ME - GY 9966

MS Project Report

The Swarm Robotic Game

Submitted by

Cuddalore Parthasarathy Sridhar

N16077628

scp392

To the department of

Mechatronics and Robotics Engineering

(Fall 2017)

Abstract

In a world where self driving cars already exist, it is time humans start to get comfortable living with robots. To get the interaction started many small robotic games have been devised that work in a way that humans control them with a remote or program them and watch the robot in actions. We take this interaction to the next step by letting the autonomous robots play with the humans.

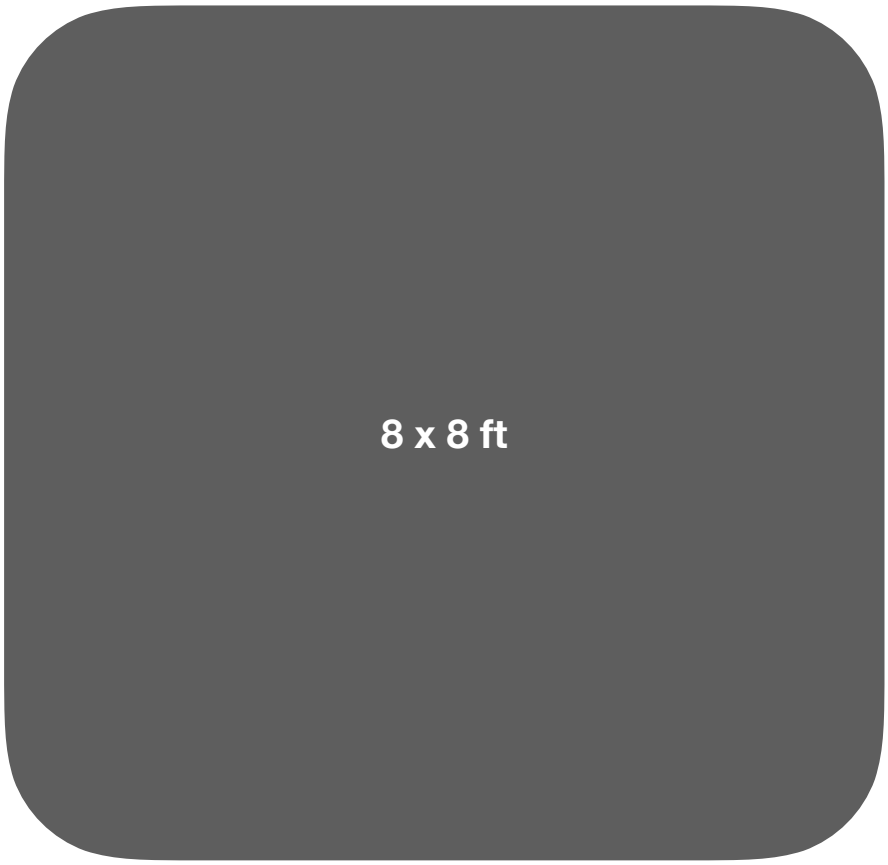
The robots we use in the game are all planar omnidirectional. The gameplay in simple worlds could be described as a catch and catch game. All the robots work together to catch the human player. The player's score is based on the time it took for the robots to catch the player. The swarm of robots communicate between themselves making dynamic formations that are hard to dodge by the player.

Acknowledgement

Thanks to Professor Vikram Kapila for providing the fixed space for this project in the Lab and also for procuring most of the components required for the project. I also would like to thank Dr Mizanoor Rahman for timely checks on progress and coordinating biweekly review. The knowledge on swarming and formation control is purely form a class with Dr. Ludovic Righetti. All his class lectures slides, papers presented, homework exercises contribute to over ninety percent of my references. I would like to thank him for taking the time to help me understand certain implementations of swarming better.

Table of Contents

1. The Arena



The arena is now planned to be a 8 ft square to accommodate around 8 to 12 robots and 1 human player. The sensing element is still not finalized. The possibilities are the following

1. One camera on the ceiling
2. Multiple cameras around the arena
3. Capacitance based sensing floor
4. Resistance based sensing floor

The Arena is supposed to be like a dark room immersive sound experience environment with lights only on the robots.

2. The Robot

Omni-Directional robot are typically robots with Mecanum wheels like the one in the picture below.

Considering the cost and time required to build these robots, an alternate choice to procure an educational robotic toy Sphere SPRK+ was chosen.

