A Customizable Simulator for Q-Learning

Lavi Zamstein
University of Florida
dino-man@ieee.org

Dr. A. Antonio Arroyo
arroyo@mil.ufl.edu,
Dr. Eric M. Schwartz
ems@mil.ufl.edu
University of Florida

Dr. Carl D. Crane
ccrane@ufl.edu
University of Florida

ABSTRACT
In this paper, we describe the learning process of an autonomous delivery robot. We will discuss the simulator used to aid in the reinforcement learning process and the design involved in creating it.

Keywords
Q-learning, reinforcement learning, simulator

1. INTRODUCTION
Koolio is part butler, part vending machine, and part delivery service. He stays in the Machine Intelligence Laboratory at the University of Florida, located on the third floor of the Mechanical Engineering Building A. Professors with offices on that floor can access Koolio via the internet and place an order for a drink or snack. Koolio finds his way to their office, identifying it by the number sign outside the door.

Koolio learns his behavior through the reinforcement learning method Q-learning.

2. KOOLIO
Koolio is controlled by an integrated system consisting of a single board computer along with a pair of smaller processors, used to organize sensor data and to drive the motors [3].

The platform itself consists of a small refrigerator mounted on a circular base. The sonar and bump sensors are located on the base, with a digital compass on top of the refrigerator. Also mounted on the base are a pair of tall poles, which hold up the side-facing cameras, side sonar sensors, and the LCD screen.

3. REINFORCEMENT LEARNING
Reinforcement learning is a method of learning by way of rewards and punishments. The learning agent will seek those choices that result in high rewards and avoid those actions that result in low rewards or punishments (negative rewards). In this way, an agent will learn to follow the decision path that results in the best possible reward.

Many reinforcement learning processes will only work properly if a Markov process is assumed. A Markov process is any process that uses only current values of inputs to determine the output. All past input values are ignored in Markov processes. This makes Markov processes very useful in the realm of robots, since the output and all calculations will rely only on the current input of the sensors, rather than memory of any past input. Not only does this remove the necessity of memory for past sensor values, but it also reduces the calculations greatly, as there are fewer variables to deal with – only the current sensor values instead of the current and many past sensor values.

One of the fundamental balances in reinforcement learning is that between exploration and exploitation. Since the agent wants to maximize its reward, it will often make the choice that offers the highest reward. However, if this best choice is always made, then many other possible choices are never looked at. It is possible that some choices that yield smaller rewards will, in the end, give higher returns, due to better states following a single sub-optimal choice now. To avoid the skipping of such cases, there must be some exploration of non-optimal states, in addition to exploitation by picking the current best choice.

4. LEARNING PROCESS

4.1 Step 1: States and Actions
For a mobile robot, the states are simply the set of all sensor input. Because most of the sensors used on Koolio have analog outputs, there is the potential for a very large number of possible states. Therefore, states must be defined using ranges of sensor values.

Aside from basic informational output such as the LCD screen, the only actions are movement of the wheels. Since the motor drivers operate independently from the learning process, it does not need to put into consideration the specifics of operating the motors directly. The only actions then are the basic movements of forward, backward, turn left, turn right, rotate left, and rotate right.

4.2 Step 2: Simulation
Reinforcement learning in episodic tasks requires a very large number of repetitions to learn. Along with this, many repetitions on the platform can result in wear of parts, and the testing area of a hallway may not always be available since it is used on a daily basis. Because of these factors, the initial parts of learning must be done in simulation.

Learning in simulation has other advantages as well. Data can be easily stored from simulation runs and referred to later [4]. Using simulation data, a graph can be constructed showing the growth of the return as the policy approaches optimal. If data is recorded on the actual robot, graphs can also be constructed. Simulations can also run much faster than a real robot, so many simulations can be run in the time of a single real robot episode.

There are some drawbacks to simulation, however. No matter how good a model of the environment for running test episodes is, it can never be perfect. A learned policy from a simulation may

not operate correctly in a real environment because of any number of imperfections in the model.

However, simulation is still an important tool in the learning process. By developing an optimal policy in the simulated environment, much of the time of real robot learning can be done before involving the actual robot. The policy can then be exported to the robot and the learning process can continue.

4.3 Step 3: Real Robot Learning

Once the simulation has reached an optimal policy, it can be brought to the robot to continue the learning. Because a good deal of learning has already taken place, this phase of the learning process is much faster than it would have been if the simulation was skipped and the learning was done solely in the real environment. By the time the policy is ready to be used on the platform, it will have been changed to avoid many time-consuming mistakes such as random wandering.

Despite the shortcuts of using a simulator for the initial learning process, this phase of real robot learning is still the most time-intensive, as episode runs of the robot can take several minutes instead of the accelerated time used in simulation. Because the policy is already refined, however, only a relatively smaller number of episodes is required for reaching a new real environment optimal policy.

5. SIMULATOR

5.1 Reason for Simulation

Reinforcement learning algorithms such as Q-learning take many repetitions to come to an optimal policy. In addition, a real robot in the process of learning has the potential to be hazardous to itself, its environment, and any people who may be nearby, since it must learn to not run into obstacles and will wander aimlessly. Because of these two factors, simulation was used for the initial learning process. By simulating the robot, there was no danger to any real objects or people. The simulation was also able to run much faster than Koolio actually did in the hallway, allowing for a much faster rate of learning.

5.2 First Simulator

The first simulator chosen for the task of Koolio's initial learning was one written by Halim Aljibury at the University of Florida as part of his MS thesis [1]. This simulator was chosen for several reasons. The source code was readily available, so it could be edited to suit Koolio's purposes. This simulator was written in Visual C. Figure 1 shows the simulator running with a standard environment.

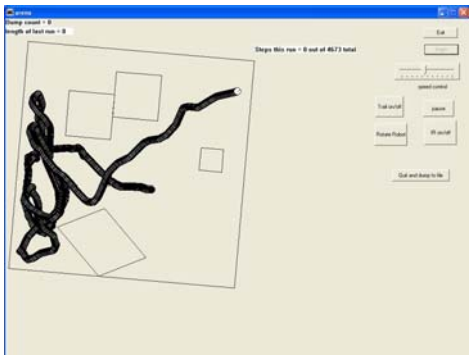

Figure 1. The first simulator in action.

The simulator was specifically programmed for use with the TJ Pro robot [2] (see Figure 2). The TJ Pro has a circular body 17 cm in diameter and 8 cm tall, with two servos as motors. Standard sensors on a TJ Pro are a bump ring and IR sensors with 40KHz modulated IR emitters. Although Koolio is much larger than a TJ Pro, he shares enough similarities that the simulator can adapt to his specifications with no necessary changes. Like the TJ Pro, Koolio has a round base, two wheels, and casters in the same locations. Since the simulation only requires scale relative to the robot size, the difference in sizes between the TJ Pro and Koolio is a non-issue.

