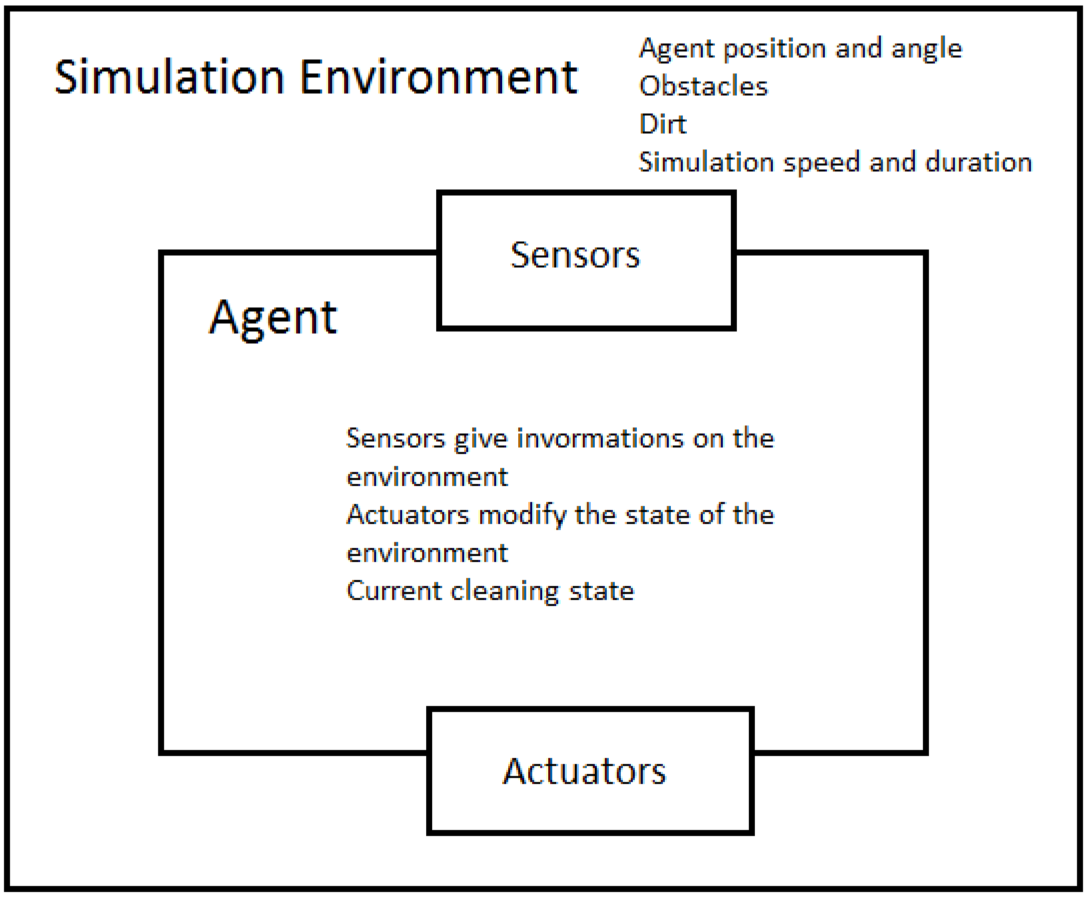
SIMULATION ENVIRONMENT

The agent and its simulation environment have been designed in such a way that the agent is not dependent on the simulation information and only the sensors and actuators can access them. This approach allows the agent to be implemented in any device without having to rewrite it, only the interface need to be implemented by the actual circuit to allow communication with the bumper, short range sensors or motors for the movement control.

