MAY 12, 2021

Miriam Hebbassi 40056283

Cyril Girgis 40028766

Faisal Bari 27517440

Huy Hoang 40027954

ROBOT SIMULATOR FINAL PROJECT

MECH 472

TABLE OF CONTENTS

ROBOT CONTROL FUNCTIONS

COMMON CONTROL FUNCTION FOR ATTACK/EVADE

This section briefly describes the control functions in common between the evading and attacking robot. In both the evading robot and the attacking robot, the same functions to actuate the motors and detect borders, opponents and our own robot are used. These functions include but are not excluded to :

- robotForward

- robotReverse

- turn_ccw

-turn CW

-centroid equations (there are multiple centroid functions however, they are very easy to identify

-obstacle detection/radius calculation

ATTACKING ROBOT

The robot's logic is quite simple and dictated by the function aim_opponent. This function calculates the difference between the robot's angle and the opponent angle, if the difference between the angles is larger than the tolerance than it will turn clockwise till that is no longer the case. Once that is no longer the case, the robot moves forward and once it is at an appropriate distance, shoots it's laser.

There are two additional features that help the robot. One is the robot's border avoiding which forces it to distance itself from the borders. The other is a set of functions that make the robot circle around an obstacle once it is near it. This will prevent evading robots from hiding behind the obstacle. A video will be present in the .zip to display this.

FRIENDLY ROBOT

The friendly robot's movement controls are identical to the enemy robot's where the inputs change for the right and left wheel only for the initial set_inputs function. In fact, the variable names also change to pw_r_o and pw_l_o for the opponentforward and opponentreverse functions for the right and left wheels, respectively.

As for the initial inputs, the friendly or opponent pulse width inputs are initially manually set to pw_l_o as 1700 and pw_r_o is set to 1500 resuling in a counter clockwise circular motion. The max_speed is set to

100 pixels per second. These variables, additionally to the laser inputs (pw_laser_o and laser_o) are called in the set_opponent_inputs function as an initialisation to test the simulation.

The pulse width of the laser (pw_laser_o) for the friendly robot as well as its state (laser_o) is initially set to 1500 for the laser servo and 0 for its state meaning it is OFF.

BORDER CONTROL

The movement of both robots is also controlled to restrict the movement to the window size using the check_borders() function. The logic of this function is identical for both robots depending on the color of the robot centroids. In fact, for the enemy robot, we use image a using the green and red centroids functions where the centroid in the x and y directions are used as conditions in a if loop (ic and jc). The conditions for the loop are identical for both centroids, where the width of the robot and window height and size are the limiting conditions: if (ic_red < 30 || jc_red < 30 || ic_red>610 || jc_red>450). The width of the robot is found to be 30 pixels so assuming the robot is situated at the lower border of the window, if the centroid of the red circle is less than 30, then the robot moves forward. The same logic is applied for the upper border where the condition is the width of the robot (30) subtracted than the width and height of the window, 640 and 480 respectively. The same conditions are applied for the green circle as well as the pink and blue centroid for the friendly robot.

As for the movement once the border is identified, the movement of the robot when the given centroid is near the border is defined considering what centroid is nearest to the border as well as the orientation of the robot. The movement control functions described in the section above, namely robotforward and robotreverse are used to change the robot's trajectory depending on the side of the robot nearest to the border.

In fact, when the enemy robot is near the border from the green centroid position, then the robot moves in reverse and forward if the border is nearest to the red centroid. The same function is used for the friendly robot where the pink centroid being nearest to the border causes it to move in reverse and the blue centroid near the border causes a forward motion.

For the friendly robot, the same logic is used using the check_opponent_borders using the pink and blue centroids and image b.

