease y by 20 pixels*

madrab2.sety(y) *# Set madrab2's new y position*

*# keyboard bindings*

wind.listen() *# Tell the window to listen for keyboard input*

wind.onkeypress(madrab1\_up, "w") *# When 'w' key is pressed, call madrab1\_up function*

wind.onkeypress(madrab1\_down, "s") *# When 's' key is pressed, call madrab1\_down function*

wind.onkeypress(madrab2\_up, "Up") *# When 'Up' arrow key is pressed, call madrab2\_up function*

wind.onkeypress(madrab2\_down, "Down") *# When 'Down' arrow key is pressed, call madrab2\_down function*

*# main game loop*

*while* True: *# Infinite loop to keep the game running*

wind.update() *# Update the screen with any changes*

time.sleep(0.01) *# Pause the loop for a short time to control the game speed*

*# move the ball*

ball.setx(ball.xcor() + ball.dx) *# Move ball horizontally: current x position + dx*

ball.sety(ball.ycor() + ball.dy) *# Move ball vertically: current y position + dy*

*# border check*

*if* ball.ycor() > 290: *# Check if ball hit the top border (y > 290)*

ball.sety(290) *# Keep ball at the border position*

ball.dy \*= -1 *# Reverse vertical direction (bounce)*

*if* ball.ycor() < -290: *# Check if ball hit the bottom border (y < -290)*

ball.sety(-290) *# Keep ball at the border position*

ball.dy \*= -1 *# Reverse vertical direction (bounce)*

*if* ball.xcor() > 390: *# Check if ball passed the right border (missed by player 2)*

ball.goto(0, 0) *# Reset ball to center position*

ball.dx \*= -1 *# Reverse horizontal direction*

*if* ball.xcor() < -390: *# Check if ball passed the left border (missed by player 1)*

ball.goto(0, 0) *# Reset ball to center position*

ball.dx \*= -1 *# Reverse horizontal direction*

*# Check for collision with the left paddle (madrab1).*

*# It checks if the ball is within the horizontal and vertical range of the paddle.*

*if* (ball.xcor() < -340 and ball.xcor() > -350) and (ball.ycor() < madrab1.ycor() + 40 and ball.ycor() > madrab1.ycor() - 40):

ball.setx(-340) *# Move the ball to the paddle's surface to prevent it from getting stuck*

ball.dx \*= -1 *# Reverse the ball's horizontal direction to make it bounce*

*# Check for collision with the right paddle (madrab2).*

*# It checks if the ball is within the horizontal and vertical range of the paddle.*

*if* (ball.xcor() > 340 and ball.xcor() < 350) and (ball.ycor() < madrab2.ycor() + 40 and ball.ycor() > madrab2.ycor() - 40):

ball.setx(340) *# Move the ball to the paddle's surface to prevent it from getting stuck*

ball.dx \*= -1 *# Reverse the ball's horizontal direction to make it bounce*

*End of Lesson 49*

**Section 10: Lesson 50**

**Implementing Score Tracking and Real-Time Scoring in Ping Pong**

**Lecturer: Ahmed Sami**

**Introduction**

Good morning, students. Welcome to Lesson 50 of Section 10. In this final core lesson of our series, we will complete the infrastructure of our Ping Pong game by implementing a **real-time scoring system**. This system will track points, update the screen dynamically, and notify players when they score.

Today’s focus is on:

1. **Score Data Management**: How to store and update player scores using variables.
2. **Graphical Score Display**: Rendering text on the screen via a dedicated turtle object.
3. **Integrated Scoring Logic**: Linking the ball’s movement to scoring events (when it passes left/right boundaries).
4. **Clean Code Practices**: Ensuring the scoreboard updates smoothly and efficiently.

By the end of this lesson, your game will fully reward players for successful defense and provide a clear heads-up display of their progress.

**Body**

**1. Setup of the Score Tracking System (Code Lines 30–42)**

**Score Variables:**

python

Download

Copy code

Wrap

score1 = 0 *# Player 1's score*

score2 = 0 *# Player 2's score*

These integers store each player’s current score.

