

BROWN UNIVERSITY
Department of Computer Science
Master's Project
CS-95-M7

“Braitenberg Vehicles in a Virtual Environment”
by
Laura Ann Dorival Paglione

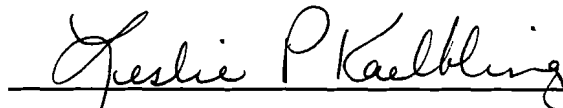
Braitenberg Vehicles in a Virtual Environment

Laura Ann Dorival Paglione

Department of Computer Science
Brown University

Submitted in partial fulfillment of the requirements for the
Degree of Master of Science in the Department of Computer
Science at Brown University

May 1995



Leslie Pack Kaelbling
Advisor

Braitenberg Vehicles in a Virtual Environment

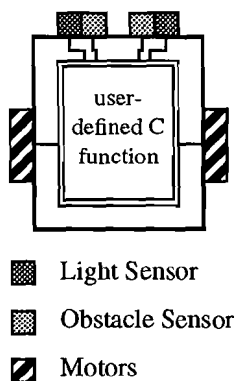
By Laura Dorival Paglione

This is a paper to report the work done under Leslie Pack Kaelbling at Brown University during CS0298 Reading and Research from Spring 1994 through Spring 1995.

Valentino Braitenberg, in his book Vehicles: Experiments in Synthetic Psychology, describes a series of hypothetical, self-operating machines which he calls "vehicles." All of the vehicles have a very simple internal structure, and are discussed as though they exhibit some sort of civilized "behavior." His vehicle's behaviors are borne out of a direct connection of input sensors to output actuators. This connection allows the vehicles to be able to directly translate what they sense into how they react. In this paper I discuss a simulator for Braitenberg-type vehicles in a virtual environment.

Introduction

All of Braitenberg's early vehicles have one or two motors (actuators) and one or more sensors. He connects the sensors to the motors directly with either inhibitory or excitatory connections. Each sensor can be connected to each actuator in one of three ways: straight, crossed, or combination. In a straight connection a sensor would be connected to the actuator on the same side of the vehicle. A crossed connection is made when the sensor is attached to the actuator on the opposite side of the vehicle. Combination connections are a combination of straight and crossed connections.



The simulator described in this paper allows vehicles (called Agents in this paper) with a fixed set of sensors and actuators to simulate different sensor-actuator combinations. The Agents are configured to have two light sensors, two obstacle sensors, and two actuators. The Agents are symmetrical with two different sensors and one actuator on each side (see the figure to the left). The Agents' movements are determined from a C function written by the user. This function specifies all of the sensor-actuator connections through equations.

The virtual world consists of Light Sources, Agents and Walls. Agents move around in this world by sensing the Lights and other objects, and reacting to them according to the user-defined function.

Implement Braitenberg vehicle behavior on an actual vehicle is trivial, because the physics of turning and reacting to a source are inherent in the vehicle's physics of movement. To recreate this behavior in a virtual environment is more difficult because models for physical movement and for sensor behavior must also be created. In addition, a graphical representation of the vehicle and its environment must be developed.

The models for the physical movement and sensor behavior are based on assumptions which will be discussed in the body of this paper. The graphical representation is developed using graphics libraries created by the Graphics Group of Brown University. Although simplified by the use of these libraries, the graphical representation presents some challenges. Movement in the virtual world needs to be specified through an alteration of transformation matrices which correspond to the changes of the physical movement model for the vehicle.

This paper describes the methods used to create this simulator. In addition, it describes possible uses, future work, and the materials and libraries needed to make the simulator work.

Materials

This project is designed to be easily portable to other systems. As a result, there is a limitation on the materials that could be used.

Materials and Libraries

An accelerated SPARC 10 with a Leo graphics accelerator card and 32 MB of memory is used for this project. This machine allows for real-time, three-dimensional rendering of the vehicle and its environment, despite its computational complexity.

Two packages, WAV and OD, developed by the Brown Graphics Group are used for the graphical rendering for this project. They are chosen primarily because of their portability to other platforms. WAV is a class which encapsulates windows and views for 3D graphics. OD is a class of objects that can be drawn in WAV windows. When compiled on a SPARC station, these packages access Silicon Graphic International's XGL library, a high-end 3D graphical package.

Other libraries that are used include Motif for the user interface and windows, and various matrix, camera, and lighting packages developed by the Brown Graphics Group.