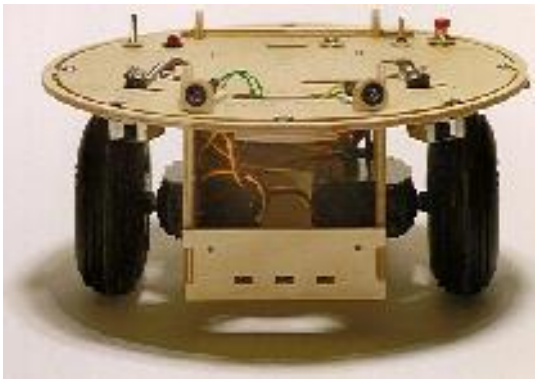

Figure 2. The TJ Pro.

5.3 Simulator Considerations

Simulations have several well-known shortcomings. The most important of these shortcomings is that a simulated model can never perfectly reproduce a real environment. Local noise, imperfections in the sensors, slight variations in the motors, and uneven surfaces can not be easily translated into a simulation. The simplest solution, and the one chosen by this simulator, is to assume that the model is perfect.

Because of differences in the motors, a real robot will never travel in a perfectly straight line. One motor will always be faster than the other, causing the robot to travel in an arc when going 'straight' forward. For the purposes of the simulation, it is assumed that a robot going straight will travel in a straight line. In the long run of robot behavior, however, this simplification of an arc to a straight line will have minimal effect, since the other actions performed by Koolio will often dominate over the small imperfection. Another concession made for the simulator is to discount any physical reactions due to friction from bumping into an object. It is assumed that Koolio maintains full control when he touches an object and does not stick to the object with friction and rotate without the motors being given the turn instructions.

However, the first chosen simulator had several issues that made it unsuitable for the learning task. It was difficult to make an arena with a room or obstacles that were not quadrilaterals, since the base assumption was that all rooms would have four sides. This assumption does not apply for many situations, including the hallway in which the robot was to be learning.

In addition, while the simulator could be edited to add new sensors or new types of sensors, the code did not allow for easy changes. Since several different sensor types were required for the simulated robot, a simulator that allowed for simple editing to add new sensors was required.

The simulator also assumed a continuous running format. However, the task for which reinforcement learning was to be performed is episodic, having a defined beginning and end. The simulator was not made to handle episodic tasks.

Because of these reasons, a new simulator was required to perform the learning procedures necessary for the task.

5.4 New Simulator

A new simulator was made to better fit the needs of an episodic reinforcement learning process. In order to do this, the simulator needed to be able to automatically reset itself upon completion of an episode. It also needed to be as customizable as possible to allow for different environments or added sensors to the robot.

When creating this new simulator, customization was kept in mind throughout. The intention was to make a simulator able to perform learning tasks for Koolio under various situations, as well as to be easily alterable for use with other robots for completely different learning tasks.

5.4.1 Arena Environment

The simulation environment, referred to as the arena from this point, is an area enclosed within a series of line segments. These segments are not limited in length or number. The line segments are defined by their endpoints, which are listed in the *SegmentEndpoint* array in order. When the simulation begins, it constructs the arena by drawing lines between consecutive points in *SegmentEndpoint* until it reaches the end of the array. Once the end has been reached, the arena is closed by drawing a line from the final point back to the first.

This treatment of the arena walls assumes that the arena environment is fully enclosed. Obstacles that touch the walls can be created by altering the arena walls to include the obstacle endpoints. Obstacles that do not touch the walls are not included in the simulator by default, but they may be added if desired. This can be accomplished by creating an additional array of obstacle endpoints and checking this array at every instance the walls are checked for movement and sensor purposes.

Once the arena walls have been entered, they are rotated five degrees clockwise. This rotation helps to prevent infinite slopes while calculating sensor distances, since there will no longer be perfectly vertical walls in the arena. However, error checks are still in the code to prevent infinite slopes entering any of the calculations.

In addition to the walls, several other landmarks were required for Koolio's episodic learning task. Since Koolio must be able to detect markers for the goal and the end of the world, these markers must also be represented in the simulation.

The simulator assumes a single goal point, defined in *GoalPoint*. If more than one goal is desired, the code must be altered to allow for this.

Multiple markers for the end of the world exist, defined in *EndOfWorldPoints*. There is not a limit to the number of end of the world markers allowed in the simulator.

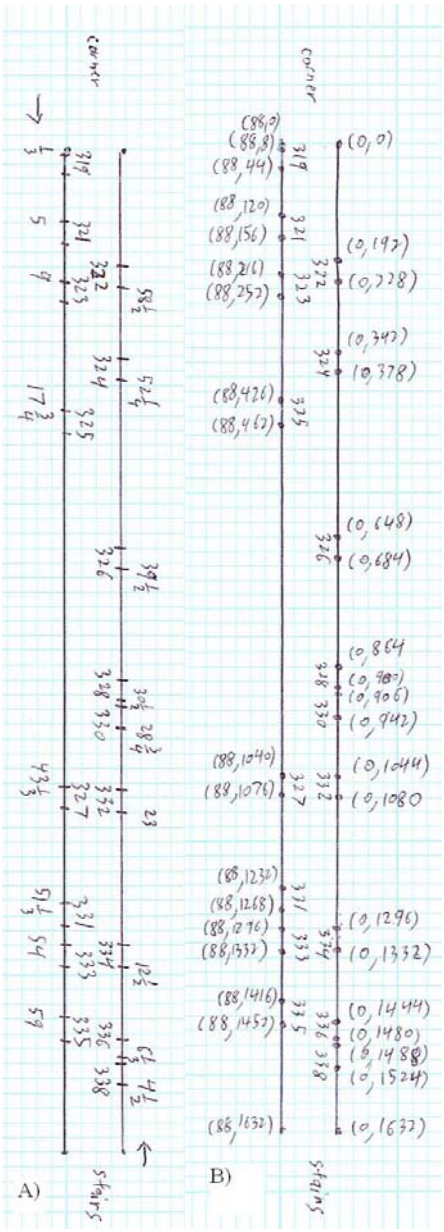

Figure 3. Scale model of the hallway environment. Room numbers indicated within the hallway, measurements outside. A) Tile measurement. The tile measurement is a count of the tiles from the end of the hallway in the direction of the arrow indicators to the leading edge of the doorway. The even side rooms begin counting tiles from the stairs side of the hall, while the odd side rooms begin from the corner side of the hall. B) Coordinate measurement. Using 36 pixels per tile side, this is an absolute coordinate scale of the door edges in inches.

