CS 1410: Pong

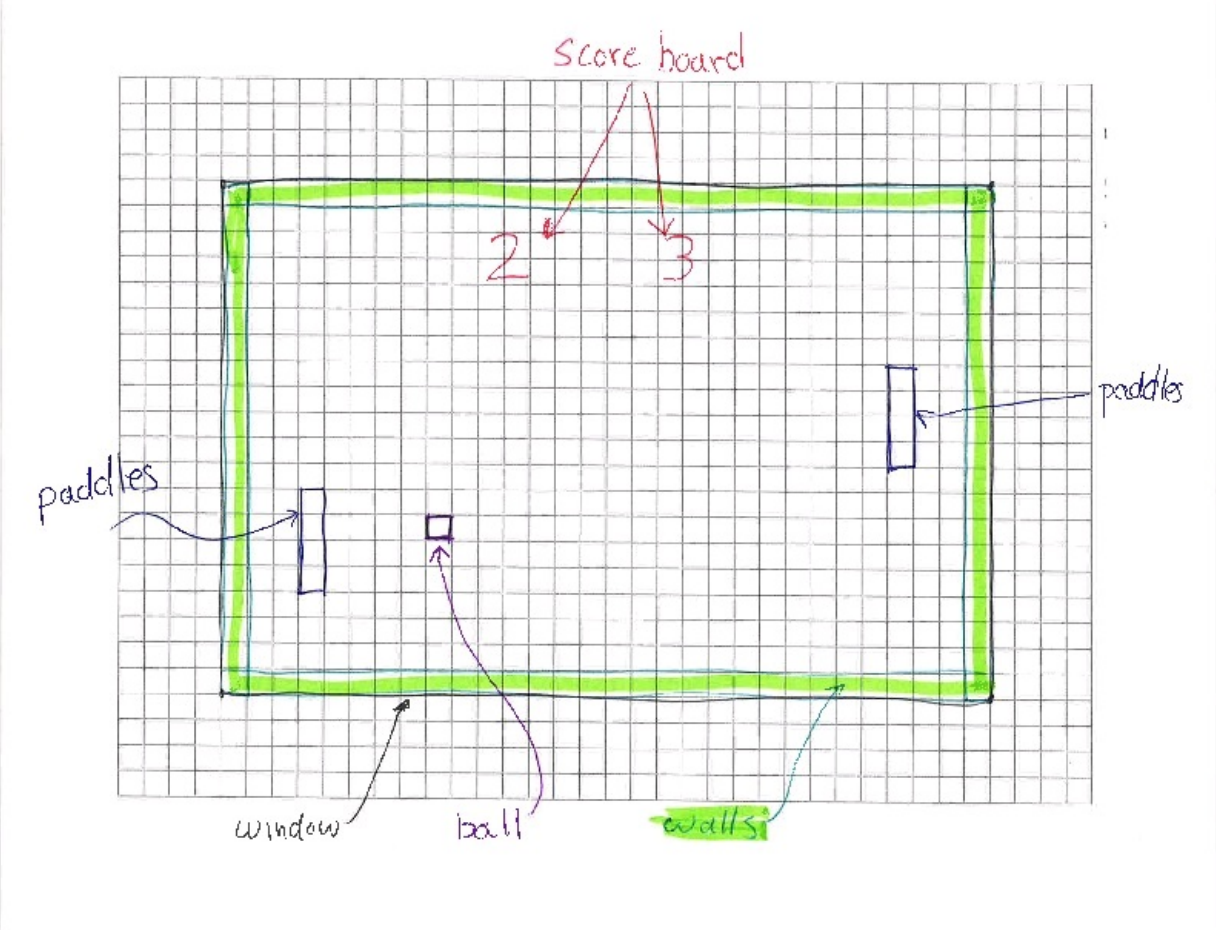
In computer science, classes and object-oriented programming are very commonly used when creating programs that involve computer graphics and graphical user interfaces. To display something visually on the screen, like a button, a class is used to represent a Button, and multiple instances of the Button 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. In the first part you are required to create the `Ball` class. We have provided Ball class unit tests.

