CSE396 - SPRING 2023

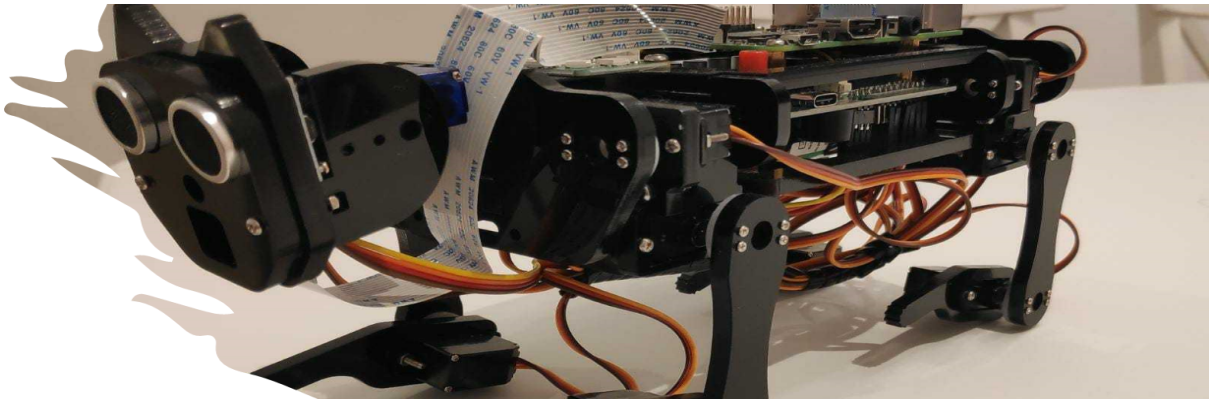
Group 2 - Rin Tin Tin

Documentation

Team Members

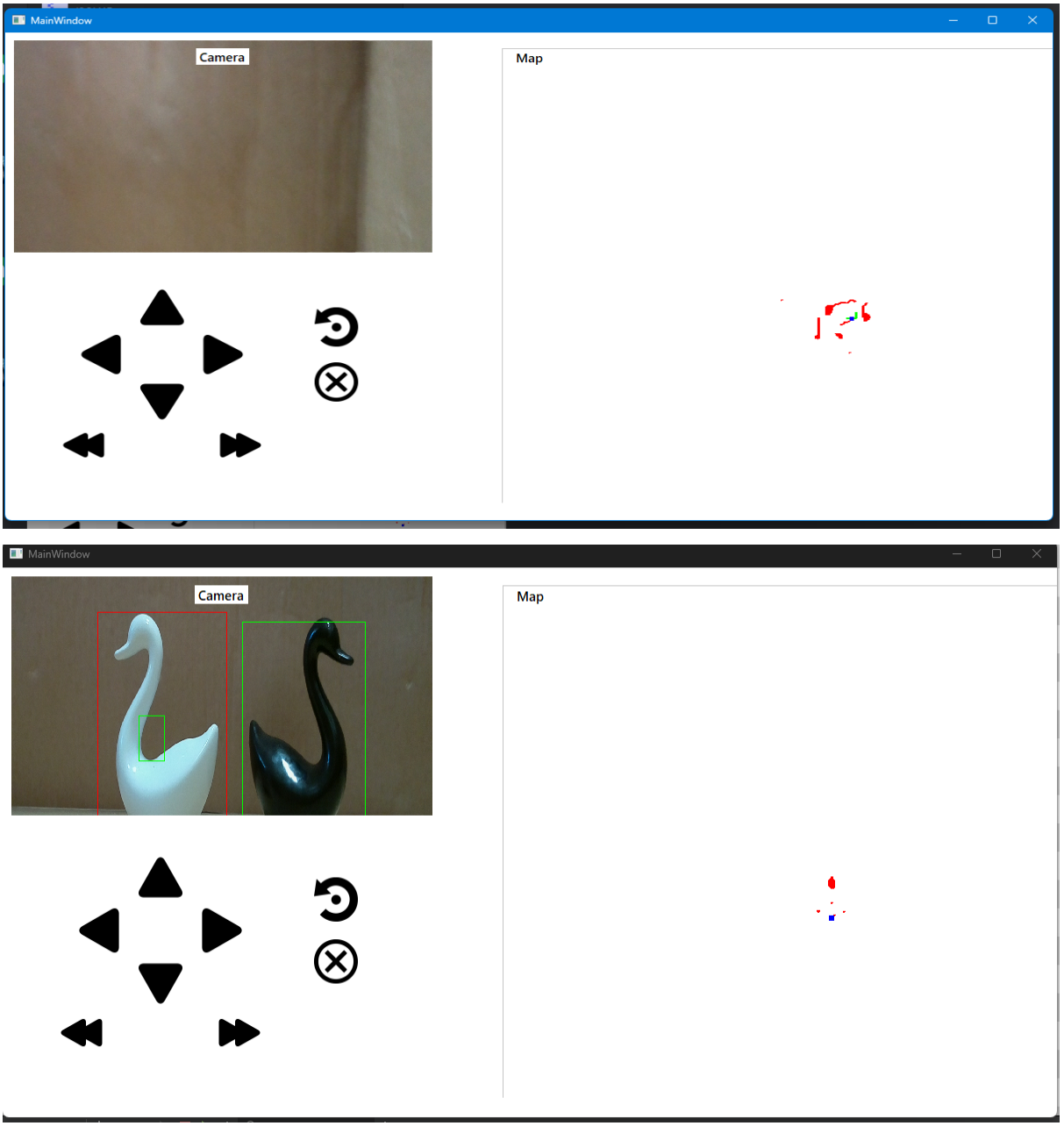
BAHADIR ETKA KILINÇ	1901042701
BURAK KOCAUSTA	1901042605
ÇAĞRI ÇAYCI	1901042629
EMRE SEZER	1901042640
ERAY YAŞAR	1901042635
HASAN DELİKTAŞ	1901042625
MUHAMMED AKİF SEVİLGEN	1901042608
MUHAMMED SEFA CAHYİR	1801042686
OZAN ARMAĞAN	200104004014
OZAN ÇOLAK	1901042645
SÜLEYMAN BURAK YAŞAR	1901042662
UĞUR ER	1901042262
YAVUZ SELİM İKİZLER	1901042617