For the initial simulation of Koolio, the arena was made as a long, narrow rectangle with T-shaped openings on either end to approximate the shape of the hallway. Ceiling tiles in the actual hallway were used as a constant-size measuring tool. Each ceiling tile is approximately twenty-four inches square and is slightly larger than Koolio's base, at twenty inches in diameter. The easiest conversion from reality to simulation was to make

each inch one pixel in the simulator. The hallway is 68 ceiling tiles long and 3 2/3 tiles wide, giving it simulator dimensions of 1632 pixels long and 88 pixels wide. Each door in the hallway is 1 1/2 ceiling tiles wide, converting to 36 pixels in the simulator. Each doorway has a plate on either the right or the left side to identify the room by name and number. Figure 3 shows the hallway environment without the T-shaped areas on the end with room numbers measured in both number of tiles and pixel coordinates.

5.4.2 Robot

Within the simulator, the robot is defined by the location of its center point, *RobotLocation*, its heading, *RobotOrientation*, and its radius, *RobotRadius*. The radius is a constant that cannot be changed, but the location and heading change as the code runs. The simulator assumes that the robot base is either circular in shape or can be easily simplified into and represented by a circle.

The two values of *RobotLocation* and *RobotOrientation* are the sole indicators of the robot's position and heading within the arena. These two values are used in determining the movements of the robot, as well as affecting the sensor readings.

5.4.3 Sensors

Koolio has several different types of sensors: a compass, an array of sonar sensors, and three cameras. These three types of sensors were implemented into the simulator.

5.4.3.1 Compass

The compass is the simplest of the sensors implemented in the simulator, since it requires only a single input, the robot's heading. The compass reading is calculated in *CalculateCompass*.

Since *RobotOrientation* is a value between 0 and 2π, it is divided by 2π and multiplied by 256 to get a value between 0 and 255 for the heading. Since the compass used for Koolio is very inaccurate, 25% noise is inserted into the signal.

The *SensorEncodeCompass* function takes this reading and compares it to known values for Northwest, Northeast, Southwest, and Southeast. The compass reading is encoded as *North* if it lies between Northwest and Northeast, as *South* if it lies between Southwest and Southeast, as *East* if it lies between Northeast and Southeast, and as *West* if it lies between Northwest and Southwest.

Because the actual compass used on Koolio was very unreliable, it was decided to remove the compass from all calculations in the simulator. The code still exists, however, and can be used or altered for any similar sensor in a future simulation.

5.4.3.2 Sonar

The sonar sensors are defined by their location on the robot in relation to the front. These angles are listed in *SonarSensors*. The order of the sensors in that list is tied to the arbitrarily-assigned sensor numbers later in the code. More sensors can be added and the order of sensors can be rearranged, but that must also be done in all uses of the sensors. The viewing arc of the sonar sensors is 55 degrees, defined in the constant *SonarArc*.

The sonar sensors used on Koolio return distance values in inches. If different sonar sensors are used, then an additional function will be required to convert real distance into the reading that the sensors output.

Each of the sonar sensors is read individually in *CalculateSonar*. The first step in reading the sensor's value is to split the viewable arc of the sensor into a number of smaller arcs. For each of these smaller arcs, the distance to the nearest wall is calculated. The reading from the sonar is the shortest of these distances.

In order to calculate this distance for each smaller arc, the simulator first finds the equation of the line that passes through the center of that arc. Since straight line can be defined by a point and a slope, all the needed information for the equation of that line is available. The slope of the line is determined by the angle between the center of the arc and the robot's heading. The point needed for the line comes from the robot's location.

For each of these lines through the smaller arcs, the distance to each wall along that line must be checked. This is done in *LineIntersectDistance*, a function that takes the equation of the sensor line and the endpoints of a wall as inputs and returns the distance from the robot to that wall along the sensor line segment. Table 1 shows the variables used in this function.

Table 1. Variables used in *LineIntersectDistance*.

Variable	Use
Slope1	slope of the sonar line segment
b1	Y-intercept of the sonar line segment
Xa	X-coordinate of one end of the wall
Ya	Y-coordinate of one end of the wall
Xb	X-coordinate of the other end of the wall
Yb	Y-coordinate of the other end of the wall
SensorTheta	angle between the front of the robot and the center of the sensor arc

Using the endpoints for the wall, the equation of the line containing that wall is found (see Table 2).

Table 2. Equations for finding the equation of the line containing the wall segment.

$$Slope2 = \frac{Ya - Yb}{Xa - Xb}$$

$$b2 = Ya - Slope2 * Xa$$

Using the two equations of the lines, the point at which the two lines intersect is found (see Table 3).

Table 3. Equations for finding the intersection (Ix,Iy) between two line segments.

$$Y1 = Y2$$

$$Slope1 * Ix + b1 = Slope2 * Ix + b2$$

$$Ix = \frac{b2 - b1}{Slope1 - Slope2}$$

$$Iy = Slope1 * Ix + b1$$

Once the intersection point is found, it must be converted from Cartesian coordinates to polar coordinates in relation to the location of the robot (see Table 4).

Table 4. Conversion of (Ix,Iy) to polar coordinates (PolarR,PolarTheta).

$$PolarX = LocationX - Ix$$

$$PolarY = LocationY - Iy$$

$$PolarR = \sqrt{PolarX^2 + PolarY^2} - RobotRadius$$

$$if\ PolarX > 0$$

$$then\ PolarTheta = \arctan\left(\frac{PolarY}{PolarX}\right) + \pi$$

$$else\ if\ (PolarX < 0)\&(PolarY > 0)$$

$$then\ PolarTheta = \arctan\left(\frac{PolarY}{PolarX}\right) + 2\pi$$

$$else\ if\ (PolarX < 0)\&(PolarY \le 0)$$

$$then\ PolarTheta = \arctan\left(\frac{PolarY}{PolarX}\right)$$

$$else\ if\ (PolarX = 0)\&(PolarY > 0)$$

$$then\ PolarTheta = \frac{3\pi}{2}$$

$$else\ if\ (PolarX = 0)\&(PolarY < 0)$$

$$then\ PolarTheta = \frac{\pi}{2}$$

Although the intersection point is known, a check must be made to ensure that it is within the viewable angle of the sensor (see Table 5). If it is not within the arc of the sensor, then the point of intersection is actually on the opposite side of the robot from the sensor. Because equations deal with lines and not line segments, this check must be made, since the line continues on both sides of the robot. In the case that the sensor cannot see the intersection point, the distance is set to the large constant *ReallyLongDistance*. Figure 4 shows an example of a situation in which *ReallyLongDistance* will be returned as the sensor reading.