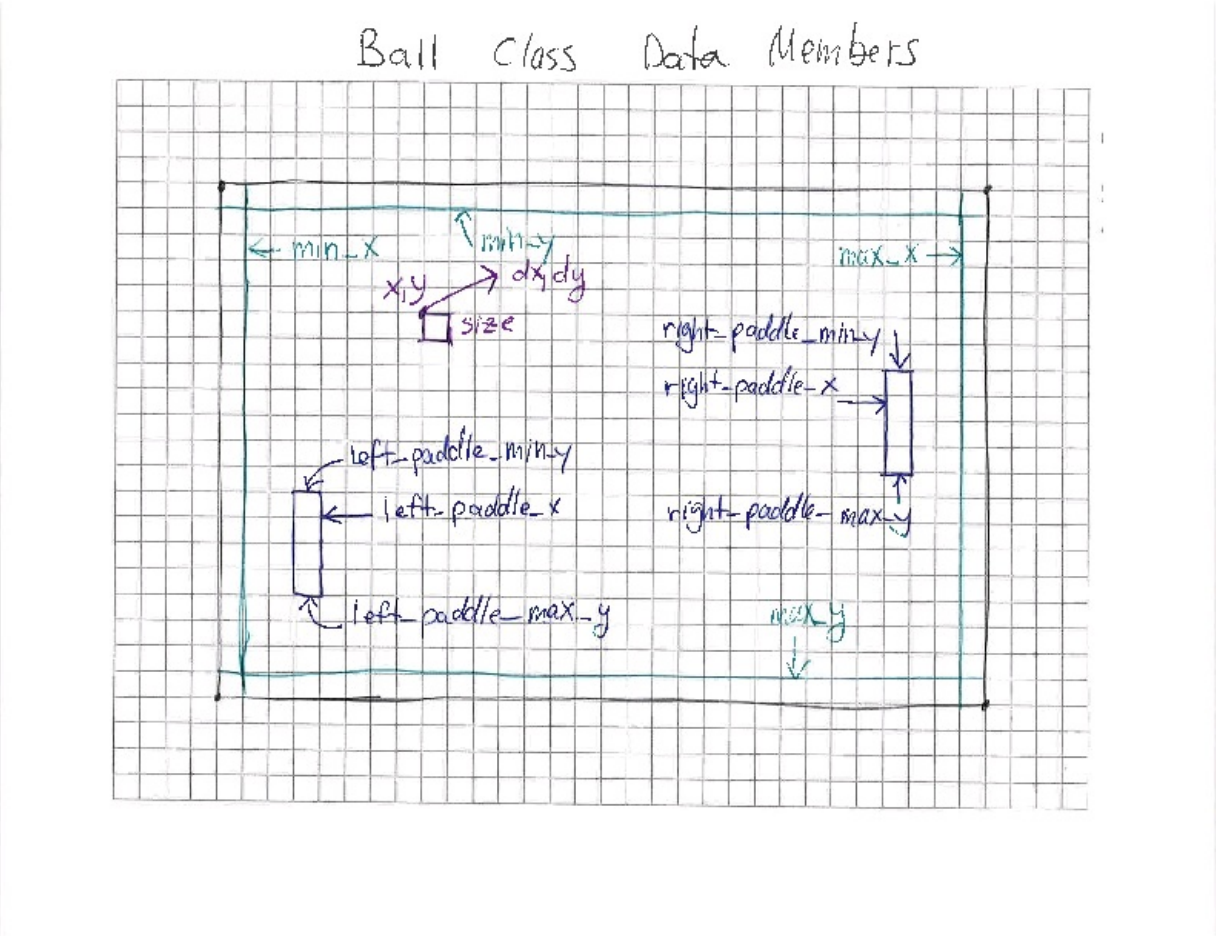
The `Ball` class has a large number of data members, getter methods for most of the members and about a dozen methods to handle the details of the ball's movement and interactions with the paddles and walls.

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. The top section lists the required data members. Note that UML does not display `self.`, as it is implied. The bottom section lists the required methods, their names, and their parameter lists. Note that UML does not display the `self` parameter, as it is implied. The `+` at the front of each line means that this method is intended to be used by other parts of the program. We use the term "public". The data members do not have a `+`. This is because other parts of the program should not use the data members directly.

This browser does not support PDFs. Please download the PDF to view it: Download PDF.

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.

`mX` and `mY`

This is the position of the top-left corner of the ball, measured in pixels. The values may be floating point numbers.

`mSize`

This is the length, in pixels, of the sides of the ball. The ball is a square.

`mDX` and `mDY`

This is the speed of the ball. DX is short for delta-x, and DY for delta-y. These are measured in pixels per second.

`mMinX`

This is the smallest value that `mX` may be set to. It represents the wall on the left side of the screen.