Spheros is by configuration a differential drive robot placed inside a sphere. The company provides ros packages and general api's that allow developers to connect with a sphero. The Sphere has 2 bright multicolored LED's, used Bluetooth as mode of communication and has a built in IMU based stabilizer.

3. Robot Operating System

The complexity of the entire system demands a heterogeneous computer cluster architecture such as ROS. Having analyzed the practical advantages and limitations on making this choice, ROS was decided not to be used for this project.

Reasons:
ROS is highly platform dependent. Each package of ROS is written very specific to a particular ubuntu release and also a specific ROS version. Combining multiple packages that are chronologically far apart in their release date will demand different release of OS or ROS which creates further complexities.

4. Python

Sphero provides general purpose API that can be used by developers to connect to spheros. I am using this api to connect with sphero. This provides us access to control l the LED's on the Sphere and the motor velocities.

5. Libraries

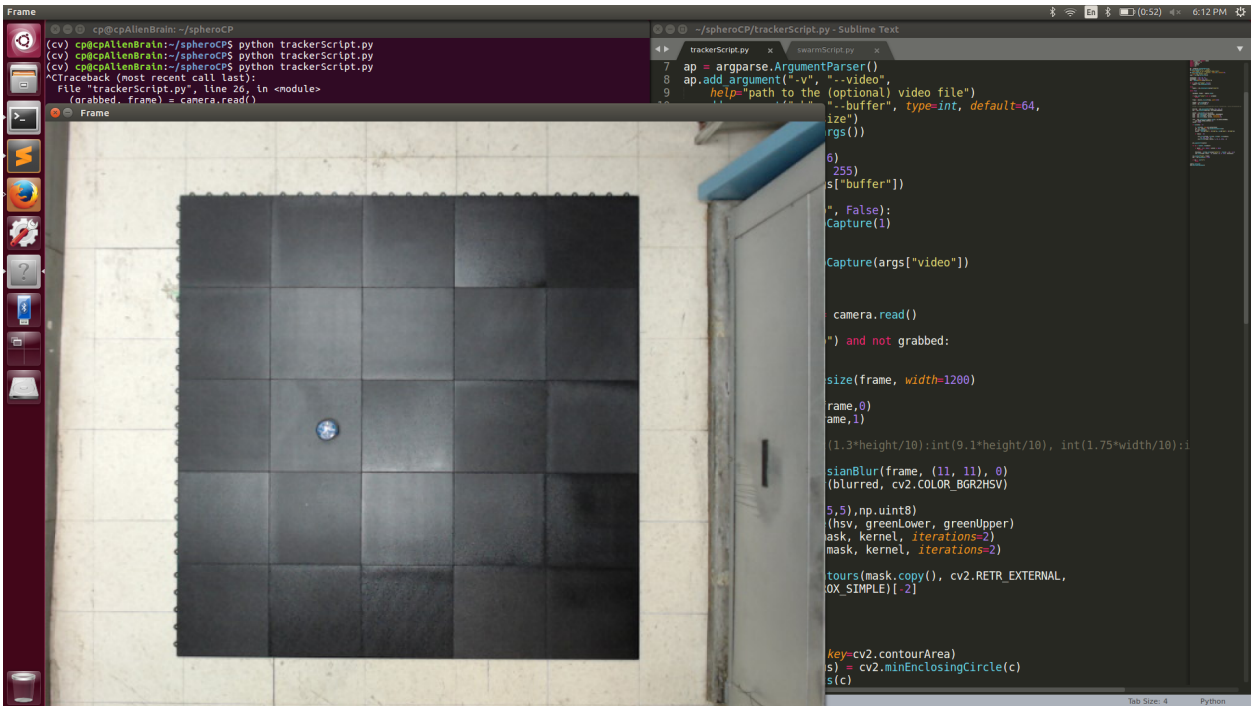
The sphere API does not handle connection to BLE devices on the hardware level. *BluePy* is a project that provides API that allows access to BLE devices from Linux machines. *BlueZ* is also another project that is the official Linux bluetooth protocol stack. All these three libraries/ API are used in combination to connect to multiple Sphere SPRK+ robots simultaneously from a linux computer.