Table 5. Checking whether the intersection is within the arc of the sensor.

$$ArcCheckMax = RobotOrientation + SensorTheta + (SensorArc/2)$$

$$ArcCheckMin = RobotOrientation + SensorTheta - (SensorArc/2)$$

$$if\ (PolarTheta \le ArcCheckMax)\&(PolarTheta \ge ArcCheckMin)$$

$$then\ distance = PolarR$$

$$else\ distance = ReallyLongDistance$$

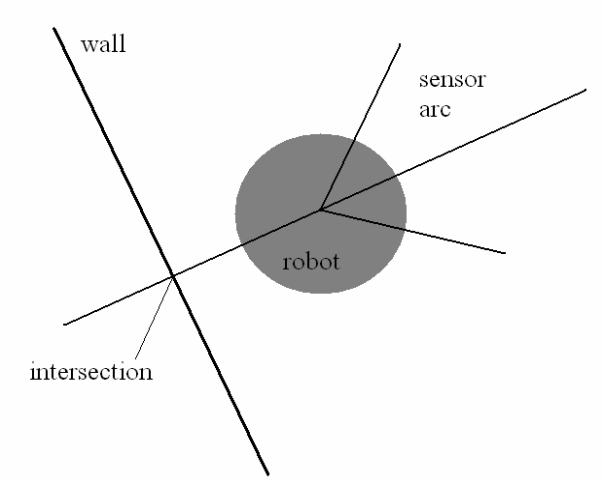

Figure 4. An example that will result in a sensor reading of *ReallyLongDistance* due to the intersection being outside of the sensor's viewing arc.

With the distance known, or set to *ReallyLongDistance* if there is no visible wall, that value is returned back to *CalculateSonar* and compared to each of the other distances within the smaller arc. Only the shortest of these distances is kept as the reading from that smaller arc. This is because the sonar sensors are unable to see through walls, and multiple different readings indicate that there is another wall past the closest one that could be seen if that closest wall was not there. This shortest distance is put into the *MinSegmentDistance* for the smaller arc.

Once each of the smaller arcs has a *MinSegmentDistane* set, they are all compared, and the shortest is set into *MinDistance*. This is the distance to the closest wall and is what the sensor actually sees. Only the closest of these readings is needed (Figure 5).

Once the reading of the sonar sensor is known, the radius of the robot is subtracted, so the sensor reads the distance from the edge of the robot, not from the center. Ten percent noise is then factored into the sensor reading.

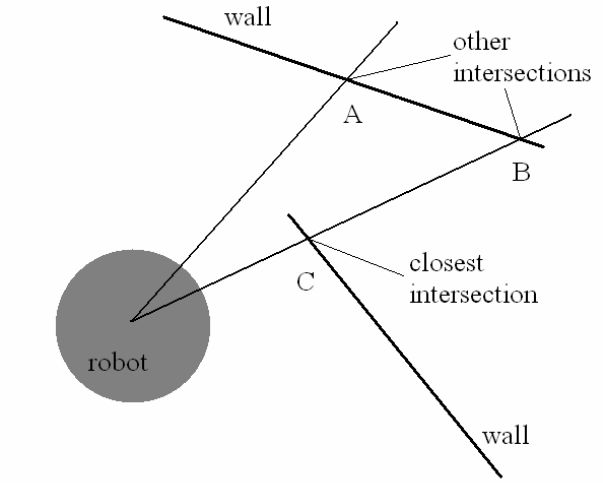

Figure 5. Only the closest wall will matter. Intersection B will be removed during *LineIntersectDistance* because intersection C is closer on that direction line, making C the *MinSegmentDistance* for that smaller arc. Since intersection C is closer than intersection A, C will also become the *MinDistance* for the entire sensor reading.

Once the sensor reading is calculated, it must be encoded into a set of intervals. *SensorEncodeSonar* takes the readings from all of the sonar sensors and returns encoded interval values. Koolio uses an array of eight sonar sensors. However, as explained below, in order to make the size of the Q-table more manageable, some of the sensors are combined. Sonar sensors 1 and 2, located 2π / 9 (40 degrees) and 4π / 9 (80 degrees) on the diagonal front-left of Koolio, have their values averaged into a single value before encoding. Likewise, Sonar sensors 5 and 6, located at the same angles on the diagonal front-right of Koolio, are also combined into a single value.

Within *SensorEncodeSonar*, the values are compared to a series of constants for intervals. A sonar sensor can return a value of *Bump*, *Close*, *Medium*, or *Far*, according to the values in Table 6.

Table 6. Range values used for Koolio's sonar sensor encoding.

Distance Classification	Range
Bump	< 6 inches
Close	6 - 24 inches
Medium	24 - 60 inches
Far	> 60 inches

If the reading is within six inches of the robot, it is returned as *Bump*. If the reading is between six inches and two feet, it is classified as *Close*. If it is between two and five feet, it is classified as *Medium*. Any sensor readings beyond five feet from the robot are considered *Far*. All of these values can be changed for sonar sensors with different specifics.

5.4.3.3 Cameras

Like the sonar sensors, the cameras are also defined by their location on the robot in relation to the front. These angles are listed in *CameraSensors*. More cameras can be added and their order can be rearranged, but that must also be done in all uses of the cameras. The viewing arc of the camera is 45 degrees, defined in the constant *CameraArc*.

The cameras on Koolio return values through external code that returns distance value in inches. If cameras or drivers are used which return distance values in some other format, then an additional function will be required to convert real distance into the reading that the sensors actually output.

Unlike the sonar sensors, which have the single function of reading distance to the walls, the cameras perform two different readings and are each treated in the code as two sensors. Each camera returns a distance to the Goal markers and to the End of the World markers. For Koolio, these markers are large colored squares (see Figure 6). In the simulator, these markers are indicated to the user as colored dots (see Figure 7).

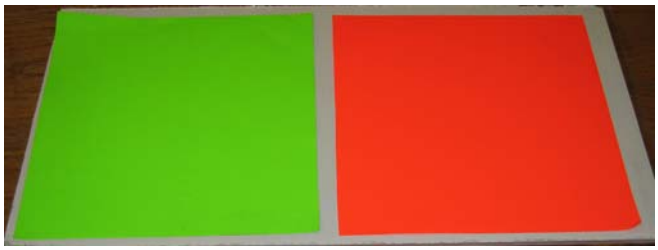

Figure 6. The Goal (left) and End of the World (right) markers used in the real environment.

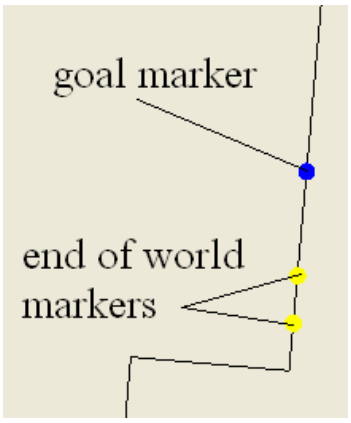

Figure 7. Goal and End of the World markers in the simulator.