**Scoreboard Turtle (Text Display):**

A dedicated score turtle renders the score text at the top center of the screen:

python

Download

Copy code

Wrap

score = turtle.Turtle()

score.speed(0)

score.color("white")

score.penup()

score.hideturtle() *# Hide the turtle arrow*

score.goto(0, 260) *# Position at the center top*

score.write("Player 1: 0 Player 2: 0",

*align*="center",

*font*=("Courier", 24, "normal"))

* .hideturtle() ensures only the text is visible.
* font uses a clear, readable format for large text.

**2. Main Game Loop Enhancements (Code Lines 55–110)**

**Scoring Logic When the Ball Misses a Paddle**

When the ball crosses the **right boundary**, Player 1 scores:

python

Download

Copy code

Wrap

*if* ball.xcor() > 390:

ball.goto(0, 0) *# Reset the ball*

ball.dx \*= -1 *# Reverse direction for next serve*

score1 += 1 *# Increment Player 1's score*

score.clear() *# Clear old score text*

score.write(f"Player 1: {score1} Player 2: {score2}",

*align*="center",

*font*=("Courier", 24, "normal")) *# Re-render new score*

For crossing the **left boundary**, Player 2 scores similarly:

python

Download

Copy code

Wrap

*if* ball.xcor() < -390:

ball.goto(0, 0)

ball.dx \*= -1

score2 += 1

score.clear()

score.write(f"Player 1: {score1} Player 2: {score2}", ...)

**Efficiency Notes:**

* score.clear() erases the previous text to prevent overlap.
* f-strings (e.g., f"Player 1: {score1}") dynamically update the displayed text.

**3. Collision Detection (Revisited)**

The paddle collision logic ensures the ball bounces back when hit. Example for the **right paddle**:

python

Download

Copy code

Wrap

*if* (340 < ball.xcor() < 350) and \

(madrab2.ycor() - 40 < ball.ycor() < madrab2.ycor() + 40):

ball.setx(340) *# Reset position to edge of paddle*

ball.dx \*= -1 *# Reverse direction*

* Vertical bounds check ensures the ball collides *within* the paddle’s area.

**4. Key Code Structure & Best Practices**

**Score Updates**

* Scores are stored in global variables (score1, score2).
* The score turtle handles all display logic.
* .write() is called *after* clearing the screen to avoid flicker.

**Ball Reset After Scoring**

* The ball returns to (0, 0) and reverses direction, giving the opponent a fair *serve*.

**Conclusion**

With today’s lesson, your Ping Pong game is now **fully functional**! Key achievements include:

1. **Live Scoring**: Scores update dynamically as players defend.
2. **Clean Display**: Text rendering avoids overlapping with .clear() and .write().
3. **Competitive Flow**: The ball resets fairly after each missed shot.

Potential enhancements for future lessons:

* Add sound effects for scores and collisions.
* Implement a "game over" screen at a set score limit (e.g., 10 points).

**Python Code Tested**

python

Download

Copy code

Wrap

*import* turtle

*import* time

wind = turtle.Screen()

wind.title("Ping Pong By Codezilla")

wind.bgcolor("black")

wind.setup(*width*=800, *height*=600)

wind.tracer(0)

*# Paddle Setup*

madrab1 = turtle.Turtle() *# Left paddle*

madrab1.speed(0)

madrab1.shape("square")

madrab1.color("blue")

madrab1.shapesize(*stretch\_wid*=5, *stretch\_len*=1)

madrab1.penup()

madrab1.goto(-350, 0)

madrab2 = turtle.Turtle() *# Right paddle*

madrab2.speed(0)

madrab2.shape("square")

madrab2.color("red")

madrab2.shapesize(*stretch\_wid*=5, *stretch\_len*=1)

madrab2.penup()

madrab2.goto(350, 0)

*# Ball Setup*

ball = turtle.Turtle()

ball.speed(0)

ball.shape("square")

ball.color("white")

ball.penup()

ball.goto(0, 0)

ball.dx, ball.dy = 1.5, 1.5

*# Score Tracking*

score1 = 0

score2 = 0

score = turtle.Turtle()

score.speed(0)

score.color("white")

score.penup()

score.hideturtle()

score.goto(0, 260)

score.write(f"Player 1: {score1} Player 2: {score2}",

*align*="center",

*font*=("Courier", 24, "normal"))