Rin Tin Tin



Website: https://rintintingtu.com.tr/#main

1- Application



We developed a GUI application with Qt. C++ is used during development. Application is ported to Windows, and POSIX Operating Systems. It is applicable on all POSIX compliant operating systems, and Windows. We only needed to change POSIX calls to win32 system calls in order to do that.

Application has 4 main functionalities:

Sending commands to Raspberry Pi:

Buttons on the left bottom of the window are for directing the robot.
In order to give directions to robot, we use TCPClient and send data to the
TCPServer running on the Raspberry pi on the same network. Forward is for moving the robot forward, back is for moving the robot back. Left is for rotating the robot to 90 degrees left and Right is for rotating the robot to 90 degrees right. Other Left and Right commands are for stepping, robot steps left without rotating.

Close button cancels the autonomous movement on the path.

Reset button resets the all map and autonomous movement.

Receiving Video From Robot's Camera:

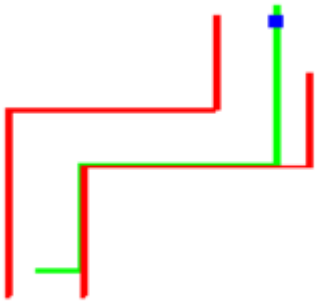
There is a frame at the top left to display video camera footage. App uses an Object Detection Module in order to detect the specific object we determined.
If the object is viewable on the robot's camera, it draws a rectangle around it on the image and displays it in the app.
With that approach the user can understand if the object is detected by the app or not. We use Opencv with C++ in order to achieve this.
We encode the frame at the server and send it through the socket to the app. App receives that data and encodes it to Opencv Mat called frame. Then, it displays that frame on the screen.
Details for Object Detection will be provided at the Object Detection part of this report.