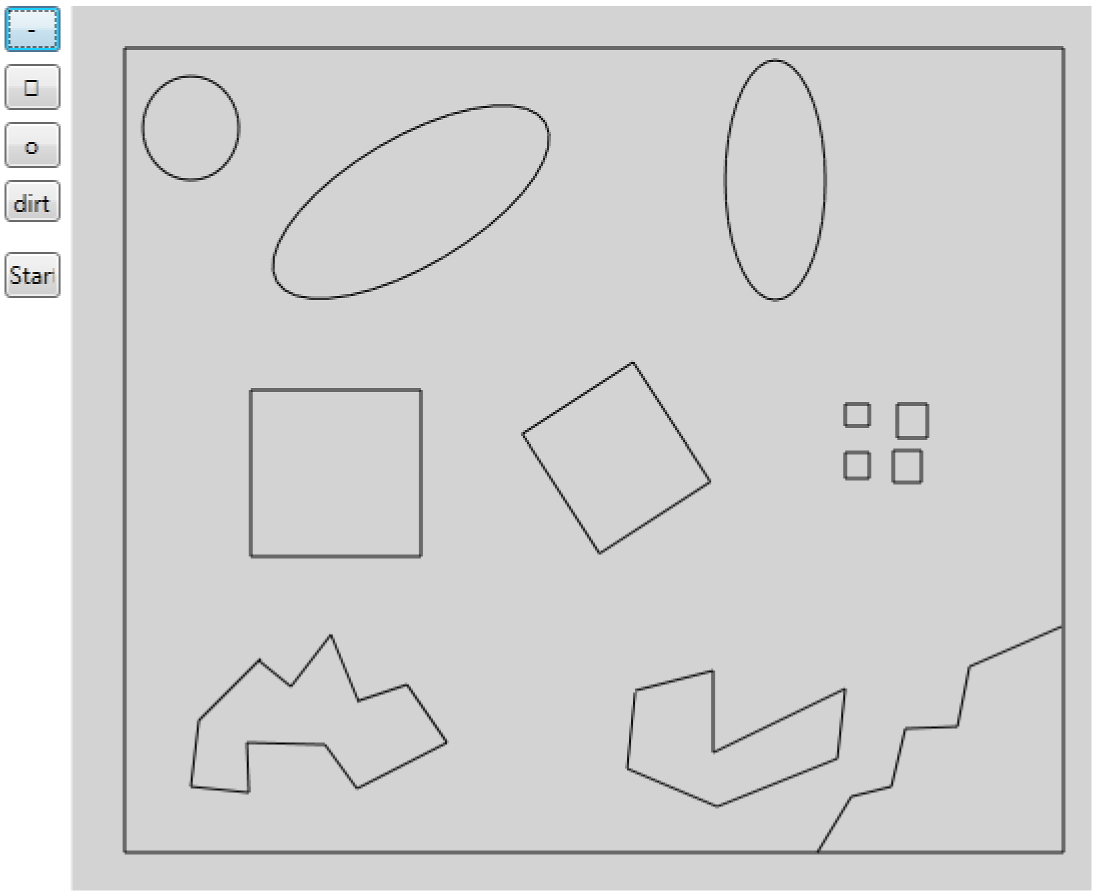

Simulation Environment
Agent position and angle
Obstacles
Dirt
Simulation speed and duration
Sensors
Agent
Sensors give invormations on the environment
Actuators modify the state of the environment
Current cleaning state
Actuators

Agent and Simulation Environment communication through Interfaces

SIMULATION ENVIRONMENT DESIGN

ROOM DESIGN

To provide a realistic and dynamic room design I have implemented room builder to allow the user to draw on the fly rectangles, ellipse and segments with a specified size and angle to see how the agent react in different type of environments. For the simplicity of the sensors simulation and efficiency of obstacles collisions, all obstacles are a set of segments and the ellipse approximation has been implemented over Danny Cohen parametric function for an ellipse approximation.



Environment design tools

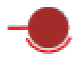
On the left the three tools segment, rectangle and ellipse can be used to draw obstacles on the room, in the case above three ellipses and rectangles have been drawn with different angles and set of shapes have been drawn at the bottom using the segment tool. One of the drawbacks of using segments only shapes is the memory usage for complex forms but in this software only 36 points for the ellipse approximation providing a good approximation and limiting the number of segments to 37. The dirt tool can be used to place “dirt” in the room and see how and when the agent will clean it.

PERFORMANCE MEASURE

The performance measure used at the moment is really basic and calculates the percentage of dirt removed at the end of the simulation; this approach provides a good idea of the cleaning performance with respect to the path followed by the agent. I looked for different approaches of the performance measurement during the design and implementation stages, one of them was not to use the dirt criteria but calculate the density of the robot footprints and calculate the uniformity; this idea was based on the wrong assumption that the dirt level in a room is uniform and that the room can be traversed at least once during the simulation time. A more complex and informative performance measure would be to use a ratio of area covered, the dirt removed for an associated value of current per hour used. Not having any concrete data of dirt density in a room I assumed in this exercise that the amount of dirt close to the walls is higher than in the rest of the room.

SENSORS

To allow the agent to interact with the environment two types of generic sensors have been implemented, the first one is the bumper which signal when the agent collides with one of the obstacle or wall in the room, the second one is a proximity sensor and can be used for either short or long range. The use of this sensors is the minimum required to provide random movements in the room and wall following.


Agent as represented in the simulation.

BUMPER

The bumper sensor have the same effect as a push button in an electronic circuit and is used to check if the agent collided with one obstacle, I have selected in my design to use an 130 degree bumper to cover the front of the robot without obstructing the short range sensor. The agent can only move forward so having information on the collision at the front is enough to ensure that every collision will be detected.