Methods

The structure of the program has three parts: the Environment, the Agents, and the Program Interface. These areas are described in the section "Program Structure". When a

simulation is started, the Agents move through a series of incremental steps. These steps depend on the sensor and physical movement models, and represent the bulk of work for this project. These steps are described in detail in the section “The Steps of the Incremental Move.”

Program Structure

The Environment. The virtual environment consists of Lights, Walls and Agents, all of which are also considered to be obstacles. Agents are discussed at length in the next section, but it is important to note that they can sense lights and obstacles. Lights can be used for attracting or repulsing Agents, and Walls are primarily used as static obstacles for an Agent to avoid.

The application maintains lists for each of the three items that exist in the world. In addition, it maintains a list for a special type of object called a Point Obstacle. This object is used to simplify the way that the agent detects obstacles by modeling each potential obstacle as one or more spheres that occupy the same area as the item. All of these point obstacles are kept in a list which the Agent uses to determine if its obstacle sensors detect anything.

Below are the internal structures of the Application, Light and Obstacle classes. The Agent class is described in the next section.

FIGURE 1. Application Class

```
class IAVRApplication {  
  private:  
    AgentList agent_list_;  
    LightList light_list_;  
    WallList wall_list_;  
    PtObsList pt_obstacle_list_;  
    [...]   
}
```

The internal structure of the Application class. The application manages lists of agents, lights, and walls. In addition, it maintains a list of point obstacles.

FIGURE 2. Light Class

```
class IAVRLight {  
    private:  
        Location    light_location_;  
        ODlist*     light_picture_;  
        int         drawn_;  
        [...]        
}
```

Internal structure of the Light class. The light knows its global location, its graphical representation, if it is drawn, and how to draw itself.

FIGURE 3. Wall Class

```
class IAVRWall {  
    private:  
        Location    wall_location_;  
        ODlist*     wall_picture_;  
        int         drawn_;  
        [...]        
}
```

Internal structure of the Wall class. The light knows its global location, its graphical representation, if it is drawn, and how to draw itself.

The Agents. Agents consist of two light sensors, two obstacle sensors, and two motors. The agent also knows its global position in the world, and the sensors and motors attain their global positions and directions from a relative position from the agent's global position. The agent knows if it has completed its run of the simulation, and if it is currently displayed on the screen.

Figure 4 shows the internal structure of the agent class. Figures 5, 6 and 7 show the internal representation of the motors and the sensors.

FIGURE 4. Agent Class

```

class IAVRAgent {
private:
    Motor        left_motor_;
    Motor        right_motor_;
    LightSensor  left_light_sensor_;
    LightSensor  right_light_sensor_;
    ObsSensor    left_obstacle_sensor_;
    ObsSensor    right_obstacle_sensor_;
    Location     agent_location_;
    Direction    agent_direction_;
    ODlist*      agent_picture_;
    int          is_stopped_;
    int          drawn_;
    [...]
}

```

Internal structure of the Agent class. The Agent has two motors and two light sensors. It knows its global location, its graphical representation, if it is stopped, and if it is drawn.

The Motors of the agent know their global position and direction in the virtual world. They obtain their value from the global user defined function.

FIGURE 5. Motor Class

```

class IAVRMotor {
private:
    Location     motor_location_;
    Direction    motor_direction_;
    int          motor_value_;
    [...]
}

```

Internal structure of the Motor class. The Motor knows its global location (which is determined though a relative offset from its associated agent), and its value. The motor value is an integer between 0 and MAX_MOTOR_VALUE_READING_.

The light and obstacle sensors also know their global position and direction. They can calculate their value based on their proximity to the lights or obstacles in the virtual environment. Each sensor determines its value from a list of things that it senses which is passed to it from the virtual world. For example, when it is time for a light sensor to calculate its value, the Application passes the sensor a list of lights. The sensor determines which lights it can see, and sets its value accordingly. More information about

this process is in the section “The Steps of the Incremental Move.” Figures 6 and 7 show the internal structure of the sensors.

FIGURE 6. Light Sensor Class

```
class IAVRLightSensor {
private:
    Location      light_sensor_location_;
    Direction     light_sensor_direction_;
    int           light_sensor_value_;
    [...]
}
```

Internal structure of the Light Sensor class. The Light Sensor knows its global location (which is determined through a relative offset from its associated agent), and its value. The light sensor value is an integer between 0 and MAX_LIGHT_SENSOR_VALUE_.

