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.

Download 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. The smaller of the two parameters should be treated as the miny and the larger as the maxy.
**setRightPaddleY**

Updates the \( \text{mRightPaddleMinY} \) and \( \text{mRightPaddleMaxY} \) data members, but only if the new values are within the minimum and maximum values specified by the data members. The smaller of the two parameters should be treated as the \( \text{miny} \) and the larger as the \( \text{maxy} \).

**checkTop**

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

![Diagram of Ball Class Check Top Method]

\[
\Delta y = \Delta y_1 + \Delta y_2 = y - \text{new}_y \\
\Delta y_1 = y - \text{miny} \\
\Delta y_2 = \text{maxy} - \text{new}_y \\
\text{new}_y'' = \text{miny} + \Delta y_2 \\
dy = -dy
\]

**checkBottom**

Receives the proposed \( \text{new}_y \) value for the ball. If the new \( y \) value would not cause the ball to bounce from the bottom wall, then return \( \text{new}_y \) unchanged. If the value would cause the ball to bounce, then reverse the sign of \( \text{mDY} \), calculate the corrected \( \text{new}_y \) 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 \( \text{new}_x \) value for the ball. If the new \( x \) value would not cause the ball to touch the left wall, then return \( \text{new}_x \) unchanged. If the value would cause the ball to touch, then stop the ball, calculate the corrected \( \text{new}_x \) value and return it. Note that this will cause the ball to stick to the wall where it touches.

**checkRight**

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

move

Receives \( \text{dt} \), the amount of seconds that have passed since the last frame. Uses \( \text{mX} \), \( \text{mDX} \) and \( \text{dt} \) 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 \( \text{mDX} \) and/or \( \text{mDY} \) if necessary. \text{move} doesn’t need to worry about it. Finally sets \( \text{mX} \) and \( \text{mY} \) 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 \( \text{x} \) 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 \( \text{mDX} \) to a randomly chosen value between \( \text{min}_dx \) and \( \text{max}_dx \). Sets the ball’s \( \text{mDY} \) 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 \( \text{x} \) 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 \( \text{mDX} \) to a randomly chosen value between \( -\text{min}_dx \) and \( -\text{max}_dx \). Sets the ball’s \( \text{mDY} \) 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**

The second part will require additional classes. Details to come.

**Extra Challenges**

- Define an end game (e.g. first to 10 points wins).
- 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 a high score as well as the current score. This requires that the restart option is available.
- Add a high score file so that the game can remember the best score ever, not just the best score since the program was launched.
- 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