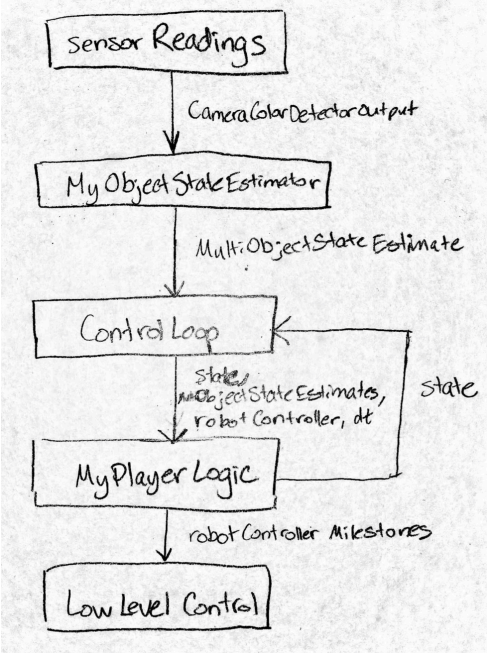
Austin Hua

System Design

A



The system first proceeds with perception by taking in the data from the camera sensors. It turns this data into estimations of object states. These are passed along to the planning part of the system where the estimates are given to the high-level control loop. The logic then plans out what the next state will be using projections based on these estimates and information from the previous states. These decisions are sent to the low level control part of the system in the form of milestones for the robot controller to carry out.

B

The sensor readings are the data that comes in from the camera detectors. The input to the sensor readings is the visual data that comes in from the camera in the form of pixel readings of a 240x320pixel dimension screen. The reference frame is the frame of the camera sensor, which in this case is located slightly behind the red ball and the robot. The sensor data outputs a list of blobs representing objects of interest based on color detected from the readings. The output format is the CameraColorDetectorOutput, which is a list of blobs categorized by their color containing the (x, y) integer position in pixels on the camera's image plane, and the integer height and width of the blob in pixels as well. This is all in the reference frame of the camera image plane. The sensor readings are read constantly each time step. Potential failure cases would be when objects of interest in the environment go out of the view of the camera sensor.

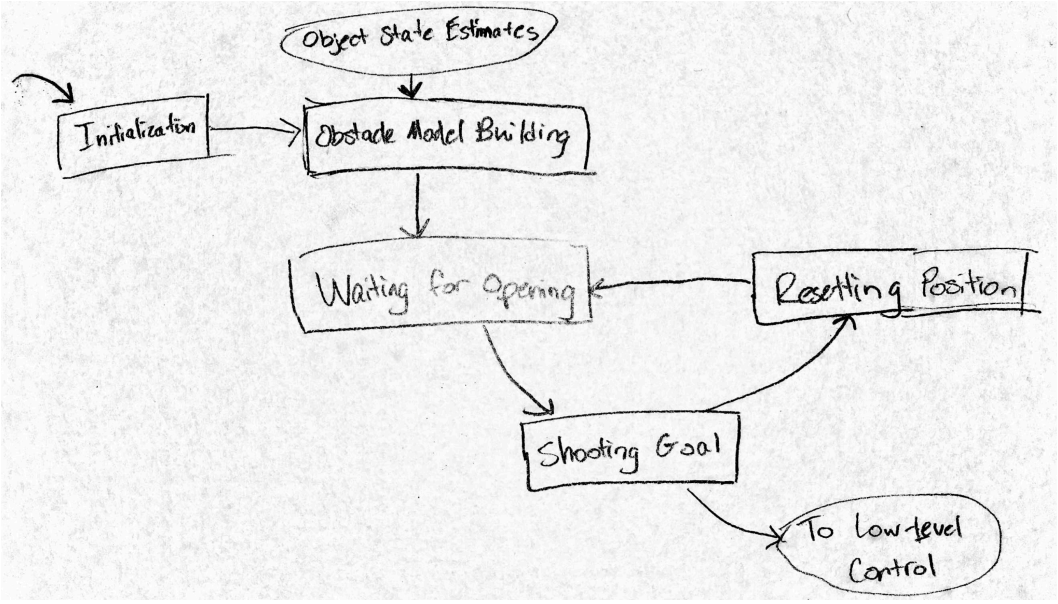
The next component is the object state estimator. It takes in the blob readings mentioned above and uses them to estimate the states of the objects in the environment, which in this case are the red ball and the 3 differently colored obstacles. It first reads in the pixel data and converts it to 3D coordinates in the local camera coordinate frame and then converts those camera coordinates into world coordinates. The output is a vector of floats representing the 3 dimensional position and velocity concatenated together, as well as an optional 6 dimensional covariance matrix. The object state estimator also runs constantly since it must continually take in the sensor readings and convert them. The state estimator is implemented using geometric calculations very similar to lab 5 and coordinate transforms. Potential failure cases would occur if incorrect data was sent from the sensor, such as a height/width and (x, y) combination that goes off the screen or is impossibly scaled.