FIGURE 7. Obstacle Sensor Class

```
class IAVRObstacleSensor {
private:
    Location      obstacle_sensor_location_;
    Direction     obstacle_sensor_direction_;
    int           obstacle_sensor_value_;
    [...]
}
```

Internal structure of the Obstacle Sensor class. The Obstacle Sensor knows its global location (which is determined through a relative offset from its associated agent), and its value. The Obstacle Sensor value is an integer between 0 and MAX_OBSTACLE_SENSOR_VALUE_.

Movement of the agents is accomplished by a series of incremental moves. Performing movement calculations in discrete steps allow the agent to react to its surroundings while it is moving, and makes the movement of the graphical representation look smooth. On each incremental move seven things happen:

1. The sensor readings for each of the light sensors is determined.
2. The sensor readings for each of the obstacle sensors is determined.
3. The values of all four sensors are stored in a global input structure.
4. The values for the actuators are determined by calling a user defined function IAVR-Propagate(). (This function describes the connections of the sensors to the actuators.)
5. The values for the actuators are stored in a global output structure.

Methods

6. The agent reads the output structure and moves accordingly.
7. Check if the simulation has completed.

Each of these sub items for the incremental move are discussed in the next section: The Steps of an Incremental Move.

The Interface. The interface consists of a number of Motif widgets which allow the user to change the simulation in the virtual environment. The user can start the simulation, quit, and move, remove and add Lights, Agents or Obstacles. The implementation of the interface is not discussed here, but information on how to use the interface can be found in the accompanying paper, Braitenberg Vehicles in a Virtual Environment: Users Guide.

The Steps of the Incremental Move

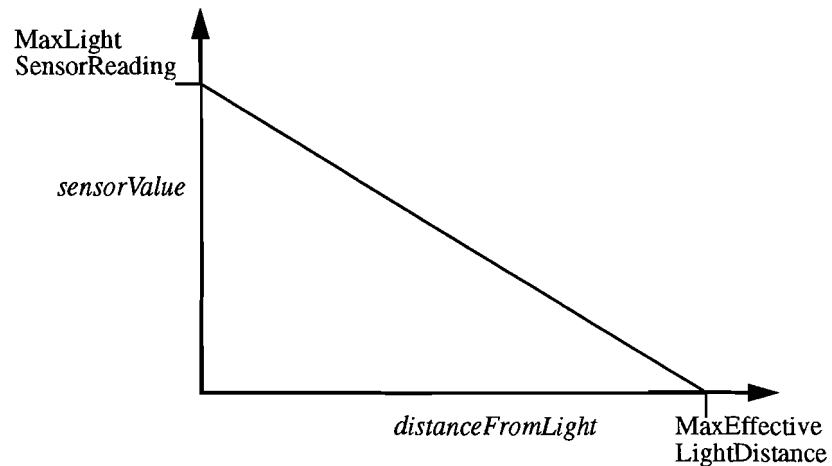
For each step of its incremental move, the agent

1. The sensor readings for each of the light sensors is determined.
2. The sensor readings for each of the obstacle sensors is determined.
3. The values of all four sensors is stored in a global input structure.
4. The values for the actuators are determined by calling a user defined function IAVR-Propagate(). (This function describes the connections of the sensors to the actuators.)
5. The values for the actuators are stored in a global output structure.
6. The agent reads the output structure and moves accordingly.
7. Check if the simulation has completed.

This section of the paper describes each of these items in detail.

1. The sensor readings for each of the light sensors is determined. Each light sensor determines its value based on a list of light sources that it is passed. For each Light, the sensor learns its distance from it using vector algebra, and determines its value from a linear equation based on its distance from the light. Figure 8 describes this equation in detail. In an actual environment, a light could be sufficiently far away that the sensor could no longer detect it. In this program this distance is called the *MaxEffectiveLtDistance*. For distances greater than the maximum, the sensor value is equal to zero.