The check_borders function can be seen in the figure below:

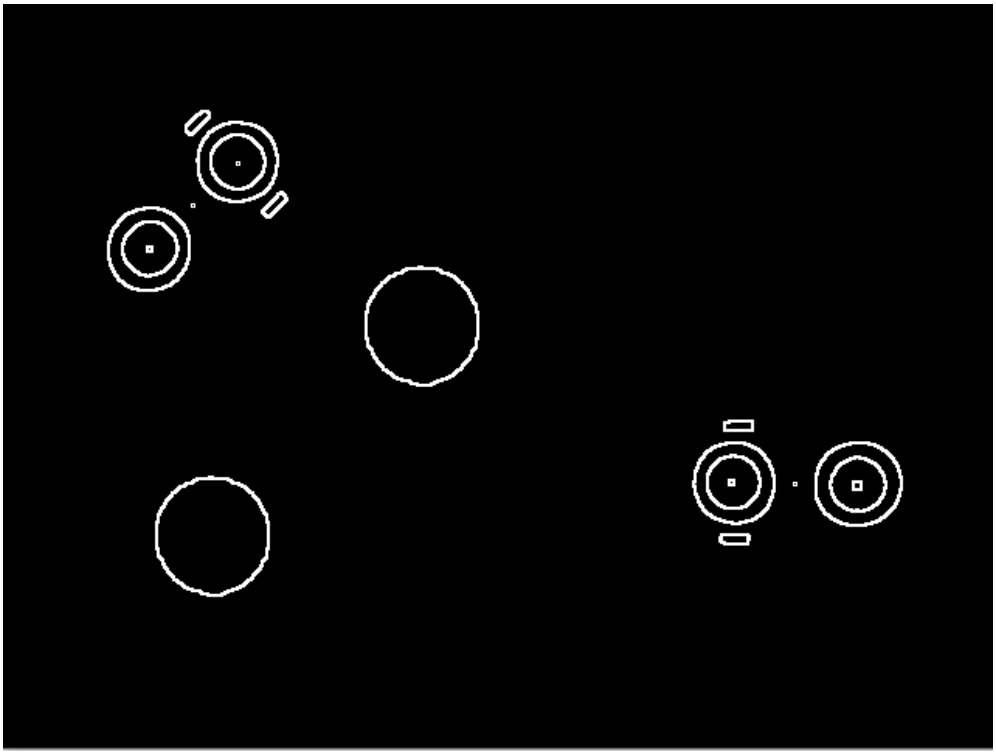
```
void check_borders(image rgb,int &pw_r,int &pw_l) {
    image a;
    double mi_red, mi_green, mj_red, mj_green, ic_red, ic_green, jc_red, jc_green;
    red_centroid(rgb, mi_red, mj_red, ic_red, jc_red);
    green_centroid(rgb, mi_green, mj_green, ic_green, jc_green);
    //draw_point_rgb(a, ic_red, jc_red, 0, 0, 0);
    if (ic_red < 30 || jc_red < 30 || ic_red>610 || jc_red>450) {
        robotforward(pw_r, pw_l);

    }

    if (ic_green < 30 || jc_green < 30 || ic_green>610 || jc_green>450) {
        //cout << "ic_green" << ic_green << endl;
        //cout << "jc_green" << jc_green << endl;

        robotreverse(pw_r, pw_l);

    }
```

COMPUTER VISION/IMAGE PROCESSING

Here, we talk about the various image processing techniques used in order for the robot to "see" the enemy player.

EDGE DETECTION

To be able to identify the objects in an image based not on color but on the shapes, a function called Sobels edge detection is needed to be created. The Sobel edge detection function works with grey images. It denotes an edge when there is a change in color intensity and marks it white. Everything else that is not an edge will be black, or a value of 0. However, one thing to note, when using edge detection filters, a lot of noise can be generated all throughout the image. As such pre-processing is required to soften or completely remove the noise. We start with the low pass filter to remove some noise and then use threshold and invert to change it to a black and white image. We then use the invert function and erode to make the image clearer. Afterwards, the Sobel edge detection function is used to get the image below.