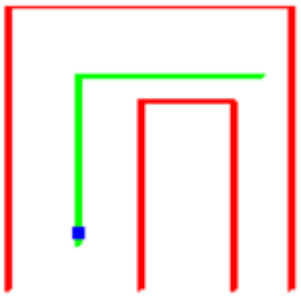
Drawing Map:

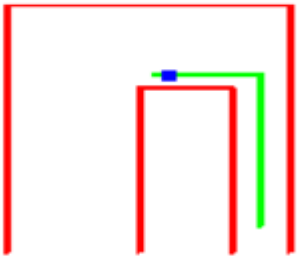
You can view the mapping provided by the Raspberry Pi at the right on the screen. Red dots represent the obstacles detected by the robot. Blue dot represents the current position of the robot. Green dots are for the shortest path to reach target position. This map is updated with the movement of the robot. You can track the 2D environment of the robot from the app. It represents a greater map, but it is scaled to a smaller map in order to fit it to the screen, but all operations are done in an unscaled map. Scaled part affects only visualization.
Details for Mapping will be provided at the Mapping part of this report.

Drawing Path:

When a pixel of the map is clicked, the robot moves to the located place on the map, and it maps the place it traversed. This movement is determined with a path finding algorithm which is explained in detail on the autonomous part of this report. Green path is drawn when a pixel is clicked and the robot moves on that path



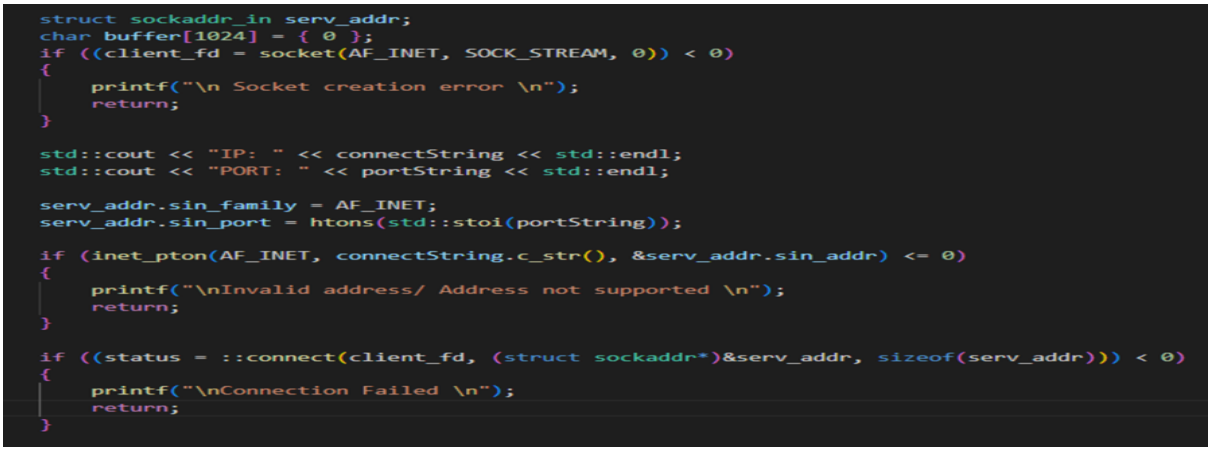




2- Communication and Synchronization

There are 3 threads on the raspberry side, those are for mapping, movement, and sending images. Image thread is working independently from the other two threads. Mapping and movement must be synchronized, because it shouldn't map and move simultaneously for proper mapping. It should also map continuously if there is no movement. We used conditional variables, and mutex for the synchronization of these two threads.