*# Movement Functions*

def paddle\_up(*paddle*):

y = paddle.ycor()

y += 20

paddle.sety(y)

def paddle\_down(*paddle*):

y = paddle.ycor()

y -= 20

paddle.sety(y)

*# Keyboard Bindings*

wind.listen()

wind.onkeypress(lambda: paddle\_up(madrab1), "w")

wind.onkeypress(lambda: paddle\_down(madrab1), "s")

wind.onkeypress(lambda: paddle\_up(madrab2), "Up")

wind.onkeypress(lambda: paddle\_down(madrab2), "Down")

*# Main Game Loop*

*while* True:

wind.update()

time.sleep(0.01)

*# Ball Movement*

ball.setx(ball.xcor() + ball.dx)

ball.sety(ball.ycor() + ball.dy)

*# Border Checks (Top/Bottom)*

*if* ball.ycor() > 290 or ball.ycor() < -290:

ball.dy \*= -1

*# Scoring and Reset*

*if* ball.xcor() > 390:

score1 += 1

score.clear()

score.write(f"Player 1: {score1} Player 2: {score2}", ...)

ball.goto(0, 0)

ball.dx \*= -1

*if* ball.xcor() < -390:

score2 += 1

score.clear()

score.write(f"Player 1: {score1} Player 2: {score2}", ...)

ball.goto(0, 0)

ball.dx \*= -1

*# Paddle Collision Detection*

*# Right paddle check*

*if* 340 < ball.xcor() < 350 and \

(madrab2.ycor() - 40 < ball.ycor() < madrab2.ycor() + 40):

ball.setx(340)

ball.dx \*= -1

*# Left paddle check*

*if* -350 > ball.xcor() > -360 and \

(madrab1.ycor() - 40 < ball.ycor() < madrab1.ycor() + 40):

ball.setx(-340)

ball.dx \*= -1

**Note:** Ensure proper indentation in your environment. This code runs seamlessly in Python Turtle graphics.

**End of Lesson 50**

**Section 10: Lesson 50**

**Implementing the Scoring System - Final Lecture**

**Lecturer: Ahmed Sami**

**Introduction**

Good morning, dear students. Welcome to Lesson 50 of Section 10, the final lecture in our Ping Pong game development journey. Today, we will complete our game by implementing a comprehensive scoring system that tracks points between the two players. This scoring mechanism transforms our simple ball-bouncing animation into a competitive, fully-functional game.

In this concluding lecture, we will focus intensively on:

1. **Line-by-line analysis** of the scoring system implementation
2. **Understanding score variable management** and how points are tracked
3. **The scoreboard display mechanism** using a dedicated turtle object
4. **Integration of scoring logic** with game events
5. **Best practices for text rendering** in turtle graphics

By the end of this final lesson, you will have mastered the complete development cycle of a classic arcade game, from basic graphics to competitive gameplay mechanics.

**Body**

**Part 1: Score Variables Initialization (Lines 39-40)**

**Line 39: score1 = 0**

This line creates an integer variable called score1 and initializes it to zero. This variable will maintain Player 1's (left paddle) score throughout the entire game session. Starting at zero ensures both players begin on equal footing.

**Line 40: score2 = 0**

Similarly, we create score2 to track Player 2's (right paddle) points. These two variables form the foundation of our scoring system - simple integers that will be incremented each time a player scores a point.

**Part 2: Creating the Scoreboard Display Object (Lines 41-48)**

**Line 41: score = turtle.Turtle()**

We instantiate a new turtle object specifically for displaying the score. This separation of concerns is crucial - having a dedicated turtle for text display keeps our code organized and prevents interference with game objects.

**Line 42: score.speed(0)**

Sets the animation speed to maximum (0). This ensures instantaneous updates when we modify the score text, preventing any visual lag during score changes.

**Line 43: score.color("white")**

Defines the text color as white. This provides optimal contrast against our black background, ensuring maximum readability for players during intense gameplay.

**Line 44: score.penup()**

Lifts the pen to prevent drawing lines when we position the score turtle. Since we only want text display, not line drawing, this is essential.

**Line 45: score.hideturtle()**

This critical line hides the turtle cursor itself. We only want to see the text output, not the turtle shape. This creates a clean, professional scoreboard appearance.