The idea was to use the label_image function and use the centroid function at two separate times or two separate frames. If the centroid has moved, then we can assume it is a robot/car. If not, it is an obstacle. Knowing the centroids of the obstacle, we can then use Sobel to find the edges, by incrementing pixel by pixel in the I-axis from the centroid of each obstacle. This distance from the centroid to the edge is the radius of the obstacles. We can do the same thing with the robots, and simplify their hitboxes to a circle. Knowing the radiuses and the distance between each obstacle and robot, we can detect whether or not there was a collision.

```
if (firstpass == 0) {
    label_image(a, label, nlabels);
    for (int i = 1; i <= nlabels; i++) {
        centroid(a, label, i, ic[i], jc[i]);
        //draw_point_rgb(rgb, ic[i], jc[i], 255, 255, 255);
        view_rgb_image(rgb);
        //pause();

    }
    firstpass++;
}
else if (firstpass == 1) {
    cout << "first pass" << endl;
    for (int j = 1; j < nlabels; j++) {
        //cout << j << endl;
        centroid(a, label, j, ictemp[j], jctemp[j]);
        //draw_point_rgb(rgb, ictemp[j], jctemp[j], 255, 255, 255);
        //view_rgb_image(rgb);
        //pause();
        for (int i = 1; i < nlabels; i++) {
            if ((ictemp[j] > (ic[i] - tol)) && (ictemp[j] < (ic[i] + tol)) && (jctemp[j] < (jc[i] + tol)) && (jctemp[j] > (ic[i] - tol)))
            {
                for (int k = 0; k < size; k++)

                OBS_IC[OBS_NUM] = ictemp[j];
                OBS_JC[OBS_NUM] = jctemp[j];
                OBS_NUM++;
            }
        }
    }
    firstpass++;
}
```

The code above shows how the obstacle is identified. And the code below shows the equation needed for collision detection. If the distance between the centroid of obstacle and the centroid of robot is less than or equal to the sum of the radius of robot and obstacle, then we say there is a collision.

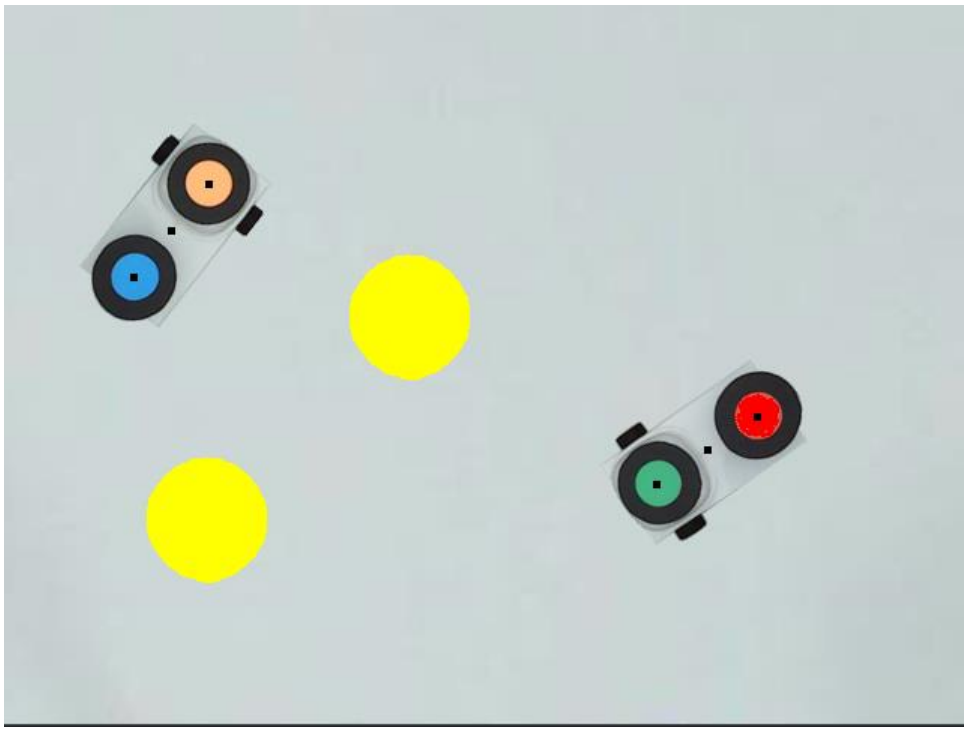
```
for (int i = 0; i < OBS_NUM; i++) {
    if (fabs((ic_car - OBS_IC[i]) * (ic_car - OBS_IC[i]) + (jc_car - OBS_JC[i]) * (jc_car - OBS_JC[i])) <= (r_car + RADIUS[i]) * (r_car + RADIUS[i])) {
        return true;
    }
    /*else if (fabs((ic_car - ic_enemy) * (ic_car - ic_enemy) + (jc_car - jc_enemy) * (jc_car - jc_enemy) <= (r_car + r_enemy) * (r_car + r_enemy))) {
        return true;
    }*/
    else return false;
}
```