In the first demo, we didn't need the synchronization of these two threads because autonomous movement was not properly implemented. When autonomous movement is implemented, commands are given continuously, this causes the faulty mapping. We solved this with the synchronization of these two threads.



```
struct sockaddr_in serv_addr;
char buffer[1024] = { 0 };
if ((client_fd = socket(AF_INET, SOCK_STREAM, 0)) < 0)
{
    printf("\n Socket creation error \n");
    return;
}

std::cout << "IP: " << connectString << std::endl;
std::cout << "PORT: " << portString << std::endl;

serv_addr.sin_family = AF_INET;
serv_addr.sin_port = htons(std::stoi(portString));

if (inet_pton(AF_INET, connectString.c_str(), &serv_addr.sin_addr) <= 0)
{
    printf("\nInvalid address/ Address not supported \n");
    return;
}

if ((status = ::connect(client_fd, (struct sockaddr*)&serv_addr, sizeof(serv_addr))) < 0)
{
    printf("\nConnection Failed \n");
    return;
}
```

We provided communication through stream sockets. It is mostly needed between different devices. Different threads communicate through shared data structures when they are on the same device. The application is ported on Windows and POSIX, so some system calls are changed in order to work it on Windows.

Mapping process on the Raspberry needed to send the distance result of each coordinate to the main machine. Message format is a simple string which consists of points for each index. It represents the mini-map of a robot scanned. Server part converts it to a string, and the client side parses the message.

Command process on the Raspberry needs to get the respected command from the main device. It is a string which represents the command it gets, correct formats are “LEFT”, “RIGHT”, “FORWARD”, “BACK”. Again the raspberry process is the server, application is the client. After raspberry gets the command, there is a message for the purpose of giving feedback. Raspberry gets the command, performs the movement, and returns the “RECEIVED” message through socket. If this message is received from the application, the map is updated.

Video process on the raspberry, records the video and continuously sends it to the application module. Message format is a serialized video frame, it is read and written with that format to the stream socket.

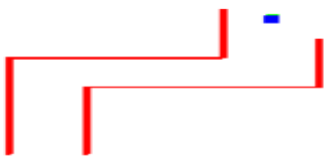
3- Mapping

We needed a scanner for mapping, then this scanner was requested from the hardware module. Hardware module's solution is putting an ultrasonic sensor with a servo to the head of the robot to scan a half circle with radius 150 cm which is the maximum distance an ultrasonic sensor can measure in front of it.Ultrasonic sensor measures the distance of the objects in 5 angles. These angles are 0, 75, 90, 105, and 180. For each angle, ultrasonic makes 20 measurements. In this way, we eliminate the outliers which are called noises in our dataset. After that, we get the mean of the sum of our dataset and get the most proper result.

Mapping thread on the raspberry, manages the servos and sends the coordinates of the obstacles which are detected by ultrasonic sensor scanning results. These points are paired as (x,y) and sent to the application. We managed the calculations for mapping with creating another thread on the application side. When the application gets these points, it converts those points into its internal data structure. This is what we call a "mini-map". When each mini-map is gotten from the raspberry, it must be put into the map according to its position, and the angle with starting angle which is 0.

Every time the robot moves or rotates, we will update the angle and position of the robot on the big map. Also mini-map position and angle is set according to the robot. In this way we create a general map of parkour.

We use an algorithm in order to beautify our map. This algorithm helps to combine points continuously which leads to creating a better map, and makes the path finding algorithm's job easier. Also we did not remove any removing operation in the big map to solidify obstacles positions.

