CS 1410: Pong

In computer science, classes and object-oriented programming are very commonly used when creating programs that involve computer graphics and graphical user interfaces. To display something visually on the screen, like a button, a class is used to represent a Button, and multiple instances of the Button class can be created to display multiple buttons on the screen at one time. This is the foundation on which all modern applications are created, whether it’s on your computer, phone, or television.

Pong was one of the first arcade games released to the public. In this assignment, you will create a version of the game. You can watch the two player version in many places.

Assignment

Your assignment is to create a program using Python and PyGame that allows two users to play Pong, using the architecture listed below.

This is a sketch of the active elements you will be creating for this project:

Part 1

The assignment is broken into two pieces. The first part requires the Ball class. We have provided Ball class unit tests.

The Ball class has a large number of data members, getter methods for most of the methods and about a dozen methods to handle the details of the ball.

It is common to use the Unified Modeling Language (UML) to describe an outline for a class in a program’s architecture. This UML diagram lists the data members and methods for the Ball class.

Ball UML Diagram
**Ball Data Members**

The data members may be easier to understand while looking at a picture.

The data members track the position, size and speed of the Ball. They also keep track of the boundaries of the ball’s travel, and the relevant location information for the two paddles. Note that this is not a complete representation of the paddles or the walls. It’s just enough to let the ball know how to move correctly.

**Ball Methods**

The Ball class has a long list of methods, but many are simple getter methods. We will not discuss them here, but you must implement them for the unit tests to pass.

---

`__init__`

This method initializes all of the data members shown in the UML diagram. Many of the data members are initialized from the parameters to the method. Set `mX` and `mY` using `min_x` and `min_y`. Set `mDX` and `mDY` to 0. Set the paddle minimum y values to `min_y` and the paddle maximum y values to `max_y`. If you’re not sure what initial value to assign to a data member, ask in the class discussion forums.

`setPosition`

Updates the `mX` and `mY` data members, but only if the new values are within the minimum and maximum values specified by the data members.

`setSpeed`

Updates the `mDX` and `mDY` data members. Does not check the values.

`setLeftPaddleY`

Updates the `mLeftPaddleMinY` and `mLeftPaddleMaxY` data members, but only if the new values are within the minimum and maximum values specified by the data members.
**setRightPaddleY**

Updates the mRightPaddleMinY and mRightPaddleMaxY data members, but only if the new values are within the minimum and maximum values specified by the data members.

**checkTop**

Receives the proposed new\_y value for the ball. If the new y value would not cause the ball to bounce from the top wall, then return new\_y unchanged. If the value would cause the ball to bounce, then reverse the sign of mDY, calculate the corrected new\_y value and return it. The picture below may help.

![Diagram](image)

\[
\Delta y = 2y_1 + \Delta y_2 = y - \text{new}\_y
\]

\[
\Delta y_1 = y - \text{min}\_y
\]

\[
\Delta y_2 = \text{min}\_y - \text{new}\_y
\]

\[
\text{new}\_y = \text{min}\_y + \Delta y_2
\]

\[
\Delta y = - \Delta y
\]

**checkBottom**

Receives the proposed new\_y value for the ball. If the new y value would not cause the ball to bounce from the bottom wall, then return new\_y unchanged. If the value would cause the ball to bounce, then reverse the sign of mDY, calculate the corrected new\_y value and return it. This is similar to checkTop, but you need to include the ball’s size in your calculations.

**checkLeft**

Receives the proposed new\_x value for the ball. If the new x value would not cause the ball to touch the left wall, then return new\_x unchanged. If the value would cause the ball to touch, then stop the ball, calculate the corrected new\_x value and return it. Note that this will cause the ball to stick to the wall where it touches.

**checkRight**

Receives the proposed new\_x value for the ball. If the new x value would not cause the ball to touch the right wall, then return new\_x unchanged. If the value would cause the ball to touch, then stop the ball, calculate the corrected new\_x value and return it. Note that this will cause the ball to stick to the wall where it touches.

The picture below may help.
**checkLeftPaddle**

Receives the proposed \( \text{new}_x \) and \( \text{new}_y \) values for the ball. If the new \( x \) and new \( y \) values would not cause the ball to touch the left paddle, then return \( \text{new}_x \) unchanged. If the value would cause the ball to touch, then bounce the ball from the paddle. This requires the \( \text{mDX} \) to change signs. Calculate the corrected \( \text{new}_x \) value and return it.

To touch the paddle, the ball’s \( \text{mid}_y \) value must be between the paddle’s minimum and maximum \( y \) values. The picture below may help.
**checkRightPaddle**

Receives the proposed \( \text{new}_x \) and \( \text{new}_y \) values for the ball. If the new \( x \) and new \( y \) values would not cause the ball to touch the right paddle, then return \( \text{new}_x \) unchanged. If the value would cause the ball to touch, then bounce the ball from the paddle. This requires the \( mDX \) to change signs. Calculate the corrected \( \text{new}_x \) value and return it.

To touch the paddle, the ball’s \( \text{mid}_y \) value must be between the paddle’s minimum and maximum \( y \) values.