The bumper sensor can be implemented physically in a lot of different ways like using infrared to detect the proximity, by using a Wheatstone resistor bridge to detect the pressure applied or simply by using a limited amount of sensitive push buttons. The most appropriate solution for this is probably to use the push buttons as the power consumption will be dynamic and not static therefore current will be used only if a collision occurs. Using a bumper also provide an accuracy of almost 100% because only two states are available.

The implementation of the bumper can be seen in the appendix, the implementation is not efficient as the complexity is O(n) where n is the number of segments in the environment but is fast enough for the simulation purpose of this software (see Appendix : Intersection Benchmark). A more efficient data structure could have been used like an edge quadtree but the memory usage introduced and the complexity of the implementation does not worth it especially when 100000 segments can be checked in 0.02 milliseconds.

PROXIMITY SENSOR

The proximity sensor has been designed in a way that it can be used for short and long range proximity detection, this design of the agent only one short proximity sensor on the right hand side to be able to use the Right Hand Maze Solving Algorithm, in this case not for maze solving but for wall following and angle calculations.

Based on a datasheet for a short range proximity sensor (see Appendix: References) an infrared proximity sensor draw a maximum of 50mA of current while its used thus not much as a mobile phone batteries have a rating of about 1800 mA per hour therefore could drive one of this sensor for a day and a half at full power. Because the proximity sensor is used for angle calculations the distance must be quite accurate or the wall following will not work as well as expected even if the agent tries to readjust its angle all the time.

The implementation of the proximity sensor is the based on the same principle as the bumper implementation described before, a segment of the maximum length of the proximity sensor is placed at the position of the agent and checked if it intersects with one of the obstacles. (See Appendix: Proximity Sensor Code for reference)

ACTUATORS

The agent is built upon two actuators, a cleaner that provides the vacuuming functionality and the movement one to control the motors. Like the sensors the actuator are based on an interface that allows an implementation on a real device easier and split the logic between the simulation and the behaviour.

CLEANER

The functionality of the cleaner is to remove the dirt in proximity of the agent position. Based on the datasheet of the Neato XV-11 (see Appendix) the DC motor used for cleaning is 12V, 3.84A therefore the vacuum motor have a much greater impact on the energy consumption of the robot than the sensors.

To implement the cleaner functionality efficiently I used a two dimensional KDTree to provide a O(log(n)) search where n is the number of elements in the tree. To be able to remove more than one dirt point at the same time the Range search algorithm as described by Gonnet and Baeza-Yates have been implemented (see Appendix: Recursive Range Search Code) returning a set of points that lies in the range defined by a upper and lower boundary.

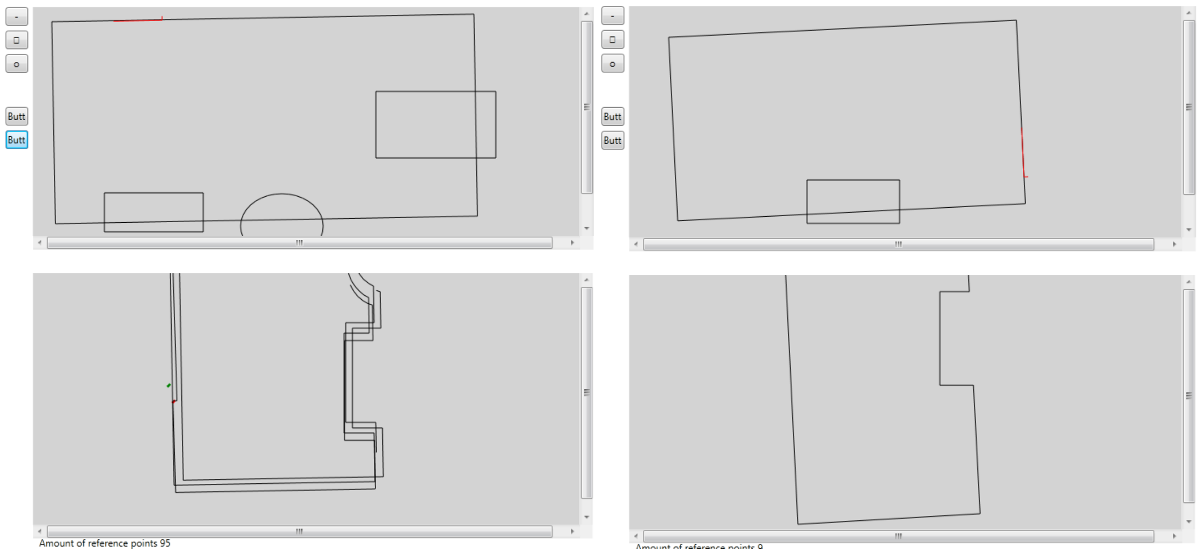
MOVEMENT

The movement actuator provide the basic movement functionality to the agent by providing three functions, one to move straight, one to turn left and one to turn right, this three functions required the speed to be specified such that the simulation can be realistic by introducing movement and rotation times. The Roomba wheels motors draw about 0.8A at max power (see Appendix: References) therefore 1.6A if we want to wheels to drive the agent.