`mMaxX`

This is the largest `X` value that any part of the ball can have. It represents the wall on the right side of the screen. Remember that the ball is `mSize` pixels wide. The largest value of `mX` must be `mSize` less than `mMaxX`.

`mMinY`

This is the smallest value that `mY` may be set to. It represents the wall on the top of the screen.

`mMaxY`

This is the largest `Y` value that any part of the ball can have. It represents the wall on the bottom of the screen. Remember that the ball is `mSize` pixels high. The largest value of `mY` must be `mSize` less than `mMaxY`.

`mLeftPaddleX`

This represents the side of the left paddle that the ball may bounce from.

`mLeftPaddleMinY`

This represents the top of the left paddle.

`mLeftPaddleMaxY`

This represents the bottom of the left paddle.

`mRightPaddleX`

This represents the side of the right paddle that the ball may bounce from.

`mRightPaddleMinY`

This represents the top of the right paddle.

`mRightPaddleMaxY`

This represents the bottom of the right paddle.

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__(size, min_x, max_x, min_y, max_y, left_paddle_x, right_paddle_x)`

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 mininum 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.

Getters

Implement all of the getters shown in the UML diagram.

`setPosition(x, y)`

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. If either of the new values is incorrect, do not change anything.

`setSpeed(dx, dy)`

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

`setLeftPaddleY(paddle_min_y, paddle_max_y)`

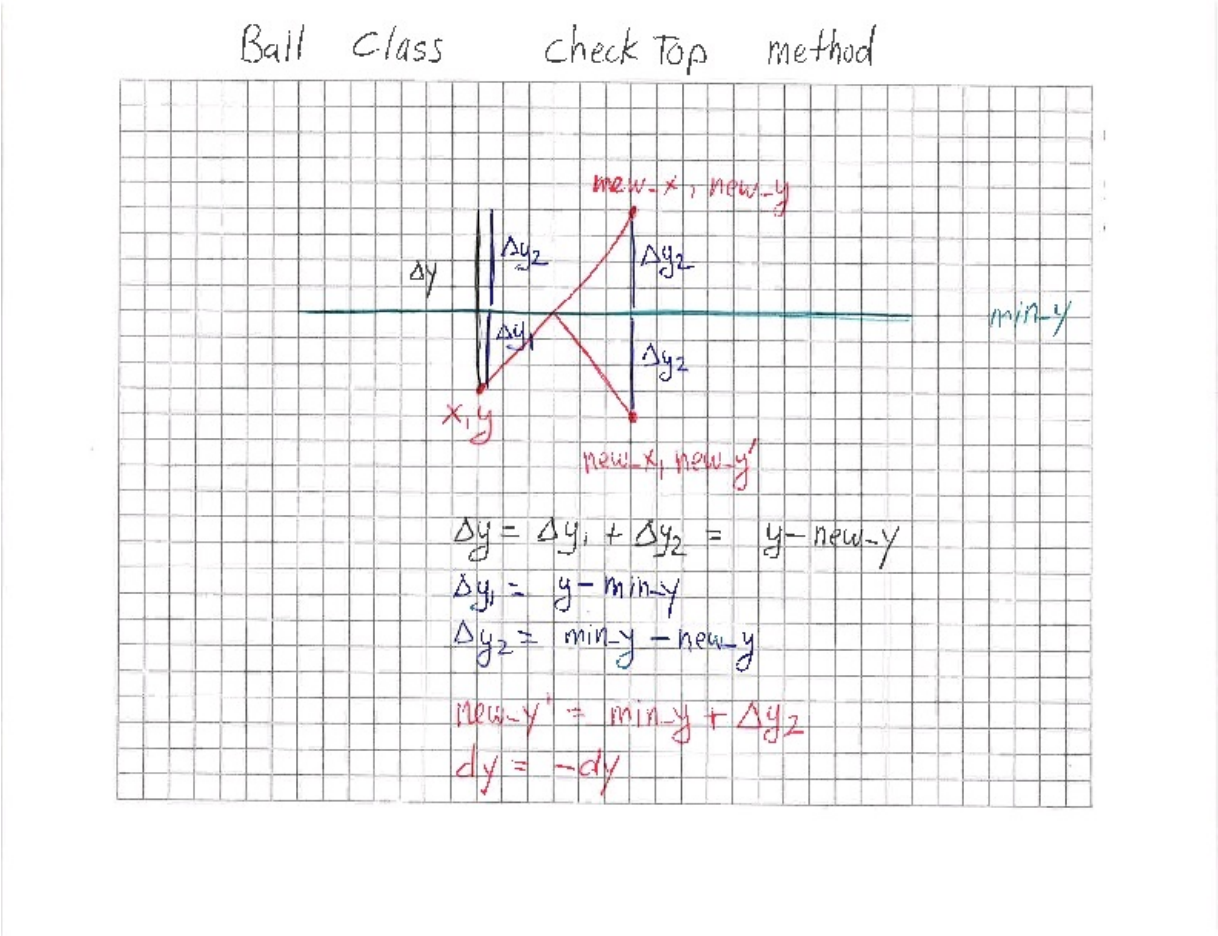
Updates the `mLeftPaddleMinY` and `mLeftPaddleMaxY` data members, but only if the new values are within the minimum and maximum values specified by the data members. Only sets the two data members if the parameters are valid. This means the minimum must not be less than `mMinY` and the maximum must not be more than `mMaxY`. Also, the minimum must be less than the maximum.