6. Lab Experiment Setup

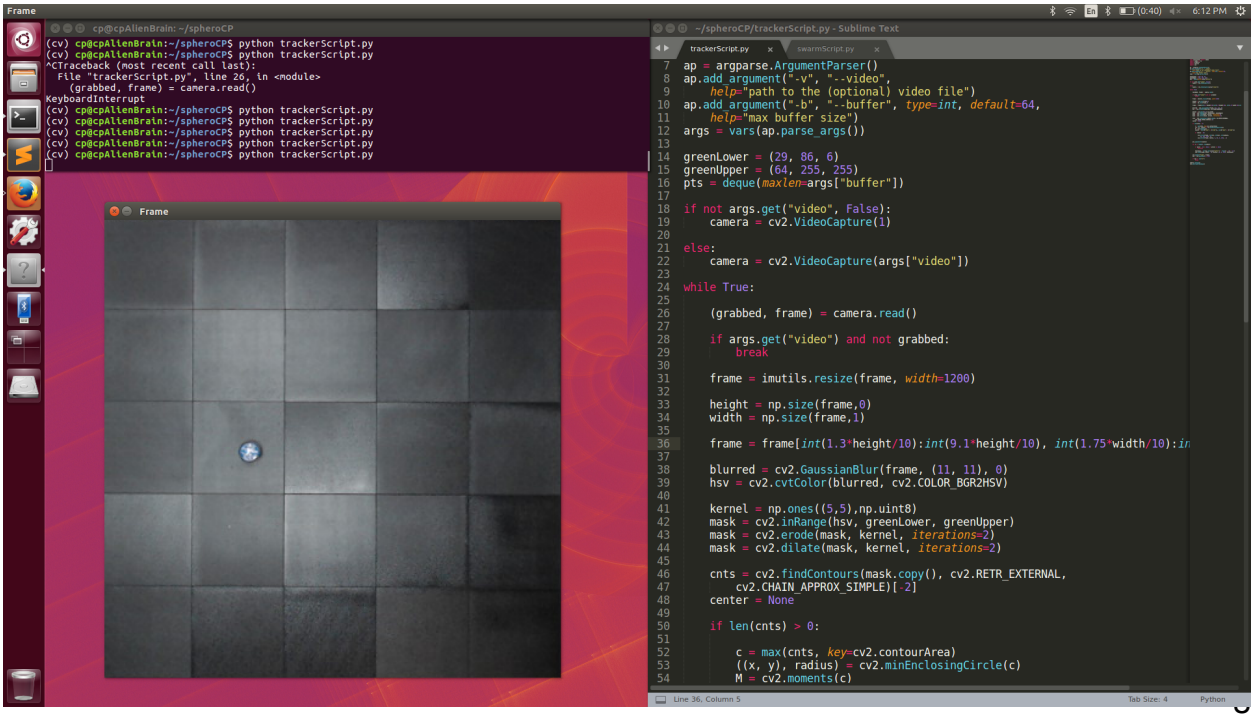
A Logitech camera is mounted on the ceiling, centered at the centre of the black floor plate to minimize parallax error.

7. Camera Feed

The above raw camera feed is cropped and mapped from pixels to 100 x 100 unit matrix to enable conventional XY terminologies.



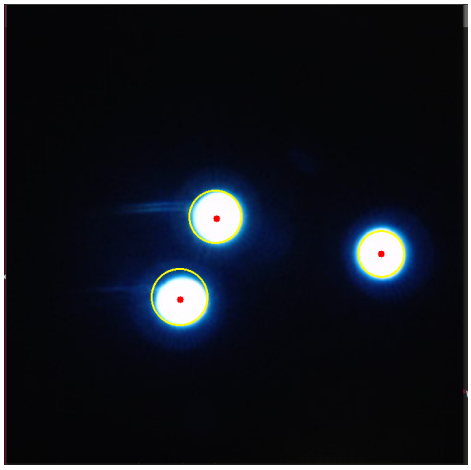
Bottom Left is [0,0]
Top Right is [100,100]

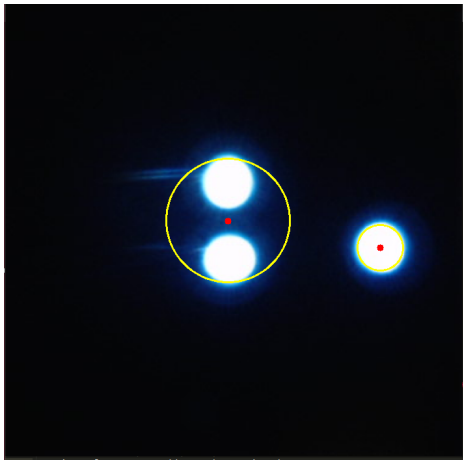


8. Sensing Robots

To start with a very simple single camera on the ceiling method is adopted to find the robots location. Since our robots do not have a unique marker on them tracking becomes a bit challenging. With the LED's on the robot turned on we do blob detection using OpenCV2 in Python on the live camera feed.