The Movement actuator has been implemented by moving the agent by only one pixel (one pixel is one centime in the environment scale) the implementation can be seen in Appendix: Movement Actuator Code. The simulation can be speeded up by decreasing the latency induced by the movement operations by modifying the simulation speed.

AGENT DESIGN

The final version (revision 25) of the agent uses a reactive approach and decides what the next action needs to be when a wall is reached. From revisions 12 to 21 the agent was mapping the room and storing the reference points on a modified version of the KDTree defined in the cleaner actuator above. The insert function of the tree had an accuracy parameter allowing two points to be set as identical if the accuracy distance was greater or equal to the distance between the point and one of the existing node in the tree. This approach was supposed to allow the agent to be able to go back to its base when the battery runs out using a continuous space version of the A* algorithm and estimates where the agent should go to optimize the cleaning path. However this solution had different issues, the main one was that the sensors had to be really accurate to provide the distance travelled since the last reference point had been inserted in the tree.



Left: Error introduced in the Room mapping   Right: Proper mapping without error

In this version (above) of the agent, the agent was represented only by a dot and was using to proximity sensors, a short range on the right and a long range at the front, by using the position of the agent when a wall was hit or the agent was turning and measuring the angle turned and the distance travelled the map could be drawn however when the code switched to a single dot agent to a circular agent the reference point were much harder to get accurately. In the physical device, a CCD sensor (like on optic mouse) could provide really accurate readings but are expensive.

The reactive approach used at the moment is fairly basic, the agent can be in two states either random state where it turns by a random angle or in wall following mode and the agent follows the wall it’s against.

STATES AND ACTION LOOPS

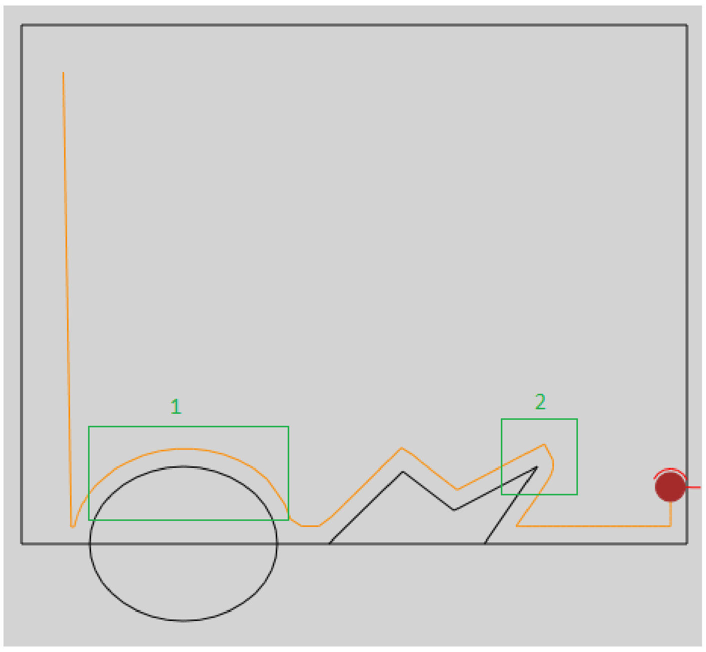
The mechanism to determine if the agent should switch in Random or Wall Following mode is based on the amount of time spent in each mode, to provide this the agent requires an internal clock that will probably be provided by the microcontrollers driving it.

```
// find what to do
if (wallTime > randomTime)
    state = EState.RANDOM;
else
    state = EState.WALLFOLLOWING;
```