FIGURE 8. Linear Equation for Light Sensors

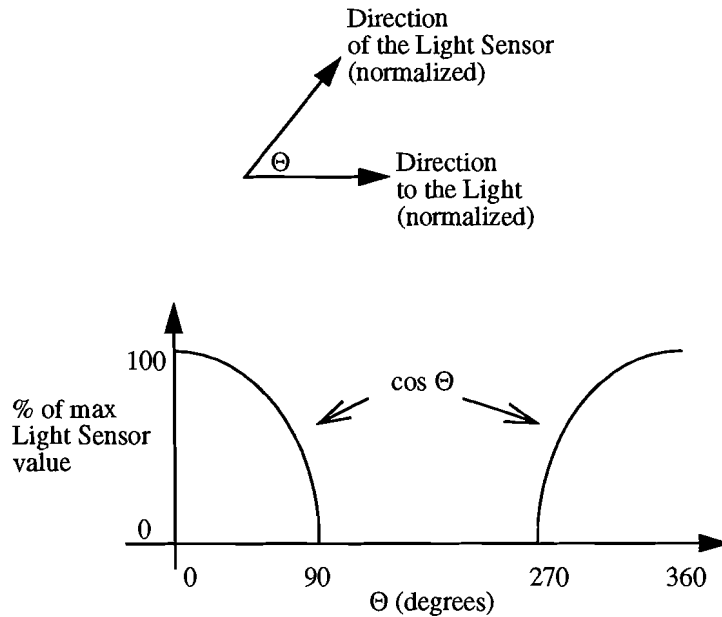


$$\text{sensorValue} = \text{MaxLtSensorReading} \left(1 - \frac{\text{distanceFromLight}}{\text{MaxEffectiveLtDistance}} \right) \quad (\text{EQ 1})$$

The sensor reading is inversely proportional to the distance it is from the light. The light sensor asks the light how far it is away from the light, and uses this value in equation 1 shown above.

Once the value of the light sensor has been determined from its distance from the light, it is adjusted to account for directionality inherent in most light sensors (for example, a light sensor is the most sensitive in the direction in which it is pointing.) This adjustment is accomplished by multiplying the sensor value obtained earlier by the vector-dot-product of the direction the light sensor is pointing, to the direction from the light sensor to the light. This equation is described in further detail in Figure 9.

FIGURE 9. Equation for Directional Light Sensors



$$\text{MaxLightSensor} = \text{LightSensorDirection} \cdot \text{DirectionToLight} \quad (\text{EQ 2})$$

The light sensors on the agent are directional. If the light is directly facing the light (i.e. the angle between the sensor direction and the direction to the light is 0) then the value of the light sensor is equal to 100% of the amount found through equation 1. The percentage of the sensor value diminishes as the angle increases, with a value of 0% if the angle is greater than 90 degrees. A cosine model is used allowing the sensor value to have larger percentage values with a quick drop off in value as the angle approaches 90 degrees.

Another advantage of using cosine to model the sensor value falloff is that it can be calculated by taking the dot product of the normalized sensor direction and direction to light vectors, a computationally easy procedure.

So far I have discussed how the value of the sensor is determined from one light, but, in actuality, a list of lights are passed to the sensor. The light sources have a cumulative effect on the sensors. Therefore, once the sensor value due to each source is determined, they are summed, resulting in the light sensor's net reading. Since each sensor has a maximum possible reading, if the net reading is higher than the maximum, the value for the sensor is set to the maximum.

2. The sensor readings for each of the obstacle sensors is determined. As with the light sensors, each obstacle sensor determines its value from a list a point obstacles that is passed. Calculations similar to those done to determine the light sensor values are performed to determine the obstacle sensor values.

3. The values of all four sensors is stored in the input structure. In the file IAVR-RexInterface.H there are two structures, input and output. These structures are global so that they can be used by all of the classes in the program and by the global function IAVRPropagate(), which is used to determine the values for the motors. Once the values for the sensors are determined, they are copied into the following structure:

```
struct IAVR_Input
{
    int left_light_sensor,    // value between 0 and 100
    int right_light_sensor,   // value between 0 and 100

    int left_obstacle_sensor, // value between 0 and 100
    int right_obstacle_sensor, // value between 0 and 100
};
```

4. The values for the actuators are determined by calling a user defined function IAVRPropagate(). A global function named IAVRPropagate() describes the connections of the four sensors to the two motors. It was made global so that it could be easily exchanged with other like functions. Since this function is usually defined by the user (although there is a default version), it is not discussed here. For further explanation of this function and how to define it, refer to the document, Braitenberg Vehicles in a Virtual Environment: Users Guide.