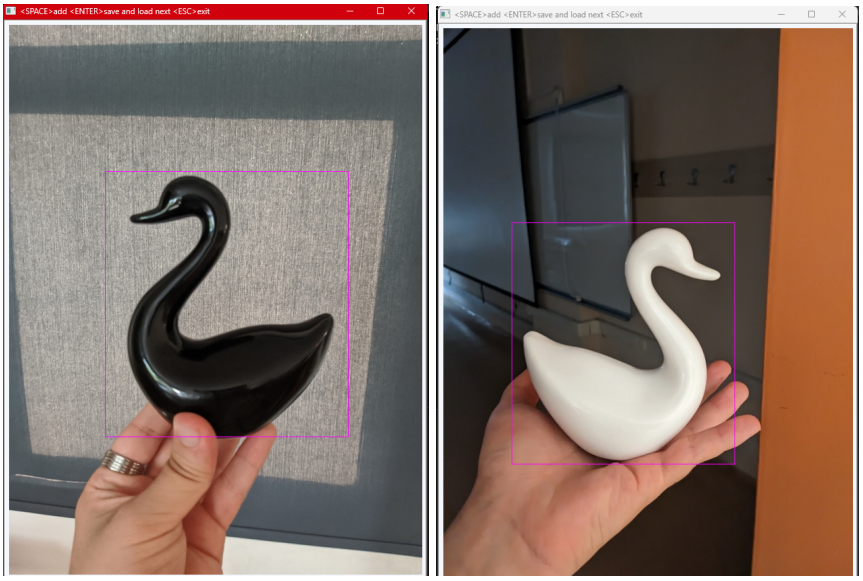


3- Image Processing

We have implemented the image processing algorithm using the Haar cascade classifier in C++ language. We have trained our classifier with custom data that we have prepared. The training data consists of 1000+ negative images and 100 positive images and also we added basic shape detecting algorithm to detect objects such as triangle,square and rectangle by detecting contours as the utility of opencv library.

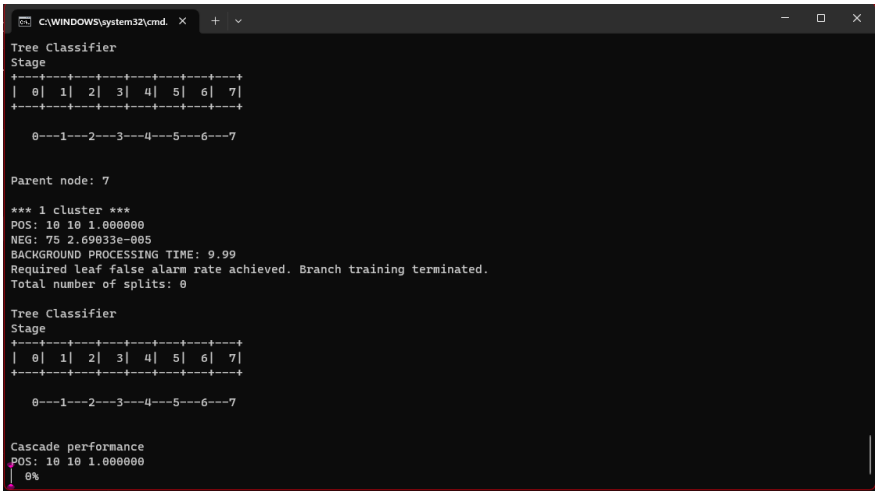
Training Process

Our positive image data set is for detecting a black swan and a white swan. All of these photos are taken from the Raspberry Pi camera for training the classifier depending on real usage situations. For training we needed to label every instance one by one to get exact result for object detection.



Our negative image data set of the parkour to train not detecting irrelevant objects.

We have used these positive and negative images for training our classifier. We have used a graphical interface that takes these positive and negative images, and trains a cascade classifier model.



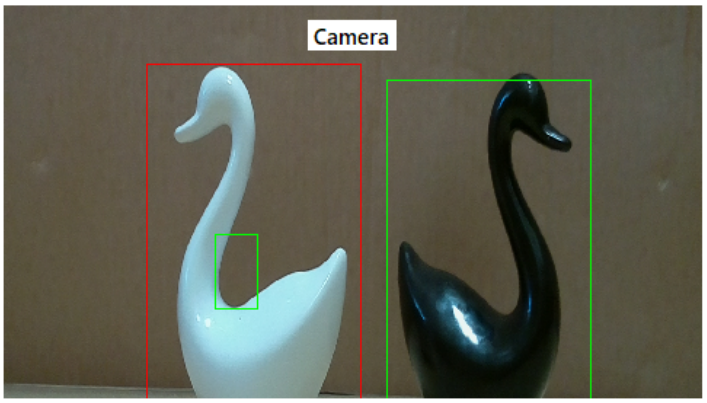
Result of our cascade classifier training is an XML file that is specially designed for our custom model.

Using the Custom Trained Classifier Model

After the training of the classifier model, we have created an XML file of the model. We have loaded this file to create a CascadeClassifier object in C++. After the loading of the classifier and receiving the video to our app, we used the detectMultiScale function of the CascadeClassifier for our classifier and created a vector of rectangles that marks the objects that have been found.