**Line 46: score.goto(0, 260)**

Positions the scoreboard at coordinates (0, 260):

* **x=0**: Centers the text horizontally on the screen
* **y=260**: Places it near the top, well above the gameplay area (remember our window height is 600, so 260 is near the top edge)

**Lines 47-48: Initial Score Display**

python

Download

Copy code

Wrap

score.write("Player 1: 0 Player 2: 0", *align*="center", *font*=("Courier", 24, "normal"))

This writes the initial scoreboard text:

* **First parameter**: The text string to display
* **align="center"**: Centers the text at the specified position
* **font=("Courier", 24, "normal")**: Specifies font family (Courier - monospaced), size (24 points), and style (normal weight)

**Part 3: Scoring Logic - Player 1 Scores (Lines 93-98)**

Now let's examine what happens when Player 1 scores (ball passes the right boundary):

**Line 93: if ball.xcor() > 390:**

This condition checks if the ball's x-coordinate exceeds 390 pixels. Since our window width is 800 (ranging from -400 to 400), and the right paddle is at x=350, when the ball reaches x=390, it has clearly passed the right paddle - Player 2 has missed!

**Line 94: ball.goto(0, 0)**

Immediately resets the ball to the center of the screen. This prepares for the next round of play, giving both players a fair restart position.

**Line 95: ball.dx \*= -1**

Reverses the horizontal direction of the ball. This ensures that after scoring, the ball will initially move toward the player who just scored, following traditional ping pong serving rules.

**Line 96: score1 += 1**

The crucial scoring line! This increments Player 1's score by 1. The += operator is shorthand for score1 = score1 + 1. This is where the actual point is recorded.

**Line 97: score.clear()**

Clears the previous score text from the screen. This is absolutely essential - without clearing, new text would overlay on old text, creating an unreadable mess. The clear() method removes all text previously written by this turtle.

**Line 98: Score Display Update**

python

Download

Copy code

Wrap

score.write(f"Player 1: {score1} Player 2: {score2}", *align*="center", *font*=("Courier", 24, "normal"))

Writes the updated score using an f-string (formatted string literal):

* **f"..."**: The 'f' prefix enables variable interpolation
* **{score1}**: Inserts the current value of score1 into the string
* **{score2}**: Inserts the current value of score2 into the string
* The formatting parameters remain the same as the initial display

**Part 4: Scoring Logic - Player 2 Scores (Lines 100-105)**

The logic for Player 2 scoring mirrors Player 1's:

**Line 100: if ball.xcor() < -390:**

Checks if the ball has passed the left boundary (x < -390), meaning Player 1 missed the ball.

**Line 101: ball.goto(0, 0)**

Resets the ball position to center, identical to Player 1's scoring scenario.

**Line 102: ball.dx \*= -1**

Reverses horizontal direction for the next serve.

**Line 103: score2 += 1**

This is the key difference - here we increment Player 2's score instead of Player 1's.

**Line 104: score.clear()**

Clears the old score display to prevent text overlap.

**Line 105: Score Display Update**

python

Download

Copy code

Wrap

score.write(f"Player 1: {score1} Player 2: {score2}", *align*="center", *font*=("Courier", 24, "normal"))

Updates the display with the new score. Note that even though only score2 changed, we rewrite both scores to maintain consistency.

**Part 5: Technical Implementation Details**

**Why Use f-strings?**

F-strings (introduced in Python 3.6) provide the cleanest way to embed variables in strings:

* More readable than concatenation: "Player 1: " + str(score1) + ...
* More efficient than format(): "Player 1: {} Player 2: {}".format(score1, score2)
* Direct variable interpolation with {variable\_name}

**The Clear-Write Pattern**

The pattern of clear() followed by write() is crucial:

1. **Without clear()**: Text accumulates, creating visual artifacts
2. **Clear() alone**: Would leave the screen blank
3. **Clear() then write()**: Provides smooth, flicker-free updates

**Integration with Game Flow**

The scoring system seamlessly integrates with:

* **Boundary detection**: Triggers when ball exits play area
* **Ball reset**: Returns ball to center for fair play
* **Direction reversal**: Implements proper serve mechanics
* **Visual feedback**: Immediate score update informs players

**Conclusion**

Congratulations! With this final lecture, we have completed our journey from a blank screen to a fully functional Ping Pong game. Today's scoring system implementation represents the culmination of everything we've learned:

**Our Complete Game Features:**

1. **Responsive paddle controls** with keyboard input
2. **Smooth ball physics** with velocity-based movement
3. **Collision detection** for paddles and boundaries
4. **Real-time scoring** with dynamic display updates
5. **Fair gameplay mechanics** with proper ball resets

**Key Technical Achievements:**

* Mastered turtle graphics for game development
* Implemented event-driven programming
* Created efficient game loops with proper timing
* Developed collision detection algorithms
* Built a complete scoring and display system

**The Journey We've Completed:**

* Lesson 46: Window setup and basic configuration
* Lesson 47: Paddle creation and movement functions
* Lesson 48: Ball physics and boundary collision
* Lesson 49: Paddle-ball collision detection
* Lesson 50: Complete scoring system

This game now stands as a testament to your programming skills. You've built a classic game from scratch, understanding every line of code and every design decision.

**Future Enhancements You Could Explore:**

* Add a winning condition (first to 10 points)
* Implement difficulty levels with variable ball speed
* Add sound effects for collisions and scoring
* Create an AI opponent for single-player mode
* Design a menu system for game options

Thank you for joining me on this incredible journey. You now possess the knowledge to create your own games and interactive applications. Keep coding, keep creating, and remember - every expert was once a beginner who never gave up.

**Python Code Tested**

python

Download

Copy code

Wrap

*import* turtle *# Import the turtle graphics library*

*import* time *# Import the time module to pause the game loop*

*# Create a window (screen) object*

wind = turtle.Screen() *# Create a window/screen object for the game*

wind.title("Ping Pong By Codezilla") *# Set the window title*

wind.bgcolor("black") *# Set the background color to black*

wind.setup(*width*=800, *height*=600) *# Set window dimensions: 800 pixels wide, 600 pixels high*

wind.tracer(0) *# Turn off automatic screen updates for better performance*

*# madrab1 (Left Paddle)*

madrab1 = turtle.Turtle() *# Create a turtle object for the first paddle*

madrab1.speed(0) *# Set animation speed to maximum (0 = fastest)*

madrab1.shape("square") *# Set the shape to square*

madrab1.color("blue") *# Set the color to blue*

madrab1.shapesize(*stretch\_wid*=5, *stretch\_len*=1) *# Stretch the square to form a paddle*

madrab1.penup() *# Lift the pen to avoid drawing lines when moving*

madrab1.goto(-350, 0) *# Position the paddle on the left side*

*# madrab2 (Right Paddle)*

madrab2 = turtle.Turtle() *# Create a turtle object for the second paddle*

madrab2.speed(0) *# Set animation speed to maximum*

madrab2.shape("square") *# Set the shape to square*

madrab2.color("red") *# Set the color to red*

madrab2.shapesize(*stretch\_wid*=5, *stretch\_len*=1) *# Stretch the square to form a paddle*

madrab2.penup() *# Lift the pen to avoid drawing lines*

madrab2.goto(350, 0) *# Position the paddle on the right side*

*# ball*

ball = turtle.Turtle() *# Create a turtle object for the ball*

ball.speed(0) *# Set animation speed to maximum*

ball.shape("square") *# Set the shape to square*

ball.color("white") *# Set the color to white*

ball.penup() *# Lift the pen to avoid drawing lines*

ball.goto(0, 0) *# Position the ball at the center*

ball.dx = 1.5 *# Set ball's horizontal speed (pixels per frame)*

ball.dy = 1.5 *# Set ball's vertical speed (pixels per frame)*

*# Score*

score1 = 0 *# Variable to store player 1's score*

score2 = 0 *# Variable to store player 2's score*

score = turtle.Turtle() *# Create a turtle object for the scoreboard*

score.speed(0) *# Set animation speed to maximum*

score.color("white") *# Set the text color to white*

score.penup() *# Lift the pen to avoid drawing lines*

score.hideturtle() *# Hide the turtle object, we only need the text*

score.goto(0, 260) *# Position the scoreboard at the top center*

*# Write the initial score on the screen*

score.write("Player 1: 0 Player 2: 0", *align*="center", *font*=("Courier", 24, "normal"))