The kernel size have been carefully chosen after multiple rounds of experimentation to minimize the blob size. As the blob size increases the distance between the robots have to be increased.

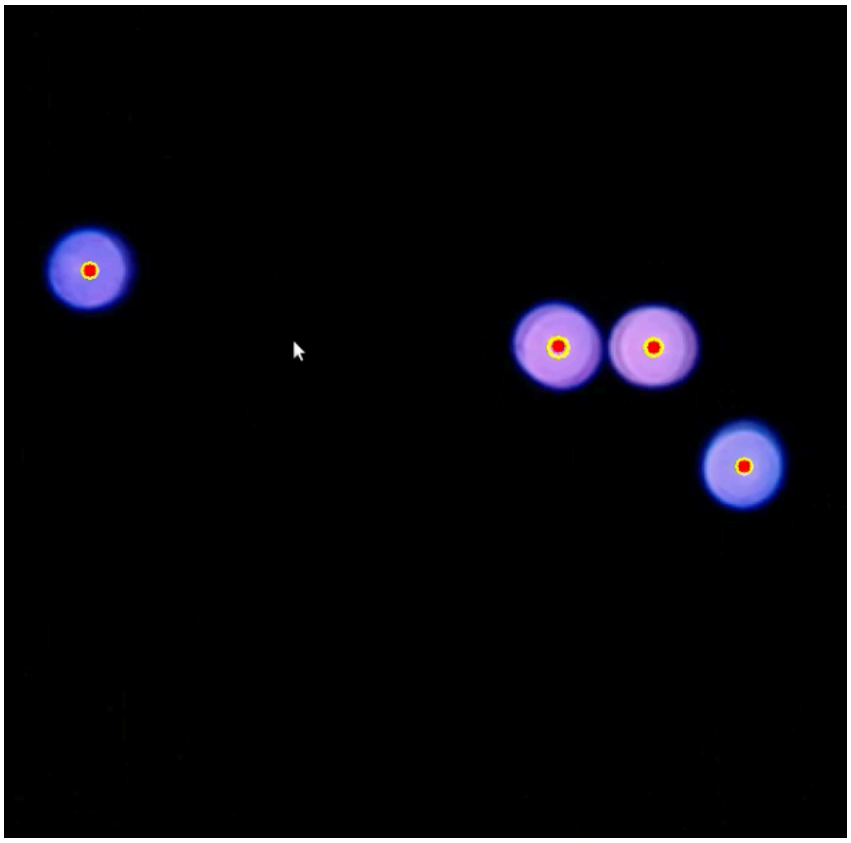




As the distance between two robots is less than 15cm, two individual blobs fuse to look like one huge blob. This limits the number of robots on the arena and the range of motions possible.

The software filters brought down the minimum Euclidian distance between two robots to be 15 cm. Another layer of hardware polarized filter was aded to the camera lens as an experiment to improve performance.

With this filter in place the robots can get as close as 1cm reopening the full range of motions possible for the robots.

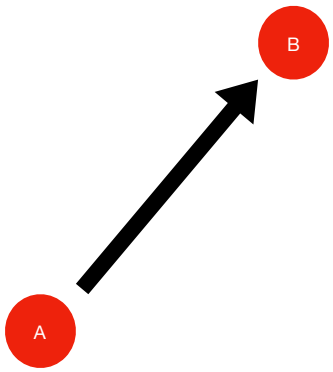


9. Robot Position

The centroid of each blob is considered its [x,y] position.

10. Robot Heading

Knowing the robot's heading is very important to control it. All the attempts made to access the IMU's raw data failed. Hence odometry based state estimation is not possible. The alternate approach taken to calculate the robot's heading is to just make the robot move and calculate its heading every frame based on the blob's location.