Like the sonar sensors, each camera is read individually. Unlike the sonar sensors, since each camera is checking for two things, there are two functions. The distance to the Goal point is checked in *CalculateCameraToGoal* and the distances to the End of the World points are calculated in *CalculateCameraToEndOfWorld*. The two functions are similar, but different enough to require distinct functions. The simulator allows for only a single Goal point. If more than one Goal point is desired, it must be added and *CalculateCameraToGoal* must be altered to allow for multiple Goal points.

The main conceptual difference between the camera reading functions and the sonar reading function is that, while a sonar sensor will always see a wall, the cameras may not see any of the

Goal or End of the World points. The camera functions have a finite list of points to check for distances and must also ensure that there is no wall between the robot and these marker points.

The first step in *CalculateCameraToGoal* is to convert the line between the robot and the Goal point into polar coordinates in relation to the robot's current location. After the polar coordinates of this line are obtained, a check must be made to ensure that this line is within the viewing angle of the camera. The polar coordinate angle is compared to the outer edges of the viewing arc. If that angle is in between the two arc edges, then the Goal point is within the viewing angle and the Boolean variable *CanSeeGoal* is set to true. Otherwise, the Goal point is outside of the viewing arc, and the distance is set to *ReallyLongDistance* and *CanSeeGoal* becomes false.

If the Goal point is inside the camera's viewing arc, then another check must be made to ensure that there are no walls in between the robot and the Goal point. If *CanSeeGoal* is false, this step is bypassed, since it would be a redundant check. The formula for the line between the robot and the Goal point is calculated using Cartesian coordinates. This line is then compared to each of the walls in the arena. The distance from the robot to each wall is calculated using *LineIntersectDistance*, the same function used for determining the readings of the sonar sensors. If this distance is less than the distance to the Goal point, then the line of which the wall is a segment is between the robot and the Goal. Since lines are infinite but the walls are only segments, the endpoints of the wall must be checked to determine if the intersection point lies on the line segment that composes the wall (see Figure 8). If the x-coordinate of the intersection point is between the x-coordinates of the wall endpoints and the y-coordinate of the intersection point is between the y-coordinates of the wall endpoints, then the intersection point lies on the wall. In this case, *CanSeeGoal* is set to false and no more walls need to be checked. Otherwise, the rest of the walls are checked.

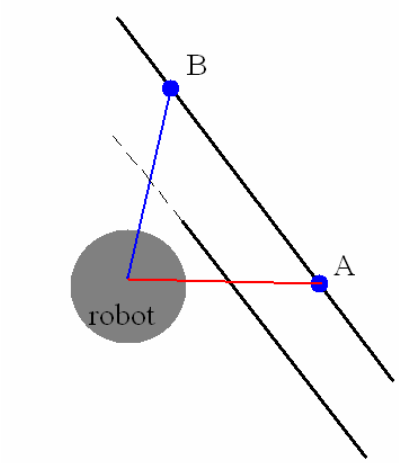

Figure 8. Two examples of line of sight to a Goal point. Point A is blocked by the wall. The full line that contains the wall lies between the robot and point B, but the point is considered visible after checking the wall endpoints.

Ten percent noise is then inserted into the distance. A final check is performed, to ensure that the goal point is within the maximum distance at which the camera can detect. If the distance is greater than *MaxCameraView*, a constant holding this maximum distance, then *CanSeeGoal* is set to false.

If *CanSeeGoal* is false at the end of the function, the distance is set to *ReallyLongDistance*. Finally, *CanSeeGoal* is assigned to the camera's global settings and the distance is returned.

Like the sonar sensors, these distance readings must then be encoded into a set of intervals. *SensorEncodeCamera* takes the readings from all the cameras and returns encoded interval values. Within *SensorEncodeCamera*, the values are compared to a series of constant intervals. A camera can return a value of *Close*, *Medium*, or *Far*, according to the values in Table 7.

Table 7. Range values used for Koolio's camera encoding.

Distance Classification	Range
Close	< 36 inches
Medium	36 - 96 inches
Far	> 96 inches

If the reading is within three feet of the robot, it is classified as *Close*. If the reading is between three and eight feet, it is classified as *Medium*. If the reading is more than eight feet from the robot, it is considered *Far*. These values can be changed for cameras and drivers with different minimum and maximum identification ranges.

Calculating the distance to an End of the World marker is similar to calculating the Goal marker distance. However, since there is only one Goal marker and several End of the World markers, the method is slightly different. *CalculateCameraToEndOfWorld* contains a loop which checks the distance from the robot to each of the End of the World points and returns the distance to the closest one. After checking the distances to each End of the World marker using the same method as in *CalculateCameraToGoal*, the distances are stored in an array. Once all End of the World distances have been calculated, the shortest one is chosen as the sensor reading. Noise is added and the reading encoded using *SensorEncodeCamera*.

5.4.4 Movement

The position of the robot in the simulated arena environment is defined by the location of its center, *RobotLocation*, and the direction it is facing, *RobotOrientation*. These are changed when the robot moves in the *PerformAction* function, which takes an action choice as input. There are five possible action choices, though more can be added if desired. The action choices are *Action_Stop*, *Action_Rotate_Left*, *Action_Rotate_Right*, *Action_Forward*, and *Action_Reverse*. For the sake of simplicity, the robot can only move forward or back, wait, or rotate. If an action that incorporates both movement and turning is desired, it can be added to this function and to the list of actions in the constant list.

If *Action_Stop* is chosen, nothing is changed. If *Action_Rotate_Left* is chosen, the global constant *RotateActionArc* is added to *RobotOrientation*. If *Action_Rotate_Right* is chosen, *RotateActionArc* is subtracted from *RobotOrientation*. If *Action_Forward* is chosen, the location is moved by the global constant *MoveActionDistance* in the direction of the current orientation. If *Action_Reverse* is

chosen, the location is moved *MoveActionDistance* in the direction opposite of the current orientation.

After the movement is made, the simulator must check to make sure that the robot did not pass through the walls of the arena, since the real environment is made of solid walls. The *RobotContacting* function detects whether or not the robot has passed through an arena wall when performing a movement action and, if it has, places the robot back inside the arena tangent to the wall. This function is only meant to work with round robots. If the robot is a shape other than a circle, or cannot at least be represented roughly by a circle, the calculations in *RobotContacting* may not work. This is an acceptable limitation, however, as most robots capable of this sort of movement are round or can be approximated by a circle. This function also assumes that there are no tight spaces in the arena. In other words, there are no locations the robot can be where it touches more than two walls. This is an acceptable assumption, since the robot should never try to move into tight spaces, if they did exist, and should not be expected to squeeze into areas that are exactly as wide as, or even narrower than, its diameter.