After that is the high-level control loop, accomplished through the Loop and MyPlayerLogic methods that can be grouped together. The purpose of the high-level control loop is to take in the state estimates to decide what is happening in the environment, update the robot's internal state, and tell the low-level controller what to do accordingly. The inputs are the position and velocity readings in a 6-element vector of floats as mentioned above which were converted to the world reference frame. Because part of the control loop acts as a state machine, another input comes from the control loop itself, the previous state. This is encoded as a string describing which state the robot was at in the previous step. This is used along with the object state estimates in the logic to determine the next state. The output is the next state the robot should go to, which is encoded in the state variable, and also depending on the state, a configuration milestone sent to the robotController. The configuration milestone is a vector containing the desired configurations that the robot should go to. The high-level control loop is also run at each time step, though many time steps, it may not do much and just remain in the same state, for example when it is waiting for a movement to finish or waiting for an opening to shoot a goal. It is implemented using a series of case statements representing the different states that direct it to the next state. Additionally the obstacles in the environment are represented using numpy sinusoidal models stored in the controller created using curve-fitting on the object estimates given. These are used to predict the future locations of the obstacles to look for openings to shoot a goal. Potential failure cases could be if the low-level control malfunctions but the high-level control still believes it is on track, or wrongly interpreted obstacle states due to noisy data that cause the ball to hit an obstacle.

The end of the system is the low-level control, which takes directions to physically move the robot. In this case, the inputs are milestones representing desired configurations for the robot joints, and the output is the change in the physical orientation of the robot. Through the use of milestones, the low-level control does not need to be constantly invoked by the high-level control and can just be invoked on request. Much of the implementation is done behind the scenes by the

robotController class. Potential failure cases are improper milestones that violate joint boundaries or velocity constraints, or malfunctions in the robot machinery.

C



After being initialized, the planner first takes a few seconds to build a model of the moving obstacles in the environment by fitting the object state estimation data to a sinusoidal curve. After this is done the planner is then ready to operate by observing the movement of the obstacles and waiting for an opening to the goal. Building the models enabled the planner to look many hundreds of time steps into the future, as it takes a few seconds for the ball to travel from the starting position past the obstacles towards the goal. When an opening is found, the controller moves to the goal-shooting state and sends a series of milestones to the low-level control that result in a fast movement being done to hit the red ball in a line towards the goal. After this is accomplished, the robot must then transition to a reset state to move the robot back to its original configuration. This requires different movements so the robot does not hit the ball on its way back its original configuration. Once back, the controller returns to the waiting state where it stalls until another opening is found.

D

In this situation, the perception components did not need all components of 3D object positions and velocities in order to perform the task. Because a sinusoidal model was built based on the change in position of the obstacles in each time step, no velocity calculations were actually needed. As long as the planner had a few seconds to observe many data points consisting of position and time for each obstacle, it was able to accurately construct a sinusoidal model of the movement of each obstacle. Once this was done future positions could be calculated with ease. The beauty of the construction of the models was that accuracy was not very

important in the measurements. By using sinusoidal curve-fitting through the scipy module on the many data-points given as input to the model, an accurate model could be constructed from rather inaccurate measurements.

E

The system worked well to score goals in the time frame given. It was harder than expected to figure out how to build a mathematical model of the obstacles, but it greatly improved the system's capabilities. One improvement that could be made, however, is that the planner first spent a few seconds to build a model of the obstacles and then used that model from then on out, where instead you could try to have the model keep on updating over time, which would greatly increase the number of data points available and thus the accuracy. This approach would make the planner more robust to more noisy data. Some additional challenges you would face when designing a robot in real life would be this noisy data, as well as the possibility of system malfunctions. These malfunctions would require adding new states to the planner to handle possible failure.