WALL FOLLOWING

The wall following loop is based on the right hand maze solving algorithm using the short range sensor as the “right hand”, the agent move straight and every times it moves the value of the short range sensor is read, using the delta between two readings and some trigonometry an optimal angle can be calculated, the optimal angle is calculated to reduce the amount of time turning as while the agent turns it does not clean new areas of the room. Two conditions have been added to cope with the end of a wall or any corners with an angle of more than 90 degrees, if the previous reading is valid but the new one is not the agent will turn by 90 degree and try to find the wall, in case the previous reading is invalid the angle is not modified and angle might not be optimal until the next iteration.

```
while (!bumperSensor.collide())
{
        double alpha;

        movement.straight(10);
        cleaner.clean();
        newReading = shortRangeSensor.getDistance();

        // if no previous reading do not modify the angle
        if (prevReading == -1)
                alpha = 0;

        // if we are not against a wall anymore do a 90 degree turn
        // to look for the wall
        else if (newReading == -1)
        {
                for (int i = 0; i < 10; i++)
                {
                        if (bumperSensor.collide())
                                break;
                        movement.straight(10);
                       cleaner.clean();
                }
                alpha = 90;
        }
        // if we got the previous and current reading calculate the inverse tangent
        // of the angle that would put the agent in the right angle
        else
                alpha = Math.Atan((newReading - prevReading) / 1) * 180 / Math.PI;

        movement.spinRight(45, alpha);
       cleaner.clean();
       prevReading = newReading;
}
```



The orange lines represent the path travelled by the agent over the room; this information is known only by the environment.

1. The agent is able to follow curves by adjusting its angle ass it move forward, this behaviour is always active but is more noticeable on curves.

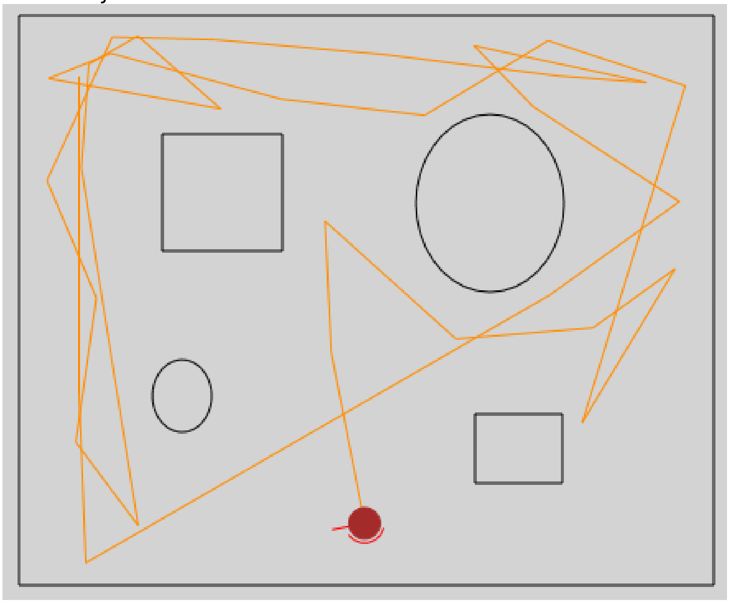
2. At this point the short range sensor is not able to read the distance, therefore the agent turns right by 90 degrees and is able to restart the wall following.

Wall following behaviour of the agent

RANDOM

In random mode the agent will go straight until it hits a wall and then will turn by a random angle until the bumper is not against a wall and go straight again, this way is really simple but provide a high probability that the entire room can be cleaned however the time for the cleaning is not optimal.

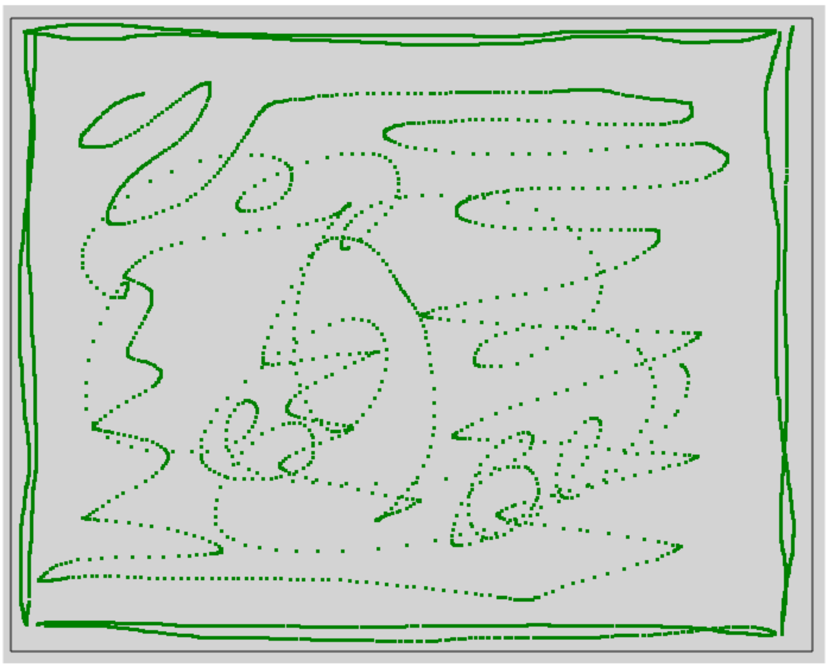
```
if (state == EState.RANDOM)
{
        while (bumperSensor.collide())
        {
                // spin left
                movement.spinLeft(45, random.Next(0, 360));
                cleaner.clean();
        }

        while (!bumperSensor.collide())
        {
                // move straight at 10cm/s
                movement.straight(10);
                cleaner.clean();
        }
}
```



