Pong Engine

Aryan Nehete

Goal

Return "up" or "down" based on which direction I should move my paddle to play Pong.

Information

1. `paddle_frect`: rectangle representing my paddle. Properties: `paddle_frect.pos[0]`, `paddle_frect.pos[1]` represent the coordinates of the top-left corner, and `paddle_frect.size[0]`, `paddle_frect.size[1]` are the dimensions of the paddle along the x and y axes.
2. `other_paddle_frect`: rectangle representing the opponent's paddle, with the same properties.
3. `ball_frect`: rectangle representing the ball, formatted the same way as `paddle_frect`.
4. `table_size`: `table_size[0]`, `table_size[1]` are the dimensions of the table along the x and y axes.

Coordinate System: (0,0) at top left; (`table_size[0]`, 0) at top right; (0, `table_size[1]`) at bottom left; (`table_size[0]`, `table_size[1]`) at bottom right.

Figure 1: Game board, with coordinate system in blue, paddle properties in green, ball properties in red.

Algorithm Overview

Reframe to centers and ball bounds. Start by reframing the information to use centers of components (center of paddles and ball), and define convenient bounds for the ball to simplify collision and intersection math.

Figure 2: Reframed coordinates to centers for simpler geometry and intersections. Also added theoretical "bounds" in orange which contain the ball's center & align with new coordinates.

Track trajectory with position. Store successive ball positions in two lists, `ballXs` and `ballYs`. Using the current and previous positions, compute the velocity vector to determine its direction (the ball moves in a straight line between bounces). For example,

$$\text{ballXs} = [..., 213, 214] \tag{1}$$

$$\text{ballYs} = [..., 105, 104] \tag{2}$$

$$\text{(1) and (2) imply that the ball is moving in the } \begin{bmatrix} 214 - 213 \\ 104 - 105 \end{bmatrix} = \begin{bmatrix} 1 \\ -1 \end{bmatrix} \text{ direction.}$$

In reality, the positions are higher precision floats, ensuring that the direction is accurate and allows for more advanced calculations.

Predict next bounces. Compute where the ball will hit a wall and bounce (if applicable) using a function `get_next_bounce(pos,dir)` that returns the collision point.
Then, `get_bounces` repeatedly calls `get_next_bounce`, updating the ball position and direction after each bounce (assuming it occurs) to forecast the path until it reaches my paddle x-coordinate. The y coordinate where the ball will collide with my paddle is `targetY`.

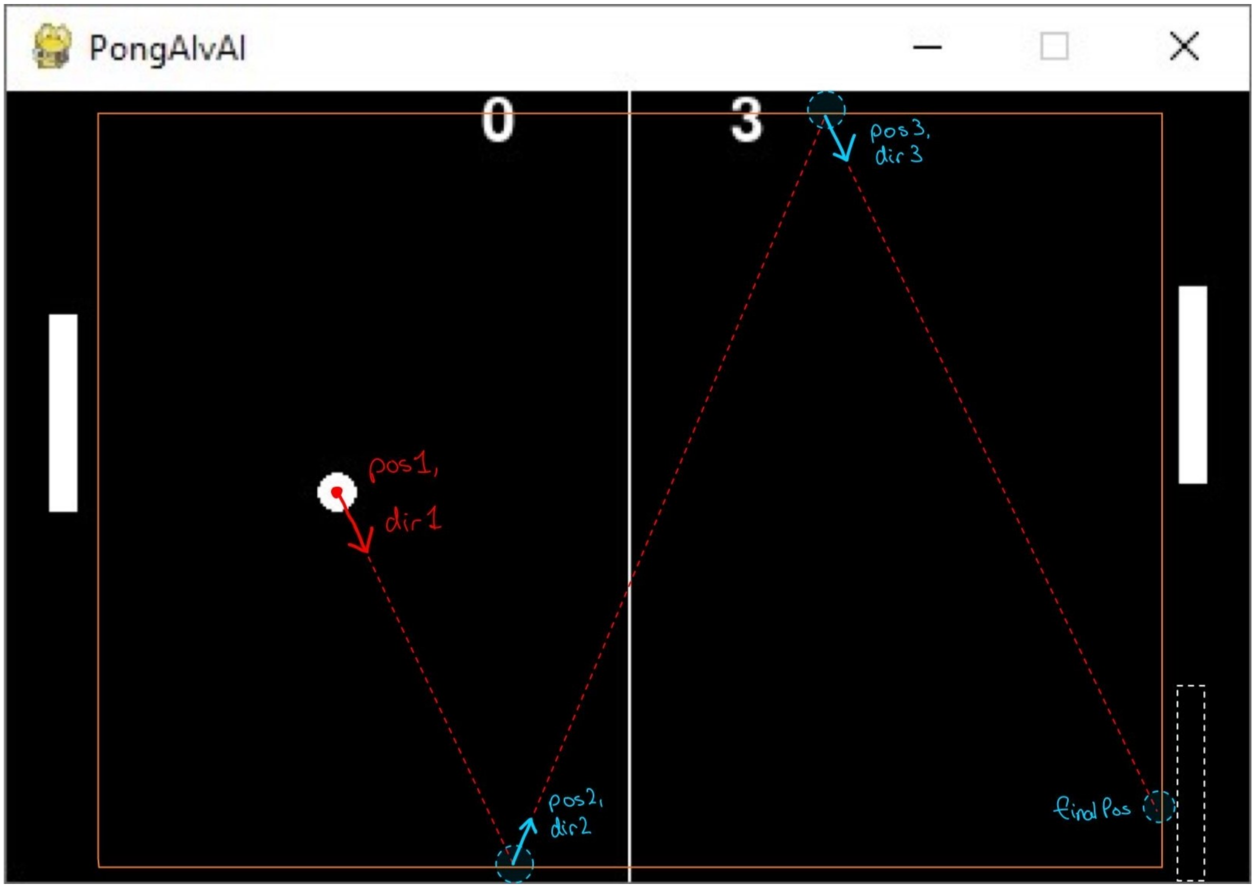

Figure 3: Projection of multiple bounces until ball intersects with my paddle. `get_bounces` starts by calling `get_next_bounce(pos1,dir1)`, then `get_next_bounce(pos2,dir2)` and more if applicable.