**move**

Receives \( dt \), the amount of seconds that have passed since the last frame. Uses \( mX, mDX \) and \( dt \) to calculate \( \text{new}_x \), the proposed new \( x \) position of the ball. Does similarly for to calculate \( \text{new}_y \). Uses \( \text{checkTop}, \text{checkBottom}, \text{checkLeft}, \text{checkRight}, \text{checkLeftPaddle} \) and \( \text{checkRightPaddle} \) to update the values of \( \text{new}_x \) and \( \text{new}_y \). Note that these methods will also change the sign of \( mDX \) and/or \( mDY \) if necessary. \( \text{move} \) doesn’t need to worry about it. Finally sets \( mX \) and \( mY \) from \( \text{new}_x \) and \( \text{new}_y \).

**serveLeft**

Recieves several parameters. See the UML diagram for the full list. Sets the ball’s position using the \( x \) parameter and a \( y \)-value randomly chosen between \( \text{min}_y \) and \( \text{max}_y \). You may want to look at the \( \text{random.uniform()} \) function. Sets the ball’s \( mDX \) to a randomly chosen value between \( \text{min}_dx \) and \( \text{max}_dx \). Sets the ball’s \( mDY \) to a randomly chosen value between \( \text{min}_dy \) and \( \text{max}_dy \).

**serveRight**

Recieves several parameters. See the UML diagram for the full list. Sets the ball’s position using the \( x \) parameter and a \( y \)-value randomly chosen between \( \text{min}_y \) and \( \text{max}_y \). You may want to look at the \( \text{random.uniform()} \) function. Sets the ball’s \( mDX \) to a randomly chosen value between \( -\text{min}_dx \) and \( -\text{max}_dx \). Sets the ball’s \( mDY \) to a randomly chosen value between \( \text{min}_dy \) and \( \text{max}_dy \).

**draw**
Uses PyGame to draw the rectangle for the ball. There are no unit tests for this method. It will be verified during the pass-off of the full game.

**Part 2**

This part of the assignment requires the addition of classes for Paddle, Wall, ScoreBoard and Pong. Each of the classes has required data members and methods. The updated UML Diagram contains all of the classes and their required methods. Not all data members or methods will be discussed below. If you have questions, ask.

All Pong unit tests.

**Paddle class**

```
__init__
```

Initialize the paddle data members from the parameters. \( \text{min}_y \) and \( \text{max}_y \) refer to the top and bottom of the field of play.

**Getters**

Implement the getters.

```
getRightX
```

Returns the x coordinate of the right side of the paddle.

```
getBottomY
```

Returns the y coordinate of the bottom of the paddle.

```
setPosition
```

Updates the y position of the paddle. If the new y position would cause the top of the paddle to go past the top or the bottom of the paddle to go past the bottom, do not make any changes.

```
moveUp
```

Updates the y position of the paddle based on the time \( dt \), and the paddle’s speed. If the paddle would move past the top of the allowed region, stop at the top.

```
moveDown
```

Updates the y position of the paddle based on the time \( dt \), and the paddle’s speed. If the paddle would move past the bottom of the allowed region, stop at the bottom.

```
draw
```

Uses PyGame to draw the rectangle for the paddle. There are no unit tests for this method. It will be verified during the pass-off of the full game.

**Wall class**

```
__init__
```

Initialize the wall data members from the parameters.

**Getters**

Implement the getters.

```
getRightX
```

Returns the x coordinate of the right side of the wall.
**getBottomY**

Returns the y coordinate of the bottom of the wall.

**draw**

Uses PyGame to draw the rectangle for the wall. There are no unit tests for this method. It will be verified during the pass-off of the full game.

**ScoreBoard class**

```python
__init__
```

Initialize the data members from the parameters. Set `mLeftScore` and `mRightScore` to 0. Set mServeStatus to 1, which means it is the left player’s turn to serve. The `mServeStatus` data member can be 1: left’s turn to serve, 2: right’s turn to serve, 3: left has won or 4: right has won.

**Getters**

Implement the getters.

```python
isGameOver
```

If the `mServeStatus` indicates the game is over, return `True`. Otherwise, return `False`.

```python
scoreLeft
```

Give a point to the player on the left. If the left player’s score is 9, then set status to left player win. This method should make no changes if the game is already over.

```python
scoreRight
```

Give a point to the player on the right. If the right player’s score is 9, then status to right player win. This method should make no changes if the game is already over.

```python
swapServe
```

If the serve status is left serve, change it to right serve. If it is right serve, change it to left serve. If the game is already over, do not change anything.

```python
draw
```

Uses PyGame to draw the score in the area defined by the data members. Use the `Text` class provided with the starter code to draw text. There are no unit tests for this method. It will be verified during the pass-off of the full game.

**Pong class**

There is a `Pong` class included with the starter kit download. It uses all of your classes to implement the game. Your acceptance test is to demonstrate the working game.

**Extra Challenges**

- Define an end game (e.g. first to 9 points wins).
- Display the winner.
- Add a start screen, and allow the user to start the game.
- Add a restart option to the game so the player doesn’t have to exit the application and start it again to restart game play.
- Add sound.
- Add images for display.
Hints

- Refer to the Pygame documentation to understand which parameters are necessary when calling each of the Pygame draw methods. Specifically, you should be interested in `pygame.draw` and `pygame.Rect`.

- When creating colors, use a helpful tool to determine the RGB values. Here are two good options: color.adobe.com and colorpicker.com