This shows the behaviour of the random angle while the agent hit a wall, as it can be seen in this the agent spent more time on the left and at the top than the rest of the room, however over time the probability of covering the entire room should be equivalent.

QUANTITATIVE EXPERIMENTS

The simulation have been ran 25 times on four different scenarios, the first two are to test the efficiency of the wall following and random against only random in an empty room, in both cases the agent was simulating a one hour run. The environment used for the empty room is displayed below.

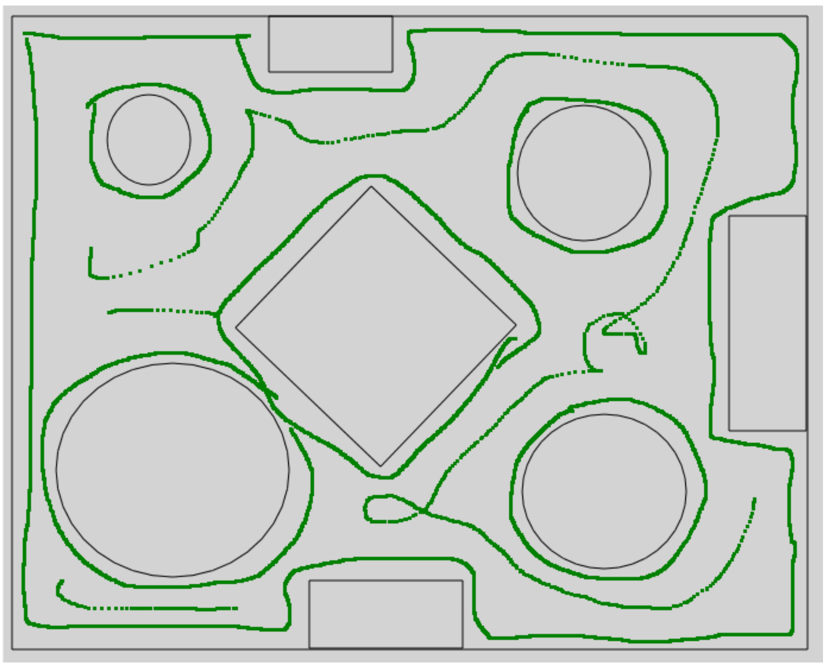


Random			Wall Following	
5256	5351		4845	5351
5284	5351		5143	5351
5279	5351		5258	5351
5277	5351		5202	5351
5292	5351		4991	5351
5296	5351		5233	5351
5298	5351		5230	5351
5295	5351		5056	5351
5298	5351		5181	5351
5281	5351		5009	5351
5271	5351		4961	5351
5267	5351		5127	5351
5280	5351		4974	5351
5298	5351		5164	5351
5267	5351		5058	5351
5293	5351		5266	5351
5236	5351		5121	5351
5274	5351		5194	5351
5289	5351		5194	5351
5292	5351		5071	5351
5296	5351		5160	5351
5282	5351		5106	5351
5285	5351		5139	5351
5290	5351		5100	5351
5295	5351		4923	5351
Average %	98.72622		Average %	95.46328

This test shows that based on the defined performance measure (percentage of dirt removed) that in an empty room with a simple rectangle as boundaries, the Random only solution performs better than the wall following and random solution.

The difference is only of 3% and in both cases the agent did reasonably well, but the wall following having worst performance show an issue in the algorithm that in this case the time spent following the walls should be lower than the time cleaning the rest of the room.

The third and fourth test scenarios are to test the same algorithms as before but this time in a room with more obstacles like the one presented below.