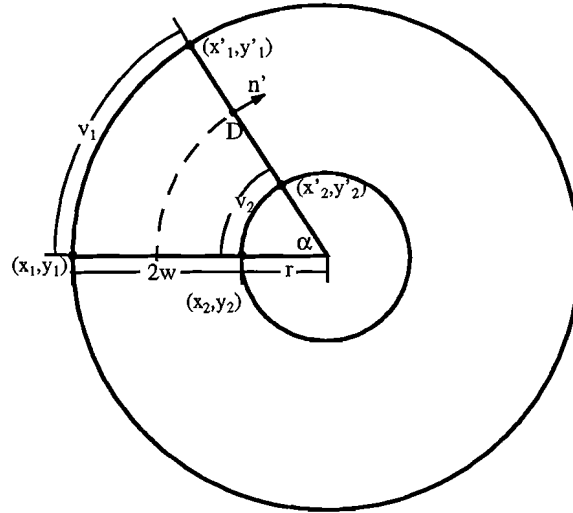
5 The values for the actuators are stored in a global output structure. The function IAVRPropagate() is responsible for copying the motor values that it determines into the following structure which is used by the agent to move in the world:

```
struct IAVR_Output
{
    int left_wheel,          //value between 0 and 100
    int right_wheel,         //value between 0 and 100
};
```

6. The agent reads the output structure and moves accordingly. The Agent reads the values from the output structure and determines its next position. The model used for the calculation of the distance traveled is shown in figure 10.

Figure 10 is example of one possible scenario in which the motor value of the left wheel is higher than that of the right wheel. In this example, the vehicle starts with its left wheel at point $(x1, y1)$, and its right wheel at point $(x2, y2)$. For the incremental step, the left wheel moves faster than the right wheel in a ratio of $v1:v2$ where $v1$ is the value of the left motor, and $v2$ is the value of the right motor. (Note: this assumes that the value of the motor is directly proportional to the distance that that wheel will travel.) This ratio will cause the vehicle to turn in a clockwise circle. By using geometrical properties (described below), the next position for the vehicle (point D) and its final direction (vector n') can be determined in coordinates relative to the initial position of the vehicle. The values of D and n' can then be translated to the global coordinate system.

FIGURE 10. Calculation of the Distance Travelled



Derivation of the equation to determine the distance traveled by the agent. The work in this paper is on a model that moves in only 2 dimensions. A 3D model derivation would be similar, but it would be based on a sphere, not a circle.

Determining the relative ending position for the vehicle is dependent on the assumption that a wheeled vehicle will travel in a circle. In this case, the ratio of the arcs v_1 and v_2 is the same as the ratio of their relative circles, or

$$\frac{v_1}{v_2} = \frac{2\pi r}{2\pi(r + 2w)} \quad (\text{EQ 3})$$

Equation 3 can be used to find the value of r , the radius of the inner circle, or the distance from the point of rotation to the right wheel

$$r = \frac{2wv_2}{v_1 - v_2} \quad (\text{EQ 4})$$

The angle α can also be found using properties of chords.

$$\frac{v_2}{2\pi r} = \frac{\alpha}{2\pi} \quad (\text{EQ 5})$$

$$\alpha = \frac{v_2}{r} \quad (\text{EQ 6})$$

Once r and α are known, the ending positions for each of the wheels can be found

$$x_1' = x_1 + (r + 2w)(1 - \cos \alpha) \quad (\text{EQ 7})$$

$$y_1' = y_1 + (r + 2w) \sin \alpha \quad (\text{EQ 8})$$

$$x_2' = x_2 + r(1 - \cos \alpha) \quad (\text{EQ 9})$$

$$y_2' = y_2 + r \sin \alpha \quad (\text{EQ 10})$$

The new location and direction for the vehicle can be found from the final positions of the wheels.

The new global position for the agent is found by converting the relative location and direction of the vehicle from local coordinates to global coordinates through linear algebra. The motors and light sensors of the Agent are then sent the Agent's new global location so that they can update their locations.

The graphical representation of the Agent also needs to be updated. Translations are implemented on a 50 to 1 scale so that the movement looks smooth. Both translations and rotations for the representation are achieved through transformation matrix manipulations on the graphical model (an in depth discussion of transformation matrices can be found in *Computer Graphics: Principles and Practice Second Edition* by Foley, van Dam, Finer, Hughes page 201 - 226.) The screen is refreshed after each change to the Agent so that the change is seen.

Further calculation is needed to prevent Agents from ending up in the same place as another item in the virtual world. When the Agent is determining the values for the obstacle sensors, it created a list of point obstacles that it sensed. This list contains a good estimation of point obstacles that the Agent may crash into during this incremental move. If the new position for the agent conflicts with the position of any of these point obstacles, the Agent stops so as to not occupy the same spot as another object.