However, the above technique was not implemented as near the end, minor issues has forced us to use another similar technique, it will be explained below.

Instead, we expanded on our obstacle detection function. This function starts by filtering the image and then finding the cg for each label in the first pass. The function is then called through each loop of the iteration. At the tenth iteration it will repeat the labeling and cg process, however, it will compare the different CG locations to the ones found in the first iteration. If there is a CG located within a couple pixels from a CG found in a previous iteration. Then it is safe to assume that the CG has not moved, that it is a immobile object and that it is thus an obstacle. The function then proceeds to paint in yellow any labeled pixed that is identified as belonging to an object. Thus, this function doesn't use color to find the CG, however it makes an assumption that if an object is immobile after 10 sim steps than it is an obstacle. A sample of the code can be seen below.

```
else if (firstpass == 1) {
    //cout << "first pass" << endl;
    for (int j = 1; j < nlabels; j++) {
        //cout << j << endl;
        centroid(a, label, j, ictemp[j], jctemp[j]);
        //draw_point_rgb(rgb, ictemp[j], jctemp[j], 255, 255, 255);
        view_rgb_image(rgb);
        //pause();
        for (int i = 1; i < nlabels; i++) {
            if ((ictemp[j] > (ic[i] - 2)) && (ictemp[j] < (ic[i] + 2)) && (jctemp[j] < (jc[i] + 2)) && (jctemp[j] > (ic[
            {
                // obstacles,save each obnstacle ic and jc and label=1

                obstacleic[w] = ic[i];
                obstaclejc[w] = jc[i];
                w++;
```

Initially, we wanted to paint it yellow only to get visual validation of the method above. However, after the minor issues during the implementation of the sobel function. We decided to use the yellow pixels to find the radius. Essentially, the pointer increments from the cg all the way until it finds a pixel that isn't yellow. The number of increments give us the radius. To validate our function we used infrared view and printed the radius which gave us the same results.



ROBOT EVASION ALGORITHMS

OBSTACLE HIDING ALGORITHM (NOT USED IN IT'S ENTIRETY)

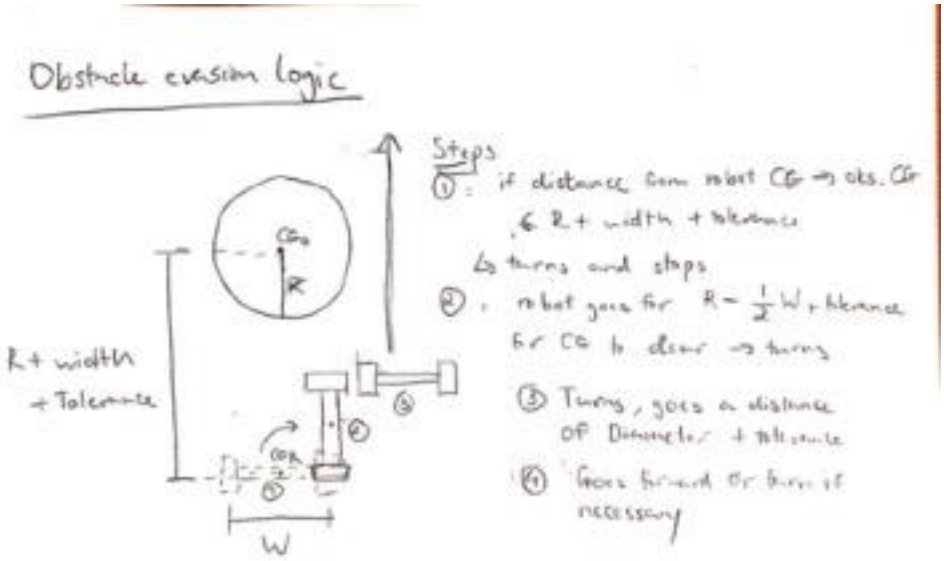
Originally our plan consisted of this. The obstacle evasion logic consists in the detection of the obstacle outlined above to modify the trajectory of the robot when an obstacle is encountered. In fact, if the robot is moving in a linear motion and approaches the obstacle within a range of the robots width and obstacle radius, found by calculating the distance separation the obstacle's centroid as well as the robot's centroid, the movement of the robot is altered. In fact, if the distance is equal to the range, the robot stops, performs a left or right turn depending on its location relative to the fixed obstacle's centroid in the x and y axes, moved forward until the centroid of the robot is superior to the obstacle's radius plus a tolerance and turns the opposite direction of the initial turn to resume its trajectory. The distance between both centroids is found using the following formula:

$$distance = \sqrt{(x_r - x_o)^2 + (y_r - y_o)^2} \leq R_o + w_r + tolerance$$

In order to determine whether the robot turns clockwise or counter clockwise, we compare its centroid in both axes to the obstacle's centroids. There are four scenarios, if the robot's centroids in the x and y are smaller than the obstacle's centroid, then the robot is located at the bottom left of the obstacle and it turns ccw (left). If the robot is located in the bottom right corner, the centroid in the x is larger than the obstacle's centroid in the x, and thus It turn cw (right) and then moves forward and left to resume its original trajectory. However, the opposite can be observed if the robot is located on the upper left of the obstacle as it turns cw first before moving forward and turning ccw to resume its original trajectory. When

the robot is located on the upper right of the obstacle, it turns ccw before moving forward to clear the obstacle and turning cw to resume its trajectory. This function was not implemented in the code.

The logic that was to be implemented is outlined in the figure below:



This algorithm proved to be difficult and we were unfortunately unable to implement it. The idea behind it was to use the obstacle's centroid as an axis of symmetry between the attacking and evading robot and as a result constantly be hiding behind it. However, we still ended up using some of the functions developed for it such as int hiding_obstacle_selection, which determines which obstacle is closest to our robot to use as an axis of symmetry. This would have allowed the robot to get the quickest hiding spot. We did not use the obstacle as an axis of symmetry but this function can still be used to determine which object are we on pace to collide. A function to determine the desired evading robot's location was developed and tested as well which will be shown in a later section in this report.

```
int hiding_obstacle_selection(double& myCGx, double& myCGy) {
        double distance;
        int selectedobject=0;
        double prev_distance = 1000000;
        for (int i = 0; i < w; i++) {
                distance = sqrt(((myCGx - obstacleic[i]) * (myCGx - obstacleic[i])) + (myCGy - obstaclejc[i]) * (myCGy -
obstaclejc[i]));
                if (abs(prev_distance) > abs(distance)) {
                        //cout << "distance= " << distance << endl;
                        prev_distance = distance;
                        selectedobject = i;
                }

        }
        //cout <<"obstacle cg : "<< obstacleic[selectedobject] << endl;
        //cout << "selected distance: " <<distance<< endl;
        return selectedobject;

}
```