Random			Wall Following	
2656	6868		3535	6868
2978	6868		3068	6868
2097	6868		3397	6868
2818	6868		3280	6868
1978	6868		3016	6868
3049	6868		3199	6868
2928	6868		2328	6868
3389	6868		3240	6868
2074	6868		4126	6868
2927	6868		3405	6868
2497	6868		3218	6868
1760	6868		2823	6868
1944	6868		3611	6868
2550	6868		1652	6868
3074	6868		2023	6868
2234	6868		3512	6868
2031	6868		2586	6868
3061	6868		3455	6868
3064	6868		2892	6868
3281	6868		3094	6868
2813	6868		4230	6868
2628	6868		3093	6868
2559	6868		3328	6868
3033	6868		3305	6868
2267	6868		2862	6868
Average %	38.25859		Average %	45.58998

In the condition of a busier room the wall following algorithm performs better than the random one but in both cases the performance is quite poor as in one hour less than 50% of the room has been cleaned.

A possible reason of both algorithm performances being poor on a room like this is the fact that they change their angle only when hit a wall.

APPENDIX

BUMPER COLLISION CODE

```
public bool collide()
{
    // Retrieve the environment singleton
    Environment env = Environment.Instance;

    // Create the initial segment for the angle and starting point of the bumper
    double a = angle / nbsegments;
    Segment vector = new Segment(env.AgentPosition, env.AgentPosition
         + new Point(-radius, 0));
    vector = vector.Rotate(-(180 - angle) / 2);
    vector = vector.Rotate(env.AgentAngle);

    // Create the bumper segments
    for (int i = 0; i < nbsegments; i++)
    {
        Segment seg = new Segment();
        seg.Start = vector.End;
        vector = vector.Rotate(-a);
        seg.End = vector.End;

        segments[i] = seg;
    }

    // Check if any of the segment collides
    lock (env.Segments)
    {
        foreach (Segment s in env.Segments)
        {
            foreach (Segment b in segments)
            {
                if (b.Intersects(s) != null)
                    return true;
            }
        }
    }

    return false;
}
```

PROXIMITY SENSOR CODE

```
public double getDistance()
{
    // Retrieve the environment
    Environment env = Environment.Instance;
    double minDistance = -1;

    // Create the offset segment
    Segment offset = new Segment();
    offset.Start = env.AgentPosition;
    offset.End = env.AgentPosition + new Point(0, lengthOffset);

    // Create the range segment
    Segment full = new Segment();
    full.Start = env.AgentPosition;
    full.End = env.AgentPosition + new Point(0, length + lengthOffset);

    range = new Segment();
    range.Start = offset.Rotate(env.AgentAngle + angleOffset).End;
    range.End = full.Rotate(env.AgentAngle + angleOffset).End;

    // Go through the list of segments, not neat O(n) but execution
    // fast enough for this simulation purpose
    foreach (Segment s in env.Segments)
    {
        Point p;
        if ((p = range.Intersects(s)) != null)
```

```
            {
                double v = p.Distance(range.Start);
                if (v < minDistance || minDistance == -1)
                {
                    minDistance = v;
                }
            }
        }

        return minDistance;
    }
```

CLEANER ACTUATOR CODE

```
    public void clean()
    {
        // Retrieve the dirt points from the Environment instance that
         // are under the agent in 16 by 16 square
        // centered on the agent coordinates
        List<BDNode> nodes = Environment.Instance.Dirt.Range(
                    Environment.Instance.AgentPosition - refPoint,
                    Environment.Instance.AgentPosition + refPoint);

        // thread safety
        lock (Environment.Instance.RemovedDirt)
        {
            foreach (BDNode n in nodes)
            {
                // Check if the dirt density is 0 if true remove the
                  // dirt marker else decrease the density
                DirtElement elem = (DirtElement)n.Value;
                elem.count--;
                if (elem.count == 0)
                    Environment.Instance.RemovedDirt.Add(elem.element);
            }
        }
    }
```