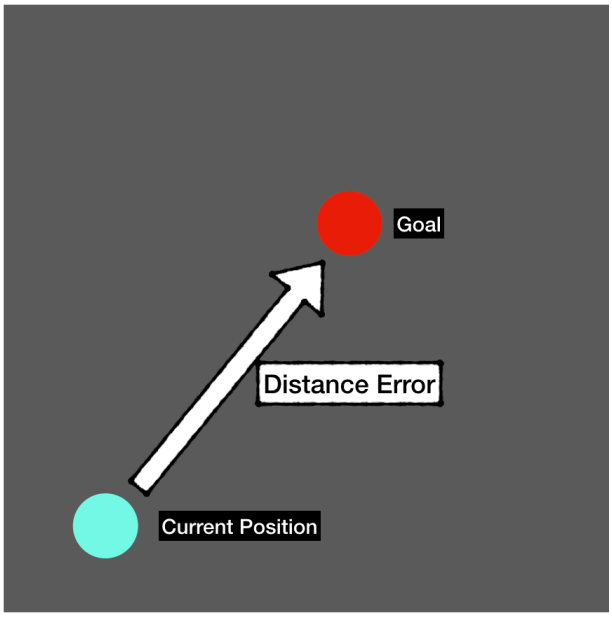
- If the blob moved from A[x,y] to B[x',y'] in time dt
- B-A [Δx,Δy] is a vector depicting the motion
- ata2(Δy/Δx) provides us with the heading of the robot

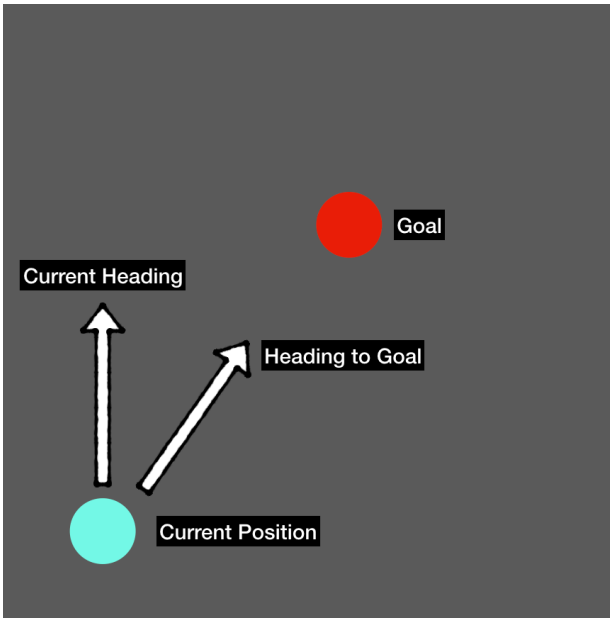
Assumptions and Limitations:

- We can find the heading of the robot only if it moves at least once
- We cannot update the heading if the robot does not move
- We will still be able to make stand and rotation to angle functions based on state prediction, provided the initial heading before the rotation is known

11. Moving to Goal

As the position and heading of the robot is known with respect to the global frame, we move the robot to any goal using a P controller.





The velocity ‘V’ of the robot is chosen proportional to the distance error and the angle error is used to calculate the angle steps ΔTheta.

Control Outputs:

Left Wheel Velocity = f(V - ΔTheta)
Right Wheel Velocity = f(V + ΔTheta)

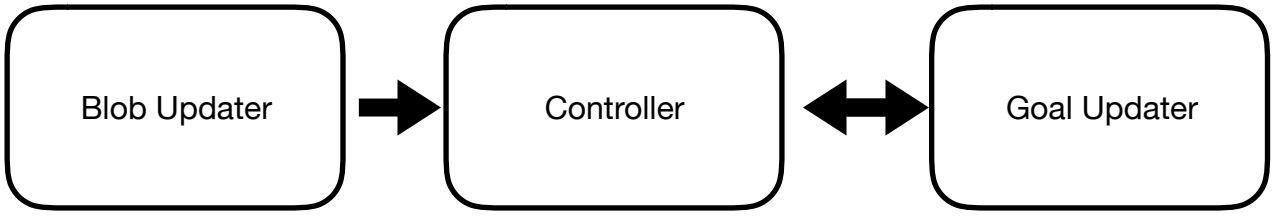
This will further be updates to a PD controller.

Now the robots move forward and eventually turn towards the goal.