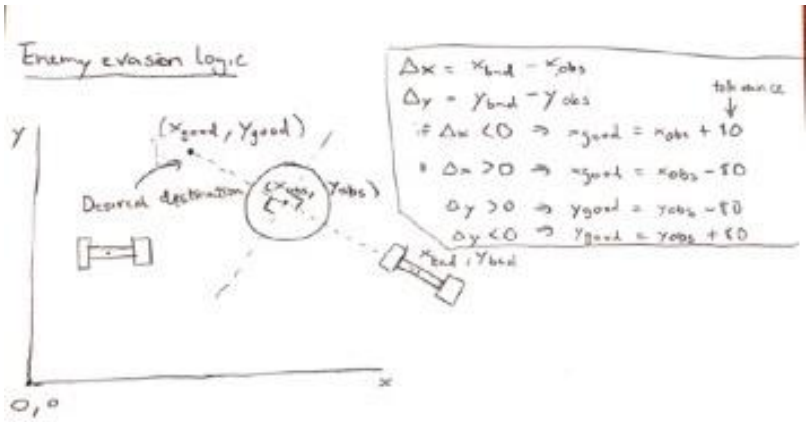
CONTOUR CIRCLING ALGORITHM

As a back up we tried to implement an algorithm where the robot contours the entire window. It's rotation and desired theta would be determined based on which quadrant it is in. A sample of the code can be seen below. We ended up using this one as it proved function albeit not without its flaws as well. In the sample below, the quandrant is determined in the if else ladder and the robot's movement is dictated as a result of it.

```
bool evasion_initial(double& robot_cx, double& robot_cy, int& pw_r, int& pw_l,double &theta) {
        static bool done = false;
        if (done)
                return true;
        else if (robot_cy > 240 && robot_cx < 340) {
                if ((theta < 270 - 10) || (theta > 270 + 10)) {
                        turn_CW(pw_r, pw_l);
                }

                else if (pinky < 420)robotforward(pw_r, pw_l);
                else {
                        pw_r = 1500;
                        pw_l = 1500;
                        done = true;
                }
        }

        else if (robot_cy < 240 && robot_cx < 340) {
                if ((theta < 90 - 10) || (theta > 90 + 10)) {
                        turn_CW(pw_r, pw_l);

                }

                else if (pinky >50)robotforward(pw_r, pw_l);
                else {
```

ROBOT/ENEMY DETECTION & EVASION

The logic we wanted to implement for calculating desired position based on enemy position is outlined in the figure below:



```
void obstacle_hiding(double& desired_x, double& desired_y, double& op_CGx, double& op_CGy, image& rgb,double& my_CGx, do
{
        int selectedObstacle = hiding_obstacle_selection(my_CGx, my_CGy);
        double deltaX = op_CGx-obstacleic[selectedObstacle]  ;
        double deltaY = op_CGy - obstaclejc[selectedObstacle] ;
        double xgood;
        double ygood;
        if (deltaX < 0) {
                 xgood = obstacleic[selectedObstacle] + radius_obstacle[selectedObstacle];
        }
        if (deltaX > 0) {
                xgood = obstacleic[selectedObstacle] - radius_obstacle[selectedObstacle];
        }
        if (deltaY > 0) {
                ygood = obstaclejc[selectedObstacle] - radius_obstacle[selectedObstacle];

        }
        if (deltaY < 0) {
                ygood = obstaclejc[selectedObstacle] + radius_obstacle[selectedObstacle];

        }
        draw_point_rgb(rgb, xgood, ygood, 0, 0, 0);
        //double distance = 50.0;
        //double term1 = atan((op_CGy - obstaclejc[selectedObstacle])/(op_CGx - obstacleic[selectedObstacle]));
        //double term2 = tan(3.14 / 2 - term1)*tan(3.14/2-term1);
//      cout <<"term1: "<< term1 << endl;
        //double nominator = sqrt(term2 + 1);
//      double ygood = distance / nominator + obstaclejc[selectedObstacle];
        //double xgood = (distance/nominator) * tan(3.14 / 2 - term1)+ obstacleic[selectedObstacle];
        draw_point_rgb(rgb, xgood, ygood, 0, 0, 0);
```

CONTRIBUTIONS

Member	List of Contributions
Faisal Bari	Edge Detection, obstacle/collision detection, robot detection, border control
Cyril Girgis	Obstacle detection and identification, radius calculation , attacking robot algorithm, code integration , opponent detection, Angle and distance computation between opponent ,our robot and obstacles.
Miriam Hebbassi	Check borders, obstacle evasion
Huy Hoang	Obstacle evasion , enemy evasion

Note: many of these tasks were done together, contributions were made by everyone.