The robot can still act strangely at some of the convex corners. This is because it attempts to cut across the corner when moving straight toward it and is put back to where it should be. Therefore, if the robot keeps attempting to cut across the corner, it will make no progress. This only happens when the robot is attempting to hug the wall, remaining in tangent contact with it. It is a side effect of the code, as it only happens when the robot is moving counter to the order of the walls. For example, if the robot I hugging the wall between *SegmentEndpoint* 4 and 3, approaching the corner formed with the wall between *SegmentEndpoint* 3 and 2, this effect may occur. However, it does not happen when the robot approaches the same corner from the other direction. If the robot is not hugging the wall and just passes near the corner, this situation never arises.

The *RobotContacting* function uses three Boolean variables as tags to keep track of the status of the robot's interaction with the walls of the arena: *testval*, *foundintersection*, and *morecontacts*. All three are initialized to false. This function also uses a two-dimensional array, *whichwalls*, which stores the starting segment point number of a wall that is intersected by the robot. Both of them are initialized to –1.

The first step in determining whether the robot is intersecting a wall is to go through each of the walls and check for intersections. For each wall, the equation of that wall is found. To check for intersections, the equation of the wall and the equation of the circle that makes up the perimeter of the robot are set equal to each other. The formulas are shown in Table 8.

If the discriminant is negative, the robot is not touching the wall. If it is 0, the robot is tangent to the wall, touching at one point. If it is positive, the robot intersects the wall at two points. If the discriminant is negative and the robot is not touching that wall, nothing is done. Otherwise, the quadratic formula is performed to find the point or points of intersection, as shown in Table 9.

Table 8. Finding the equation describing the intersection(s) between a line and a circle.

Equation of a circle:

$$(x - LocationX)^2 + (y - LocationY)^2 = radius^2$$

Equation of a line: $y = Slope * x + Intercept$

Expand the circle equation:

$$x^2 - 2x * LocationX + LocationX^2 + y^2$$
$$- 2y * LocationY + LocationY^2 - radius^2 = 0$$

Substitute line equation into circle equation:

$$x^2 - 2x * LocationX + LocationX^2 +$$
$$(Slope * x + Intercept)^2$$
$$- 2(Slope * x + Intercept) * LocationY$$
$$+ LocationY^2 - radius^2 = 0$$

Simplify:

$$(Slope^2 + 1)x^2 + 2\begin{pmatrix} Slope * (Intercept - LocationY) \\ - LocationX \end{pmatrix} x$$
$$+ \begin{pmatrix} LocationX^2 + LocationY^2 + Intercept^2 \\ - radius^2 - 2 * Intercept * LocationY \end{pmatrix} = 0$$

This is now a quadratic equation of the form $ax^2 + bx + c = 0$

Where:

$$quadraticA = Slope^2 + 1$$

$$quadraticB = 2\begin{pmatrix} Slope * (Intercept - LocationY) \\ - LocationX \end{pmatrix}$$

$$quadraticC = LocationX^2 + LocationY^2 +$$
$$Intercept^2 - radius^2 - 2 * Intercept * LocationY$$

$$discriminant$$
$$= quadraticB^2 - 4 * quadraticA * quadraticC$$

Table 9. Finding the point(s) of intersection using the quadratic formula.

$$IntersectX1 = \frac{-qB + \sqrt{discr}}{2 * qA}$$

$$IntersectX2 = \frac{-qB - \sqrt{discr}}{2 * qA}$$

$$IntersectY1 = Slope*IntersectX1 + Intercept$$

$$IntersectY2 = Slope*InterxectX2 + Intercept$$

In the rare case that there is actually only one intersection, both of these two intersection points will be identical. As in the case with the sonar and camera sensors, the intersection points may not be on the line segment that contains the wall. The check formulas are shown in Table 10

Table 10. Check formulas for wall intersection points.

$$Check1 = \frac{IntX1 - WallEndX1}{WallEndX2 - WallEndX1}$$

$$Check2 = \frac{IntX2 - WallEndX1}{WallEndX2 - WallEndX1}$$

If the first endpoint is farther to the left than the second and the x-coordinate of the intersection is between them, the check becomes a positive over a positive for a positive number. If the intersection is not between them, the check becomes negative over positive for a negative number. If the second endpoint is farther to the left than the first and the x-coordinate of the intersection is between them, the check becomes a negative over a negative for a positive number. If the intersection is not between them, it becomes a positive over a negative for a negative number.

In addition, if a point on a line is in between two points, then the distance from one endpoint to that middle point should be less than the distance between the endpoints. If the x-coordinate of the intersection point is in between the two wall endpoints, then, the check number should be less than or equal to 1.

Only the x-coordinates of the intersection point are checked, since the intersection point is in between the y-coordinates of the wall endpoints if and only if it is also in between the x-coordinates of the wall endpoints. Also, since these check numbers are only used for their values relative to 0 and 1 and not their actual numbers, the y-coordinates do not need to be checked.

Given these properties for the check numbers, at least one of the intersection points are valid if and only if *Check1* is less than or equal to 1 and either *Check1* or *Check2* are greater than or equal to 0. While the extra comparison to 1 is not necessary, it is still valid. If these conditions are met and there is a valid intersection point, *testval* is set to true.

If *testval* is true and *foundintersection* is false, the average of the two intersection points is taken. The *atan2* function is used to find the direction of the line from the center of the robot to the average of the intersection points, which is stored in *Direction*. If no intersection has been found to this point, then *whichwalls*[0] will still have its initial value of –1. If this is the case, it is instead set to the identifying number of the wall that is being intersected. After this, *foundintersection* is set to true and *testval* is set to false.

If both *testval* and *foundintersection* are true, then the robot is intersecting more than one wall. The *whichwalls* array is checked to make sure that there is a value stored in *whichwalls*[0] and *whichwalls*[1] is still at its initial –1 value, then *whichwalls*[1] is set to the identifying number of the second wall. The Boolean *morecontacts* is set to true and the loop that checks all walls is broken. This means that if the robot is touching three walls at once, it may become stuck. However, such an arena is unlikely to be made and can be avoided by not allowing arenas with any tight spaces where three or more walls make an alcove that the robot would not be able to physically enter.

The loop to check each of the walls ends when either all the walls have been checked with no more than one intersection or has been ended prematurely by the existence of two intersections.