In the future this will be changed to join to goal and start moving if the robot is initially at rest.

12. Data Structure

As having one long while loop that keeps iterating the states of the robot one after the other would be too computationally inefficient, the tasks have been split into three scripts running individually.



The *Blob Updater* script continuously detects blobs in the video feed and stores the position of each blob in global frame in to binary memory location

The *Controller* script connects to the robots, reads output from the *Blob Updater* and carried out multiple tasks.

1. Decides the [Robot IP, Blob] Location pair based on continues state monitoring
2. Decides the [Robot IP, Goal] pair base on the least cost for the entire system
3. Provides wheel velocities for each robot to go to its corresponding goal

The *Goal Updater* script reads the [Robot IP, position and heading] pair from the *Controller* and provides the *Controller* new set of goals that are estimated by the *Formation Controller* programmed in this script.

Communication between scripts now use pickle. The communication module will be updated to use Google Protobufs as it is has a reduced computational cost as opposed to using pickle.

13. Multi-Threading and OOP

As all the robots needs to move together and we have a central computer that connects to all the robots, the *Controller* script cannot address robots one after the other. State tracking, goal allocation and determining the wheel velocities for each robot has to happen in parallel. This demands multi-threading.

Each robot is made an object of the class *bots* and each robot can access only the data of its neighbors making the system decentralized.

Each robots runs an individual thread inside the *Controller* script for each of its tasks. For one complete iteration we have n threads where n = no. of robots x no. of tasks

14. Initialization

The blobs location are known to the main controller but the controller has no clue as to which blob is which robot. Hence we do sequential initialization of all the robots one after the other. All the robots are placed in a random location on the floor plate with all LED's turned off. Now we turn on One robots LED's and map the new blob that appeared to this robots IP.

Limitations:

- No robots can move until all the robots are initialized
- All the robots have to be initialized at the start itself

This is a very crude of way of getting things done. The game play demands dynamically adding and removing robots into the arena. So I am planning to restructure the initialization module completely.

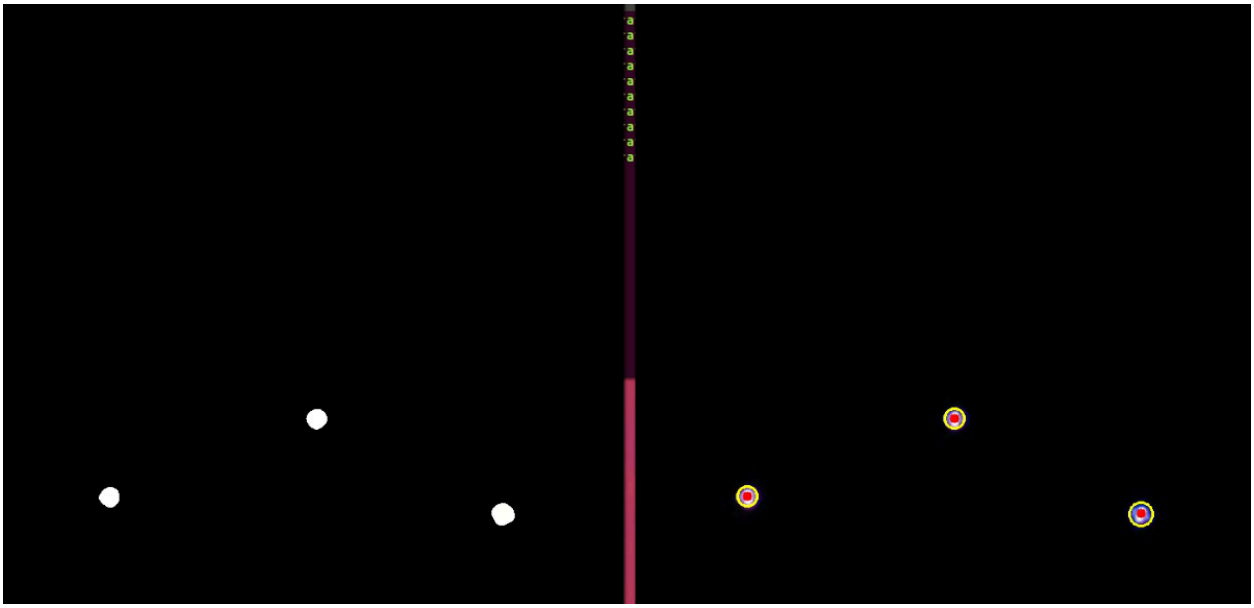
15. Tracking

As the robot moves based on the control output, the feedback we get is the new location of blobs. We iterate these blob location with each of the robot object's state-position to measure the Euclidian distance. The robot_xy to new_blob_xy distance will be minimum for the true pairs as the robot cannot jump or hit each other. The new_blob_xy position is updated as the robot objects current position.