MOVEMENT ACTUATOR CODE

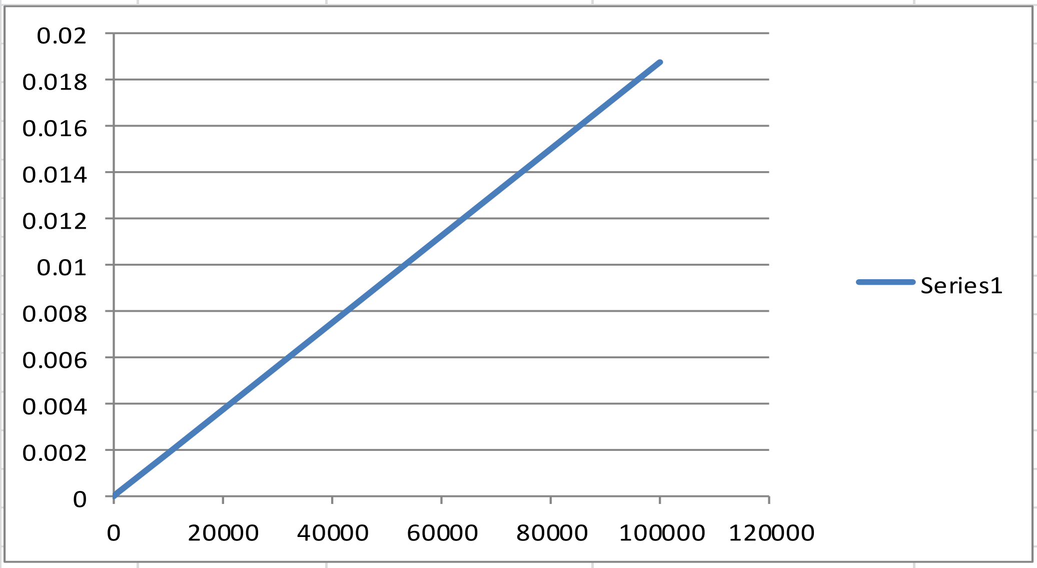
```
    public void straight(int speed)
    {
        // Retrieve the environment singleton
        Environment env = Environment.Instance;

        // Calculate the next position
        Segment nextPos = new Segment();
        nextPos.Start.X = env.AgentPosition.X;
        nextPos.Start.Y = env.AgentPosition.Y;
        nextPos.End.X = env.AgentPosition.X;
        nextPos.End.Y = env.AgentPosition.Y + 1;
        nextPos = nextPos.Rotate(env.AgentAngle);

        env.AgentPosition = nextPos.End;

        // Sleep to give the speed limitation behaviour
        System.Threading.Thread.Sleep((int)(1000 / (speed * env.SimulationSpeed)));
    }

    public void spinLeft(int speed, double angle)
    {
        Environment env = Environment.Instance;

        for (int i = 0; i < Math.Abs(angle); i++)
        {
            if (angle < 0)
                env.AgentAngle += 1;
            else
                env.AgentAngle -= 1;

            // Sleep to give the speed limitation behaviour
            System.Threading.Thread.Sleep(((int)((1 * 1000) / (speed * env.SimulationSpeed))));
        }
    }

    public void spinRight(int speed, double angle)
    {
        Environment env = Environment.Instance;

        for (int i = 0; i < Math.Abs(angle); i++)
        {
            if (angle > 0)
                env.AgentAngle += 1;
            else
                env.AgentAngle -= 1;

            // Sleep to give the speed limitation behaviour
            System.Threading.Thread.Sleep(((int)((1 * 1000) / (speed * env.SimulationSpeed))));
        }
    }
```

RECURSIVE RANGE SEARCH CODE

```
    private static void rsearch(Point lowk, Point uppk, BDNode tree, int level, List<BDNode> v)
    {
        // if the tree is null stop the recursivity
        if (tree == null)
            return;

        // if the current working dimension is smaller or equal to the current node value
        // recursive range search on the left sub tree
        if ((level % 2 == 0 && lowk.X <= tree.Coord.X) ||
             (level % 2 == 1 && lowk.Y <= tree.Coord.Y))
            rsearch(lowk, uppk, tree.Left, level + 1, v);

        // if the keys does not lies in the node range
        int j = 0;
        if (lowk.X <= tree.Coord.X && uppk.X >= tree.Coord.X)
        {
            j++;
            if (lowk.Y <= tree.Coord.Y && uppk.Y >= tree.Coord.Y)
                j++;
        }

        // if the node lies in the boundary values
        if (j == 2)
            v.Add(tree);

        // if the current working dimension node is less than the upper key boundary
        // recursive range search on the right sub tree
        if ((level % 2 == 0 && uppk.X > tree.Coord.X) ||
             (level % 2 == 1 && uppk.Y > tree.Coord.Y))
            rsearch(lowk, uppk, tree.Right, level + 1, v);
    }
```

INTERSECTION BENCHMARK

segments	iterations	Time (ms)	Average Time (ms)
10	100	0.0010001	0.000010001
100	100	0.0020001	0.000020001
1000	100	0.0190011	0.000190011
10000	100	0.1850106	0.001850106
100000	100	1.8811076	0.018811076



REFERENCES

Short Range Infrared proximity sensor: http://www.farnell.com/datasheets/90547.pdf

Neato XV-11 Vaccum Motor datasheet:
http://www.robotreviews.com/chat/download/file.php?id=3656

Roomba wheel motor datasheet:
http://www.standardmotor.net/sc_vas/pdf_generation/product_pdf.php?id=60