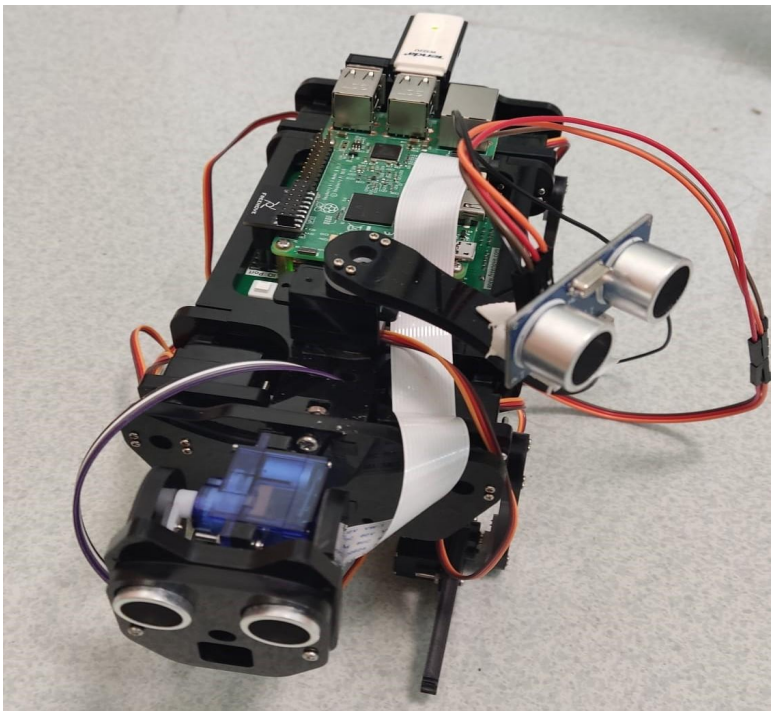
```
vector<Rect> white_swan, black_swan;
faceCascade.detectMultiScale(gray, white_swan, 1.1, 3, 0, cv::Size(30,30));
faceCascade2.detectMultiScale(gray, black_swan, 1.1, 3, 0, cv::Size(30,30));
```