`setRightPaddleY(paddle_min_y, paddle_max_y)`

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. See notes in `setLeftPaddleY`.

`checkTop(new_y)`

Receives the proposed `new_y` value for the ball. If traveling from the current y position to the new y position 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 the corrected value. The picture below may help.



`checkBottom(new_y)`

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 the corrected value. This is similar to `checkTop`, but you need to include the ball's size in your calculations. This is because the bottom of the ball will bounce from the bottom wall. This is similar to the way `checkRight` accounts for the right side of the ball touching the right wall.

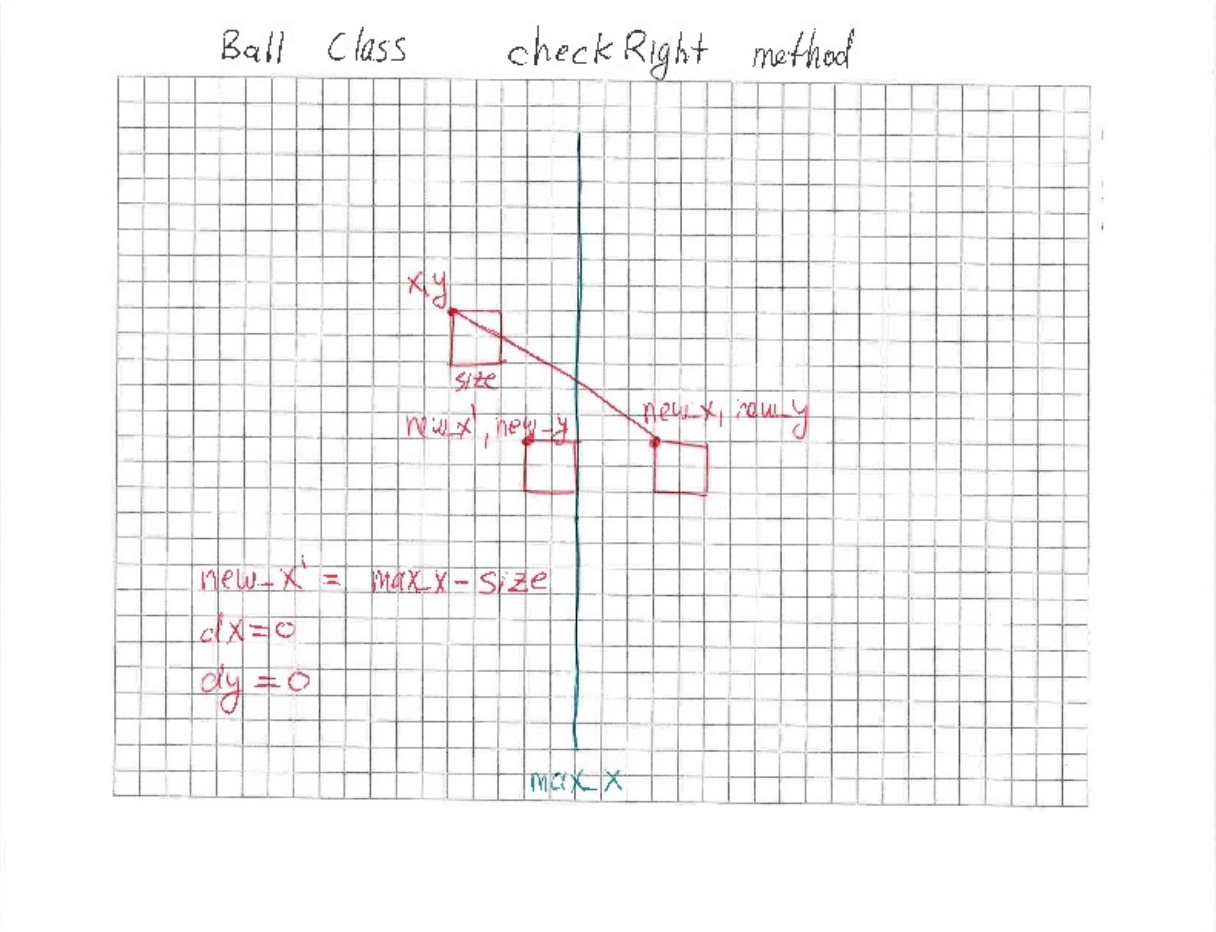
`checkLeft(new_x)`

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(new_x)`

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.



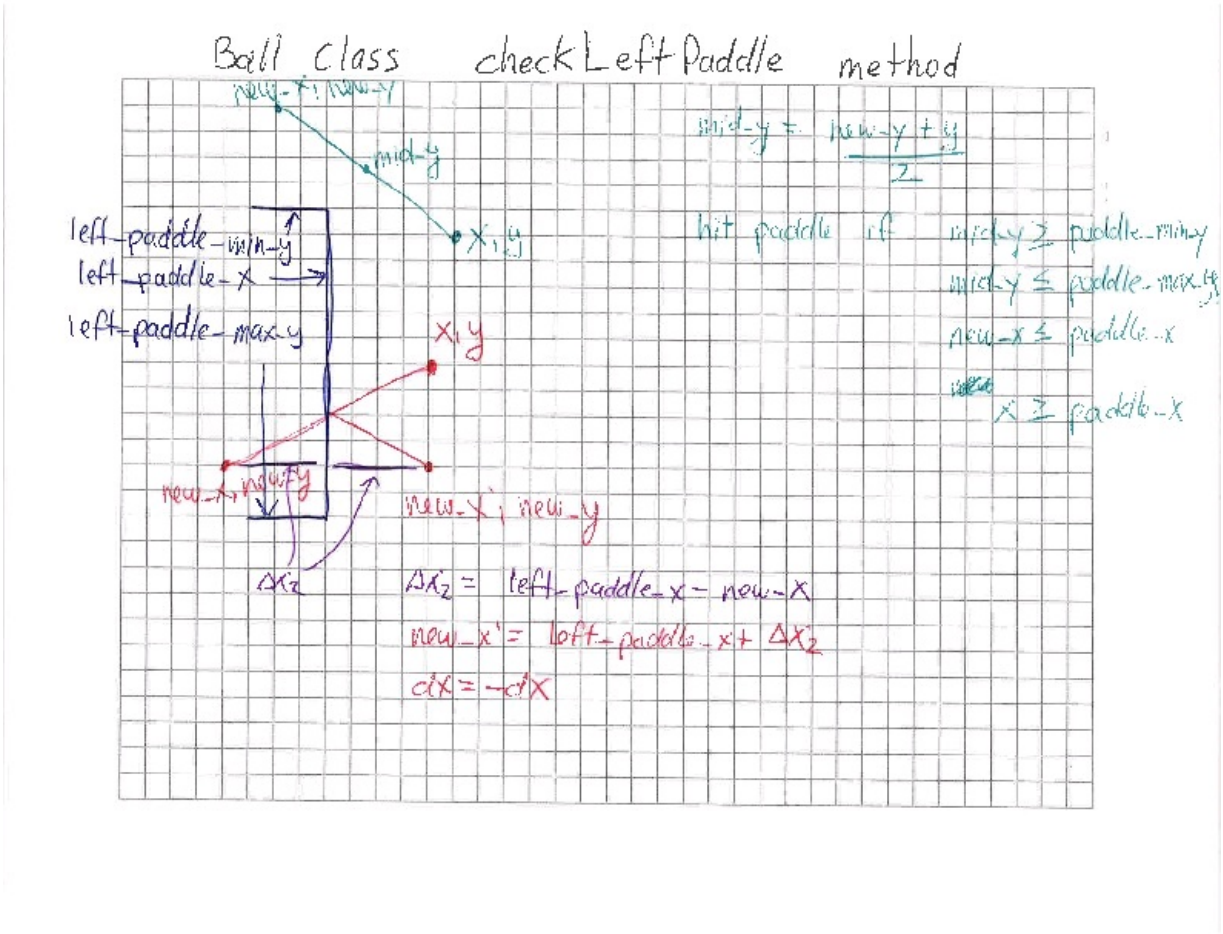
`checkLeftPaddle(new_x, new_y)`

Receives the proposed `new_x` and `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 `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 `new_x` value and return it.