Intercept at the paddle. Move my paddle to the y-position where the predicted ball path intersects my paddle's vertical line and hold it there until contact.

Exploit opponent position and paddle "shape". The current strategy is good, but does not utilize all the available information; it ignores the opponent's paddle. A Pong paddle behaves like a bracket-shaped collider in code, not a perfect rectangle. By positioning my paddle precisely, I can target specific outgoing angles.
Ideally, I would send the ball as far away from my opponents paddle as possible. By reading the opponent paddle position, I can target the corner farthest from it. Using trigonometry, compute the necessary outgoing angle. Then, by reverse-engineering the game's paddle collision rules (the effective "shape"/contact mapping), I compute how far up/down to place my paddle to achieve that trajectory.

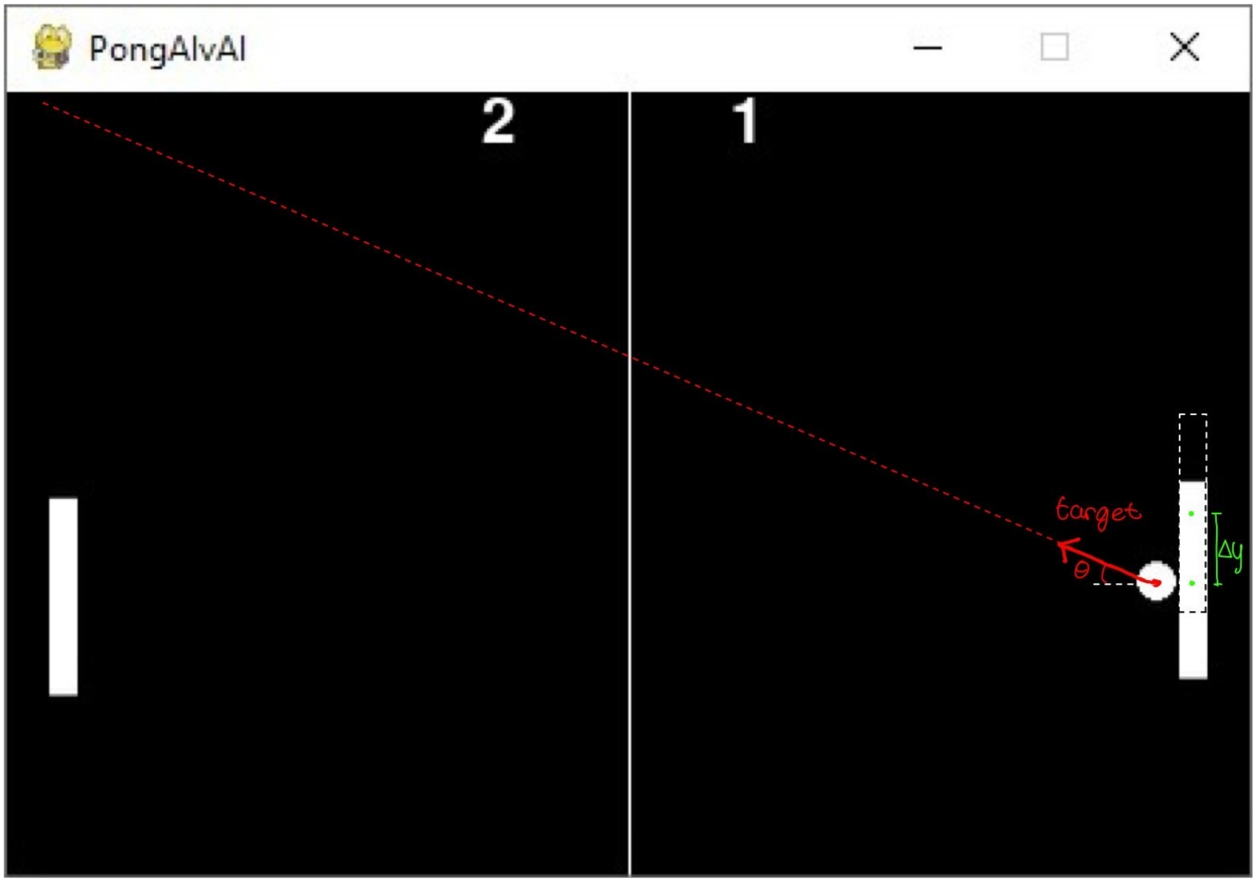

Figure 4: Aiming strategy: calculate θ as a function of target coordinates, ball position, and opponent paddle position, then calculate Δy as a function of θ.

Finalize positioning. Finally, set my paddle's final target as `finalTargetY` $=$ `targetY` $+ \Delta y$. If my paddle's y position is below `finalTargetY`, return `"up"`; otherwise `"down"`.