After all of these process, we can find our object and mark it successfully

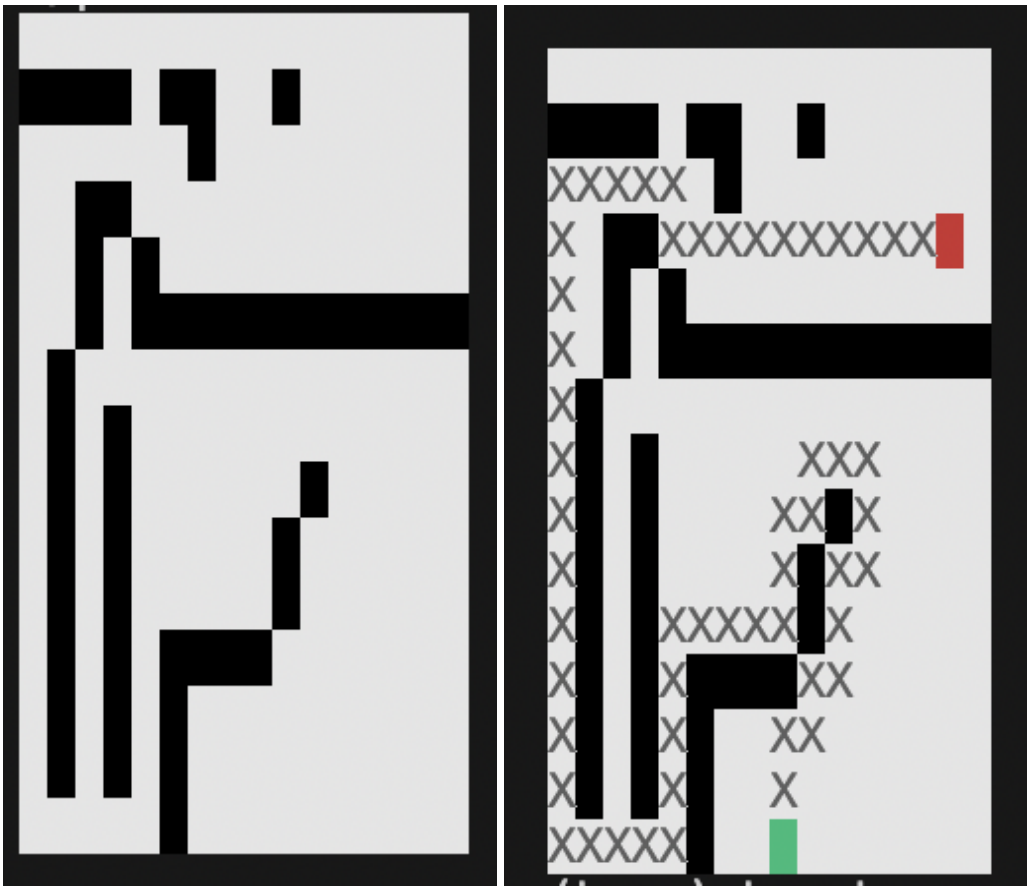


5- Hardware

In this phase of the project, we preserve most of the parts in the robot generally. Building upon its foundation, we integrated a new servo and a new ultrasonic sensor to the hardware with the request of the mapping module. The newly integrated servo has been securely attached to the robot's body, enabling it to execute swift and seamless rotations spanning a full 180 degrees. The newly integrated ultrasonic sensor is attached to this new servo with some components. In this way, ultrasonic sensor can scan obstacles in a half circle area with a diameter of 150cm.



6- Autonomous Movement



Problem Definition:

1. Our autonomous robot is designed to navigate towards a specific target on a map. The map is represented by a network of nodes, where each node represents a specific location. The robot explores the nodes on the map using the A* algorithm, aiming to find the shortest path.

The A* Algorithm:

2. The A* algorithm is employed to discover the shortest path by traversing the nodes on the map. The algorithm prioritizes the nodes based on their distance and estimated distance to the target. As a result, the robot advances towards the node that is closest to the target. We are not using A* algorithm directly, because this algorithm is not designed with directions. It must be implemented with respect to directions, and in order to make proper mapping, It must rotate, without stepping. If it rotates, and continues forward, the map will be better. Because of this we updated the A* algorithm with the knowledge of robots degree. In normal A* algorithm, it checks directions in the same order all the time. But to achieve our purpose, we changed the algorithm by checking the forward of the robot first according to its direction. When the algorithm produces a left or right step, the robot rotates according to its degree and performs the proper command. If it needs to go back, It rotates 180 degrees. Also this algorithm works well on simulation, but when the modules are integrated, it is changed a lot. Because 1 pixel on the map is not enough to rotate the robot. It rotates early for proper rotation.

Pathfinding:

3. The robot identifies the target node based on the user's input and utilizes the A* algorithm to determine the shortest path. The algorithm explores the nodes from the starting node to the target node and calculates the shortest path. This process enables the robot to efficiently identify the most optimal route by reducing the distance to the target.

Recalculation Based on New Data:

4. After each click or encountering an obstacle, the robot recalculates its path based on new data. This data may include the status of obstacles or unobstructed areas on the map. Using the A* algorithm, the robot determines a new path based on the updated data and continues its movement accordingly.

Robot Movements:

5. The A* algorithm determines the robot's next movement at each step. The robot executes the designated movement and then recalculates its path based on the updated data. This process enables the robot to progress towards the target efficiently. Algorithm movement patterns did not work well when modules are integrated, in order to solve that we add additional algorithms to processes that produce commands which are also designed with the knowledge of mapping, and rotation. After making those changes, integration is completed.

Conclusion:

Our autonomous robot aims to employs the A* algorithm to identify the shortest path towards the target on the map. After each click or encountering an obstacle, the robot recalculates its path based on new data, allowing it to navigate autonomously and reach the target in the shortest possible time.