If the ball's x coordinate is already to the left of the paddle's coordinate, there is no collision. If the ball is moving right, there is no collision.

To touch the paddle, the ball's `mid_y` value must be between the paddle's minimum and maximum y values. `mid_y` is the average of the ball's current y position, and the new y position. The picture below may help.

Note that this is not a perfect collision algorithm, but it will suffice for the game we are creating.



`checkRightPaddle(new_x, new_y)`

Receives the proposed `new_x` and `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 `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 `new_x` value and return it.

To touch the paddle, the ball's `mid_y` value must be between the paddle's minimum and maximum y values.

`move(dt)`

Receives `dt`, the amount of seconds that have passed since the last frame. Uses `mX`, `mDX` and `dt` to calculate `new_x`, the proposed new x position of the ball. Does similarly to calculate `new_y`. Uses `checkTop`, `checkBottom`, `checkLeft`, `checkRight`, `checkLeftPaddle` and `checkRightPaddle` to update the values of `new_x` and `new_y`. Note that these methods will also change the sign of `mDX` and/or `mDY` if necessary. `move` doesn't need to worry about it. Finally sets `mX` and `mY` from the possibly updated values of `new_x` and `new_y`.

`serveLeft(x, min_y, max_y, min_dx, max_dx, min_dy, max_dy)`

Recieves several parameters. Sets the ball's position using the `x` parameter and a y-value randomly chosen between `min_y` and `max_y`. You may want to look at the `random.uniform()` function. Sets the ball's `mDX` to a randomly chosen value between `min_dx` and `max_dx`. Sets the ball's `mDY` to a randomly chosen value between `min_dy` and `max_dy`

`serveRight(x, min_y, max_y, min_dx, max_dx, min_dy, max_dy)`

Recieves several parameters. Sets the ball's position using the `x` parameter and a y-value randomly chosen between `min_y` and `max_y`. You may want to look at the `random.uniform()` function. Sets the ball's `mDX` to a randomly chosen value between `-min_dx` and `-max_dx`. Sets the ball's `mDY` to a randomly chosen value between `min_dy` and `max_dy`

`draw(surface)`

Uses PyGame to draw the rectangle for the ball. There are no unit tests for this method. It will be verified during the acceptance tests for 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.

This browser does not support PDFs. Please download the PDF to view it: Download PDF.

All Pong unit tests.

`Paddle` class

Paddle Data Members

`mX` and `mY`

This is the position of the top-left corner of the paddle, measured in pixels. The values may be floating point numbers.

`mWidth` and `mHeight`

This is the horizontal and vertical size of the paddle's rectangle, measured in pixels.

`mSpeed`

This is the vertical speed of the paddle, measured in pixels per second.

`mMinY` and `mMaxY`

These describe the position of the top and bottom walls. The paddle may not cross into either wall.

Paddle Methods

`__init__(x,y,width,height,speed,min_y,max_y)`

Initialize the paddle data members from the parameters. `min_y` and `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(y)`

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

`moveUp(dt)`

Updates the y position of the paddle based on the time `dt`, and the paddle's speed. If the paddle would move

into the top of the allowed region, stop at the top.

`moveDown(dt)`

Updates the y position of the paddle based on the time `dt`, and the paddle's speed. If the paddle would move into the bottom of the allowed region, stop at the bottom.

`draw(surface)`

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

Wall Data Members

`mX` and `mY`

This is the top-left position of the wall, measured in pixels.

`mWidth` and `mHeight`

This is the horizontal and vertical size of the wall, measured in pixels.

Wall Methods

`__init__(x,y,width,height)`

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(surface)`

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

ScoreBoard Data Members

`mX` and `mY`

This is the top-left corner of the rectangle that contains the score board. Measured in pixels.

`mWidth` and `mHeight`

This is the horizontal and vertical size of the rectangle that contains the score board. Measured in pixels.

`mLeftScore` and `mRightScore`

These are the numeric scores of the left and right players. Stored as integers, measured in points.

`mServeStatus`

This records information about which player should serve next. `1` means left player serves next. `2` means right player serves next. `3` means left has won. `4` means right has won.

ScoreBoard Methods

`__init__(x,y,width,height)`

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.

`isGameOver()`

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

`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.

`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.

`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.

`draw(surface)`

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