Karol Vargas

Austin Bristol

Activision JV squad presents:

Pymunk Soccer

The motivation for us to do this is because we found it would be very interesting to implement an AI to a game that we both have played a few times called bonk.io. It is pretty much a 2D soccer game with a physics engine but does not have an AI. Instead it requires another player to connect to the same game lobby. Another reason we wanted to create this AI was because we thought it would be a good way to be able to use some of the methods we learned in class and have some fun with it. Some of the other things that have already been done with bonk are shooting with the spacebar instead of having to push the ball into the goal. We had this function in the first version of our game but we decided that this function would not be possible to implement with our AI with the time constraint so we decided it best to remove it.

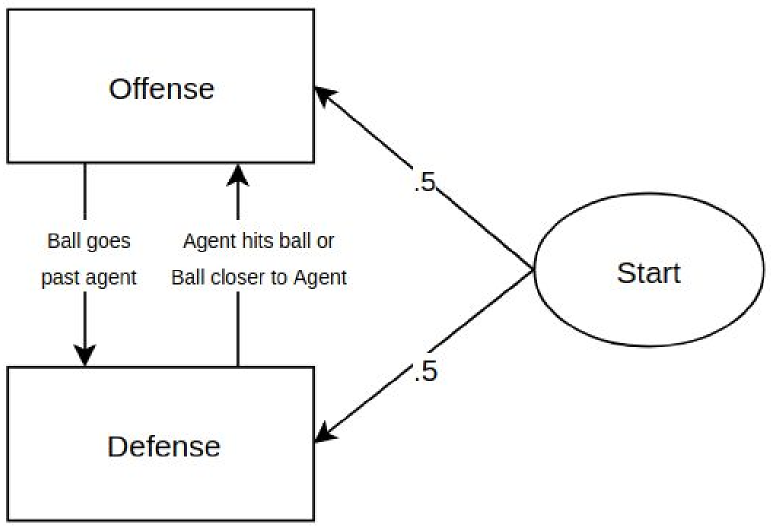
The first step we decided to do was find a library that would be able to create a 2D environment so we can draw a pitch with two goals. The next requirement we decided to assign was create a physics engine. This was so the ball could move and bounce off the out of bounds of the soccer pitch. We decided instead to research for a physics library. We decided on pymunk because its physics library fit our program requirements the best. Our first step with implementation was to a physics environment with pymunk. This was a large chunk of code as we had to specify what the out of bounds were in the environment to ensure that the ball did not slide off the users screen. The way the line segments work in pymunk is using the static_lines class and specify the two endpoints of the line. After you specify the two endpoints of the line you then decide how thick you want the lines to be.

We created a rectangular out of bounds arena with static bodies that would not allow the ball to pass through them. We placed these borders in our createBorder class. The parameter takes in the space variable we created in our set up class which is a pymunk object that allows us to use the physics engine in pymunk. In our draw_lines function this is where we take in our same space parameter and create lines that only appear on the screen but do not affect any objects that are static. These are for show and we used them to draw the line for for the box of the two goals and also creating the midline of the pitch. The next thing we created was the actual ball. This was a static object which could be affected by other static objects but still goes over the screen lines as well.

The next thing that we wanted to implement was a goal check function. The way we did this was to create an if else saying if the balls location on the x axis was greater than 675 it was the goal on the right side which is team 2. If the balls x location was less than 25 then it was team 1s goal which is the goal on the left side of the space. The next part of the program was to add a make impulse function into the program. This basically allows us to use the physics library to calculate the angle and at what magnitude the object will reflect if it hits a static object. The function will return true whenever a physics object and static object collide.

Another function we had to implement was the reset ball function. This is the part of the program that is really where score checking comes into play. Whenever the ball crosses the boundary of the two x points that we had put into our goal check function, the score counter will add one depending on which side the ball was scored. We then destroy that physics object, and spawn another ball in the very center of the pitch. The game is up to three but if you must win by two to win the game.

One of the more integral parts of this final project was the artificial intelligence. This part of the project took lots of tinkering and trial and error. It was added difficulty that we were not using any external libraries that aided in the development of the AI. All we had at our disposal were the functions of the Pygame and Pymunk libraries. Our approach to the AI was relatively simple. We wanted the agent to be able to be on offense or defense depending on the current situation. For this type of approach we used a finite state machine (FSM) much like the one we used earlier in the year. In the figure below, you can see our finite state machine.



As you can see in the diagram for our finite state machine, the agent has two possible states. These states are either offense or defense. At the start of the finite state machine, that is when the game starts or after a goal is scored, there is a 50% chance that the agent will start on offense and a 50% chance that the agent will start on defense. After this decision occurs, there are no longer random probabilities used. In order to transition states, certain conditions must be met. In the defensive state, the AI sits on the goal line while following the ball on the y-axis, which protects the ball from the goal. In the offensive state, the AI will follow the location of the ball while trying to get further on the outside of it in order to create an angle to shoot.

In order to switch from the offensive state to the defensive state, the ball simply has to go past the agent's location towards the goal. More specifically, if the soccer ball's x-location is greater than the agent's x-location, then the agent will switch to the defensive state by immediately returning to the goal line to defend the goal. When in the defensive state, there are two conditions which will switch the agent to the offensive state. First, if the agent hits the ball in an attempt to clear the ball from the net, it will switch to the offensive state. Also, if the agent is closer to the ball than the user, by a margin of 100 or more, then they will switch to the offensive state. The reason that this is a condition is that if the bot can beat the user to the ball easily, they should go on offense. This is the finite state machine that we implemented in our game.