*# functions*

def madrab1\_up(): *# Define function to move madrab1 up*

y = madrab1.ycor() *# Get the current y coordinate of madrab1*

y += 20 *# Increase y by 20 pixels*

madrab1.sety(y) *# Set madrab1's new y position*

def madrab1\_down(): *# Define function to move madrab1 down*

y = madrab1.ycor() *# Get the current y coordinate of madrab1*

y -= 20 *# Decrease y by 20 pixels*

madrab1.sety(y) *# Set madrab1's new y position*

def madrab2\_up(): *# Define function to move madrab2 up*

y = madrab2.ycor() *# Get the current y coordinate of madrab2*

y += 20 *# Increase y by 20 pixels*

madrab2.sety(y) *# Set madrab2's new y position*

def madrab2\_down(): *# Define function to move madrab2 down*

y = madrab2.ycor() *# Get the current y coordinate of madrab2*

y -= 20 *# Decrease y by 20 pixels*

madrab2.sety(y) *# Set madrab2's new y position*

*# keyboard bindings*

wind.listen() *# Tell the window to listen for keyboard input*

wind.onkeypress(madrab1\_up, "w") *# When 'w' key is pressed, call madrab1\_up function*

wind.onkeypress(madrab1\_down, "s") *# When 's' key is pressed, call madrab1\_down function*

wind.onkeypress(madrab2\_up, "Up") *# When 'Up' arrow key is pressed, call madrab2\_up function*

wind.onkeypress(madrab2\_down, "Down") *# When 'Down' arrow key is pressed, call madrab2\_down function*

*# main game loop*

*while* True: *# Infinite loop to keep the game running*

wind.update() *# Update the screen with any changes*

time.sleep(0.01) *# Pause the loop for a short time to control the game speed*

*# move the ball*

ball.setx(ball.xcor() + ball.dx) *# Move ball horizontally based on its dx value*

ball.sety(ball.ycor() + ball.dy) *# Move ball vertically based on its dy value*

*# border checking*

*if* ball.ycor() > 290: *# Check if ball hit the top border*

ball.sety(290) *# Keep ball at the border position*

ball.dy \*= -1 *# Reverse vertical direction (bounce)*

*if* ball.ycor() < -290: *# Check if ball hit the bottom border*

ball.sety(-290) *# Keep ball at the border position*

ball.dy \*= -1 *# Reverse vertical direction (bounce)*

*if* ball.xcor() > 390: *# Check if ball passed the right border (Player 1 scores)*

ball.goto(0, 0) *# Reset ball to center position*

ball.dx \*= -1 *# Reverse horizontal direction for the next round*

score1 += 1 *# Add a point to Player 1*

score.clear() *# Clear the old score from the screen*

*# Update the score display with the new scores*

score.write(f"Player 1: {score1} Player 2: {score2}", *align*="center", *font*=("Courier", 24, "normal"))

*if* ball.xcor() < -390: *# Check if ball passed the left border (Player 2 scores)*

ball.goto(0, 0) *# Reset ball to center position*

ball.dx \*= -1 *# Reverse horizontal direction for the next round*

score2 += 1 *# Add a point to Player 2*

score.clear() *# Clear the old score from the screen*

*# Update the score display with the new scores*

score.write(f"Player 1: {score1} Player 2: {score2}", *align*="center", *font*=("Courier", 24, "normal"))

*# Check for collision with the right paddle (madrab2).*

*# It checks if the ball is within the horizontal and vertical range of the paddle.*

*if* (ball.xcor() > 340 and ball.xcor() < 350) and (ball.ycor() < madrab2.ycor() + 40 and ball.ycor() > madrab2.ycor() - 40):

ball.setx(340) *# Move the ball to the paddle's surface to prevent it from getting stuck*

ball.dx \*= -1 *# Reverse the ball's horizontal direction to make it bounce*

*# Check for collision with the left paddle (madrab1).*

*# It checks if the ball is within the horizontal and vertical range of the paddle.*

*if* (ball.xcor() < -340 and ball.xcor() > -350) and (ball.ycor() < madrab1.ycor() + 40 and ball.ycor() > madrab1.ycor() - 40):

ball.setx(-340) *# Move the ball to the paddle's surface to prevent it from getting stuck*

ball.dx \*= -1 *# Reverse the ball's horizontal direction to make it bounce*

*End of Lesson 50 - Final Lecture of Ping Pong Game Development*