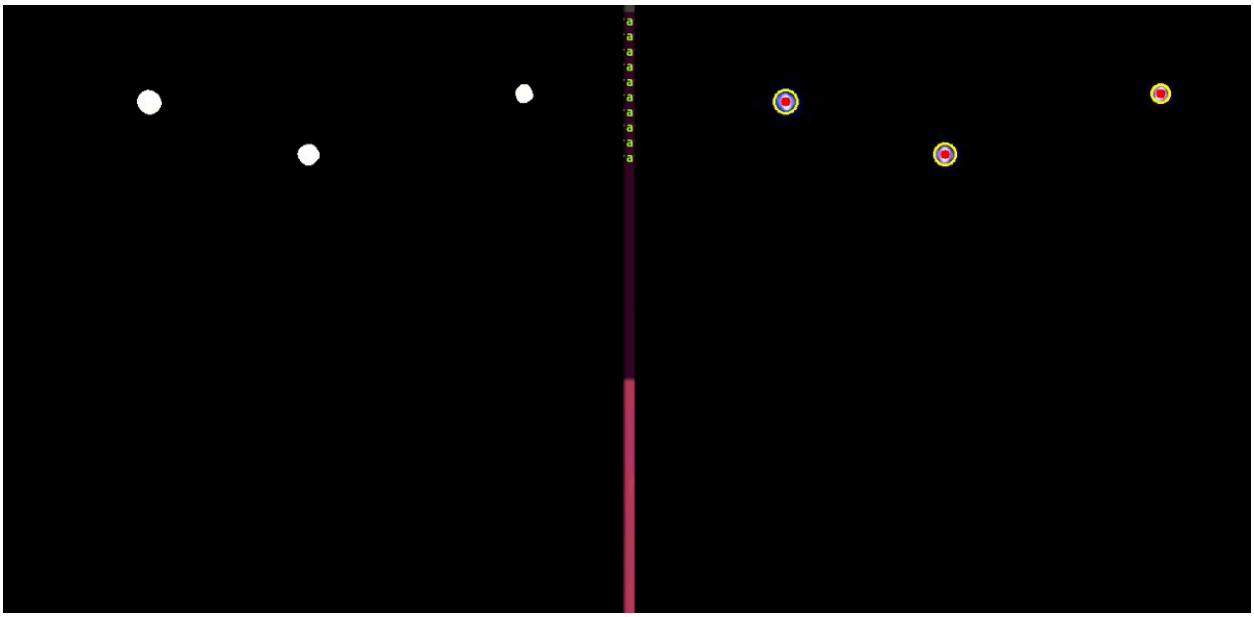
16. Swarming

16.1 Adaptive Swarming

The robots move to take a formation. If the formation is disturbed the robot moves back to the formation dynamically reallocating positions based on the least cost for the system.



They can switch formations. Here I show them switching from a upward arrow formation to a downward facing formation



16.2 Formation Control

In formation control robots do not switch places. A predefined network is built for each formation and at the start each robot moves to take its position to maintain edge controls provided by the graph network.

The graph network we have is always a routed out branching and lets say ‘robot A’ is the root. Now if any robot other than robot A is moved that robot will come back to its position to hold formation. Instead if robot A is moved, all the other robots move corresponding to robot A to hold the formation.

Right now Adaptive Swarming is implemented on real robots with a few limitation in the range of operation as robots cannot collide.

Formation Control is now being implemented in a simulation with six differential drive robots. After completing a series of experiments in the simulation and understanding the behavior of the swarm, I will soon implement the same on sphero robots.

17. Plans for the next semester

1. Update P to PD controllers
2. Change adaptive swarming to true formation control
3. Track humans and robots simultaneously without occluding the tracking of any robot
4. Restructure the entire project to reduce computation cost
5. Convert static initializer function to a dynamic initializer
6. Find a trade of between being centralize and decentralized in terms if data handling with higher priority to minimizing computational cost.