Next, we will go over exactly how the artificial intelligence was implemented into our game. First, we should go over the functions which aid in the artificial intelligence. One function is *get_distance* function. This function has two parameters, which are both pymunk Body objects; objects like the players and the soccer ball. The function returns the distance between the two objects using the classic distance function seen. $\sqrt{(x_1 - x_2)^2 + (y_1 - y_2)^2}$. This is useful for hitting the ball and making other decisions.

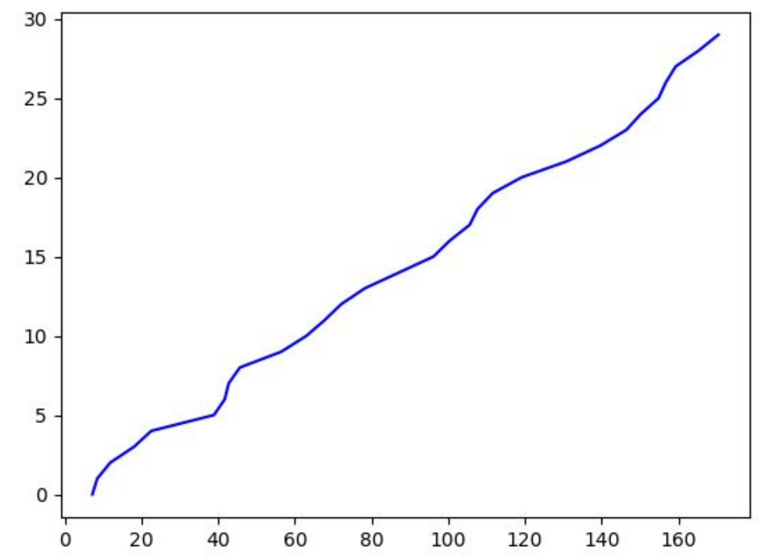
The next set of functions return a list of commands that the agent wants to carry out. In order to actually move the agent around, we must emulate a series of key presses. This is because the move function that is used in the code expects a tuple of key presses. This means that whether or not the character pressed to go up, left, right, or down. This function will take these key presses and change the directional speed of the agent and the location of the agent accordingly. In order to make the AI agent actually perform actions. We hold a list of these key presses and feed them into the move function which actually moves the agent around the field.

The first of the functions which actually produces a series of key presses to make the agent perform different actions is called *path_find.* This function takes the ai player's pymunk body object, as well as the goal location where the agent wants to go. Also, it takes information about how fast the agent is currently moving in the game. It first determines the y-distance that the agent wants to go which is simply the difference between the agent's current location and the goal location. Then, it simulates the agent's movement on this y-axis. To do this, we continue to simulate the location of the agent by adding the current speed that the AI is moving. By doing this, we provide a more accurate pathfinding. Based on this simulation, we know how many key presses up or down the bot will need to do to get to where we want to go. We do a very similar process for finding the x-axis distance and the amount of key presses left or right we need to do. By adding all the key presses from the simulated movement, we can provide a list of key presses to get to the goal location. We use this key press list by sending the first index of the list to the move function and then popping it off so that we do not continue to do the same movement. This process is only stopped when there are no more keys to press or another event occurred which causes us to change the agent's current action. The condition for an event to occur which resets the current keys to press is pretty lengthy. What this event comes down to is whether or not the agent's state changes, or the ball moves far enough from the current location.

Another function we created was a shoot function. This event occurs any time the ball is within striking distance from the agent. When this happens, the *shoot* function creates 13 key presses in the direction of the ball. For example, if the ball is below the agent, they will move in a diagonal direction left and down in order to hit it better.

These functions are called based on the current agent's state, and where the ball is. These help the AI agent complete the goal of either saving the ball from their own net, or trying to shoot on the opponent's goal. The core logic where these functions are called from can be found starting at line 394.

We also collected various results from our agent in order to see how it was doing during some play tests. The x-axis in this diagram is the number of seconds into the game we are, and the y-axis is the number of state changes that have occured. This means the number of times the agent has switched from offense to defense or vice versa.



As you can see in the diagram, the amount of state changes steadily increases throughout the game. More specifically, this data helps to see more so what the opponent or human player is doing. When ever the opponent becomes more aggressive in their play, there is a large increase in the number of state changes. This means that the user shoots on the goal and then right away tries to keep the ball in the agent's zone in order to score it again. This causes the agent to switch from defense to offense and back again very quickly in order to protect their net. Likewise, flatter areas on the graph means that the user is not shooting the ball that quickly or is taking their time.

We faced plenty of challenges while developing this software system. One challenge we faced was getting the physics of the game to work. We struggled to find a good library to do this;

we even tried to implement our own physics. We eventually settled with Pymunk, where we still had to finagle the settings a lot in order to get the physics to work in the most logical way possible. The physics also didn't work great around walls; we had to add additional code to ensure the ball did not phase through the side walls and out of the play area. Another challenge we faced was getting pathfinding to work. We first tried to have the key presses mimic the rise and run of the slope between the agent and its destination. This worked but the agent went very slow, so we ended up having the agent travel on the y-axis first and then on the x-axis in order to keep its momentum. We made the agent overshoot its y-axis distance along with a shoot function in order for the agent to create an angle to shoot the ball on the net. The final challenge we ran into is making the players move smoothly. It was rigid before, so we implemented speeds which accelerated and decelerated in all four directions to provide a smoother experience. Overall, the project turned out very good, and it provides a game which is very playable against a competent opponent. We gained a lot of experience with python, creating games, developing AI, and working with physics. A demo of our game in action can be found by following the link below.

Demo:

https://www.youtube.com/watch?v=c2HdOJEepuU&feature=youtu.be