If *foundintersection* is true and *morecontacts* is false, then a single intersection was found. The distance that the edge of the robot overhangs the wall, *Overlap*, is calculated by subtracting the distance between the robot's center and the intersection point from the robot's radius. Once *Overlap* is found, the robot's location is moved that distance away from the wall in the direction of *Direction*.

If both *foundintersection* and *morecontacts* are true, then there were two intersections found. First, the *Overlap* is calculated for the first wall that was encountered and the robot is moved away from it as above. The robot's position is then checked in relation to the second wall that was encountered, performing a single iteration of the intersection-finding loop on that wall. The new intersection, if any, is calculated, and the robot is moved away from the second wall by a newly-calculated *Overlap* value.

After all the location corrections, if any, are performed, the function ends, returning *foundintersection* to indicate whether one or more intersections were found.

After the simulator ran for several thousand episodes, it was observed that there was a small chance the robot could become stuck against a corner under certain circumstances. To avoid that, another check was added. If the user-indicated *Auto_Rescue_Toggle* is turned on, the episode has run for more than MaxStepsBeforeRescue steps, and the sensors indicate that the robot is in a *Bump* condition, then the *Rescue_Robot* function has a *Rescue_Chance* chance of being called. This function forces the robot to execute a random number (between 0 and the constant *Rescue_Max_Backup*) of reverse actions. After each reverse action, *RobotContacting* is called to ensure that it cannot be rescued through a wall to the outside of the arena. After the reverse actions are performed, the robot is spun to face a random direction. After this rescue, all of the sensors are recalculated and *Has_Been_Rescued* is set to true to indicate that a rescue occurred in the current episode.

5.4.5 Displays and Controls

Aside from the arena walls, the Goal and End of the World markers, and the robot's position and direction, the simulator also has several optional displays that can be toggled on or off by the user.

The *Show Sensors* button toggles extra information on the robot itself. When it is activated and the robot receives a Bump signal from one of the sonar sensors, the robot is drawn pink instead of red. When the robot has run off the End of the World, it is displayed in purple. When the robot reaches the goal, it is displayed in black. This toggle also displays lines indicating the sensors, with black lines for the sonar sensors, blue lines for cameras seeing the Goal, and yellow lines for cameras seeing the End of the World. Figure 9 shows some examples of the sensor displays.

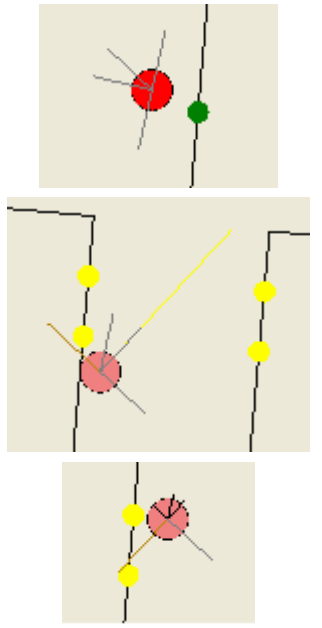

Figure 9. Sensor displays. Top: The robot near the start area. Center: The robot in a Bump state seeing the End of the World markers in the front camera (Medium) and the left camera (Close). Bottom: The robot in a Bump state seeing an End of the World marker in the left camera (Close).

The encoded readings for the sensors can also be displayed (see Figure 10). This display toggles with the *Show HUD* button. When turned on, a list of all the sensors appears with readings of *Close*, *Medium*, or *Far*, as well as a field that indicates of any of the sonar sensors read a Bump state.

HUD	BUMP
Front Sonar:	Close
Front Left Sonar:	Close
Front Right Sonar:	Far
Left Sonar:	Close
Right Sonar:	Close
Back Sonar:	Medium
Front Goal Camera:	Far
Left Goal Camera:	Far
Right Goal Camera:	Far
Front End of World Camera:	Close
Left End of World Camera:	Far
Right End of World Camera:	Far

Figure 10. The sensor HUD.

A statistics display for the simulation can be toggled with the *Show Stats* button (see Figure 11). This shows the reward, number of steps, and number of bumps for the current episode, as well as the total number of episodes, the total number which ended by reaching the goal, and the average reward, number of steps, and number of bumps per episode.

Simulation Statistics	
Current Episode Reward:	-25000
Current Episode Steps:	105
Current Episode Bumps:	25
Number of Episodes:	67689
Total Number of Goals:	21708
Average Reward per Episode:	-5289046.46318397
Average Steps per Episode:	555.167370878147
Average Bumps per Episode:	525.548162155774

Figure 11. Simulation Statistics.

Two other special displays are shown, indicating when the auto rescue function and the clockless timer have been turned on. A tag also displays if the robot had to be rescued in the current episode.

Besides the buttons to turn the display toggles and function toggles on and off, there are also buttons to begin and pause the simulation, to manually force *Rescue_Robot* to run, and to exit the simulation (see Figure 12). If the normal *Exit* button is pressed, the simulation closes and factors the current episode into the Q-table. If the *Exit Without Save* button is pressed, the simulation closes and discards all data from the current unfinished episode. There is also a slider to control the speed of the clock.

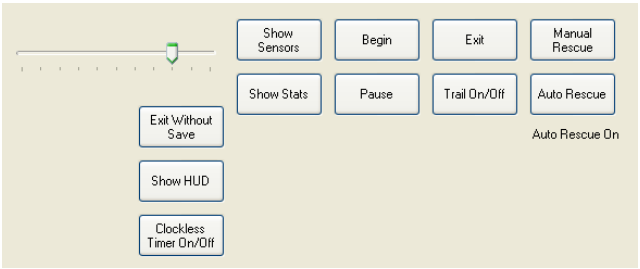

Figure 12. The control panel.

5.4.6 Learning Algorithm

The Q-learning formula in the simulator involves five steps: choose an action, find the reward for that action choice, read the next state, update the Q-table, and move to the next state.

The first step is to choose an action. In order for Q-learning to allow for exploration, there must be a ε chance of selecting an action at random. A random number is generated. If that number is greater than 1-ε, a second random number between 0 and *NumActions*, the number of different possible actions, is generated. This random number determines the action choice, where each choice is an equally likely outcome.

Otherwise, the greedy algorithm is performed. The Q-table values for the current state and all possible action choices are read, and the action choice that yields the highest reward is selected. The robot's current position and direction are stored in *OldLocation* and *OldOrientation*, and *PerformAction* is called with the selected action choice.

After the action is performed, the new reward for that action choice must be determined. There are seven possible rewards: Goal, End of World, Bump, Stop, Reverse, Turn, and Forward, in order of priority. If the action chosen was stop and the front camera reads a Close distance to the goal, then the robot receives the Goal reward. If the robot has run off the end of the world,

then it receives the End of World reward. If it is in a bump state, it receives the Bump reward. If none of these three conditions are present, the reward depends solely on the last action taken.

Once the reward has been chosen, the next state must be read. Since the state is defined by the sensors, all of the compass, sonar, and camera sensors are read and encoded.

The Q-table must then be updated using the Q-formula (see Table 11). Although the sensor readings of the next state are known, the action choice is not yet known. However, all that is needed for the Q-formula is the value of the highest possible reward. The rewards for all possible action choices for the current sensor state are read, and the highest saved. The reward for the current state is also read, and the Q-formula calculates the new value for the reward of the current state.

Table 11. The Q-Formula. α and γ are global constants. α is the step size parameter and γ is the discounting factor.

$$NewQ = CurrentQ + \alpha * \left(reward + \left(\gamma * \max\left(NextQ\right)\right) - CurrentQ\right)$$

The final step is to move to the next state.

5.4.7 Q-Table Implementation

Initially, the simulator was designed with fifteen sensors: one compass, six cameras (three each for the goal and the end of the world), and eight sonar sensors. The compass could contain four values (North, South, East, and West). Each of the cameras could contain four values (Close, Medium, Far, and Very Far). Each of the sonar sensors could contain five values (Bump, Close, Medium, Far, and Very Far). However, the large number of possible readings let to a combinatorial explosion. With seven sensors containing four possible values each and eight sensors containing five possible values each, there were $6.4*10^9$ possible combinations. Each of these sensor states can have five different action choices, making a Q-table with $3.2*10^{10}$ different states. Each state requires a 64-bit double type for its value, resulting in a Q-table size of 238.4 GB. This file size is obviously much too large to be of any use.

In order to fix this, the number of states had to be reduced. Since the compass is an erratic and unreliable sensor, it was dropped entirely. Some of the sonar sensors are combined as well. Sonar sensors 1 and 2 are combined into a single reading, and sensors 5 and 6 are also combined into a single reading. Finally, the Very Far value was removed from all sonar and camera sensors.

With these changes, there were six camera sensors with three values each and six sonar sensors with four values each. This made for less than $1.5*10^7$ different states in the Q-table. With this smaller number of states, the Q-table size was reduced to approximately 113.9 MB, more than two thousand times smaller and a much more manageable file size.

This file size is a maximum and will only be reached if the Q-table is completely full. However, because of the nature of Q-learning, the thirteen-dimensional matrix will be extremely sparse. For example, a situation will never arise when all three cameras read the Goal point as being at a Close distance. Because the matrix is sparse, the use of a data structure such as a large array would result in a large amount of wasted space. In order to avoid that, a different data structure is required.

It was decided to implement the Q-table using the Dictionary data structure, a modified hash table quick consists of keywords and values. The function *GenerateQKeyword* takes the sensor readings and the action choice and creates an integer keyword unique to that combination. This is done by converting a list of the sensor values into binary. The camera goal readings come first, followed by the end of the world readings, the sonar readings, and the action taken. This gives six cameras with three readings each, which requires two bits each to encode. Six sonar sensors with four readings each also require two bits each to encode. Five different action choices require three bits to encode. To make the keyword, each component is multiplied by a different power of two and added together to make a single integer (see Table 12).

Table 12. The keyword-generating function.

$$\begin{aligned}\text{Qkeyword} = {} & \text{GoalCam0}*2^{25} + \text{GoalCam1}*2^{23} + \text{GoalCam2}*2^{21} + \\ & \text{EoWCam0}*2^{19} + \text{EoWCam1}*2^{17} + \text{EoWCam2}*2^{15} + \\ & \text{Sonar0}*2^{13} + \text{Sonar1-2}*2^{11} + \text{Sonar3}*2^{9} + \text{Sonar4}*2^{7} + \\ & \text{Sonar5-6}*2^{5} + \text{Sonar7}*2^{3} + \text{Action}\end{aligned}$$

5.4.8 Runtime

When the simulator is first started, several initializations are made in the *Arena_Init* function. First, all endpoints of the arena walls, the starting location, and the location of all goal and end of the world markers are rotated five degrees. This prevents the arena walls from being vertical lines, which would result in infinite slopes in many of the calculations in the code.

If the Q-table file and the Q-table backup file don't exist in the current directory, empty files are created. The Q-table is then read into a global variable. At the end of the initialization function, the robot's starting orientation is set to a random angle to ensure the Exploring Starts condition required by Q-learning.

The majority of the runtime of the simulator takes place in *RobotTimer_Tick*, which is called with each tick of the timer. First, the Q-learning process is begun when an action is chosen and performed. After the action is performed, the *RobotContacting* function is called to ensure that it did not pass through any arena walls. The reward for the chosen action is then found.

Once the reward for the action is found, it is added to the current reward tally for the episode and the number of steps in the episode is increased by one. If the robot is in a bump state, the count of wall hits in the current episode is increased by one. After rewards are assigned, if the robot has reached the goal or passed the end of the world, *SimulationReset* is called to end the episode, write the Q-table to its file, and start a new episode.

All the sensors are then read to find the current state, and the *RobotAtEndOfWorld* function determines if the robot has passed the end of the world. After this, the new reward is calculated with the Q-formula and is added to the Q-table.

A check is then made to see if the robot has to be rescued, if the *Auto_Rescue_Toggle* is turned on. The state is advanced to the next state and the graphical display is refreshed.

The final check is for the clockless timer. If this toggle is activated, then the contents of the timer loop are executed *Clockless_Loop_Count* times immediately. *Clockless_Loop_Count* is a global constant initialized to 100, though it can be changed for faster computers. The shortest interval between timer ticks is one millisecond, so a fast computer can run 100 steps per millisecond when this function is turned on.

5.4.9 Future Considerations

The main consideration for future work on the simulator is to add a floating control panel and data output. With the control panel and displays on a separate window, the graphical arena can be more easily scrolled to view any part of it without losing access to the user controls.

6. REFERENCES

[1] Aljibury, H. (2001). "Improving the Performance of Q-Learning with Locally Weighted Regression." Masters Thesis, University of Florida.

[2] Mekatronix. http://www.mekatronix.com

[3] Sara Keen, Lavi Zamstein, Wenxing Ye, Blake Sutton Gene Shokes, Gorang Gandhi, Eric M. Schwartz, A. Antonio Arroyo. *Koolio: An Autonomous Refrigerator Robot.* Proceedings of the 2006 Florida Conference on Recent Advances in Robotics, May 25-26, 2006, FIU, Miami, Florida.

[4] Su, M., Huang, D., Chou, C., and Hsieh, C. *A Reinforcement Learning Approach to Robot Navigation*. Proceedings of the 2004 IEEE International Conference on Networking, Sensing & Control, Taipei, Taiwan, 2004.