7. Check if the simulation has completed. There are three ways to determine if a simulation is completed: the stop simulation button was pressed, the speed of both of the wheels are zero, or the global location of the agent has not changed.

When both wheels have a value of zero, the agent has stopped. When the Agent stops, it is not going to sense the world any differently than on its last incremental move. In this situation, the simulation has effectively ended. In some cases (i.e. when the Agent stops at an obstacle) the Agent's wheels are not zero, but the Agent has effectively stopped because there is no way that it can move unless the obstacle is moved.

When the agent stops, its value for its `is_stopped_` member is set to `TRUE`. If something is changed externally (such as a light being added, or an obstacle being moved), the `is_stopped_` member is set back to `FALSE` so that it can react to the new surroundings.

In some cases the agent can not see the light (i.e. the light is behind the agent). The program has a cut-off of 100 steps. If the agent does not stop at the light by then, it stops, reports that it could not find the light, and returns control to the user.

Discussion

There are two areas of extreme difficulty for this project: (1) Creating the proper sensor model for the agents including the proper placement of the sensors to get interesting behavior, and (2) Developing a movement model that is both believable, yet computationally simple enough to keep movement smooth. Doing calculations quickly became a theme in both the sensing and movement models for the agent, and although I make an effort to maintain simplicity, movement performance in the final project still suffers somewhat.

Sensing the World

The agent starts each step of its movement by sensing the virtual world. The ability of the sensors to detect the world with some accuracy is an important stage in the movement of the agent. The first time that I created it, the sensors on either side of the agent seemed to always see the items in the world at the same level (e.g., the values for the sensors on both sides of the agent are always the same). After some investigating, I realized that, since the sensors were not sufficiently far enough from each other, there was not enough of a differential in the distances of the sensors to the object they were trying to detect. To resolve the problem, I could either (1) make the measurement of distance to a higher precision, or (2) widen the distance between the sensors so that there is more of a difference in the readings.

In the final program, as many calculations as possible are done using integers because calculations can be done faster than when using floating point numbers. The distance reading for each of the sensors is also calculated using integer math, and increasing the precision for calculating the distance would require floating point calculations. I decided to forgo this method unless others did not work. It is important that the sensor readings of the Agent are based on calculations that are to the level of precision that the Agent can actually move. Because of the pixelated nature of computer screens, none of the objects can be placed anywhere on the screen with any higher precision than integer values. Sensor readings based on floats when locations are based on integers can lead to unnecessary precision.

Widening the distance between the sensors increases the distance differential enough such that the sensors end up with different values when the Agent is not directly facing the detection source. No higher precision is needed to increase this differential, calculation times are kept low, and small differences in sensor distance from a detected object are ignored.

Moving in the World

The movement of the Agent can be based on one of three models: instantaneous -- the Agent instantly moves to its final destination; constant velocity -- the Agent takes even steps to get to its destination; acceleration -- the Agent follows acceleration models to switch between velocities to get to its destination.

Instantaneous Model. The instantaneous model is clearly not acceptable because the interesting part of the Braitenberg-type vehicles is how they get to their destination, rather than where they end up. Using the instantaneous model on an incremental scale (e.g., discrete steps) can lead to realistic movement. I ended up using an instantaneous model on an incremental scale for this project.

Continuous Model. The continuous velocity model is accomplished by determining the path that the agent travels, and then reconstruct this path so that intermediate steps are evenly spaced. This method is very useful when a single equation for the movement of an object can be determined, as well as how long it will take the object to get to its final destination. Creating an equation of the movement of an Agent is very difficult. Since the movements of the Agent depend not only on the position of all of the other items in the world, but also on the dynamic position of the Agent, and possibly the dynamic position of other Agents (and consequently time). Only after this complex equation is determined (which is no small task), the movement of the Agent(s) begin(s). At each tick of the clock, the equation is reevaluated for that point in time to determine the position of the Agent at that time.

This type of computation is not acceptable for all simulation environments since movement can not always progress in a smooth manner. However, there are some interesting aspects of this approach which would make it attractive for some types of applications.

For example, if the simulator was used to analyze a car accident, determining an equation might be a good alternative. The driver can be programmed to have eye sensors, and actuators that step on the gas and break pedals of the car, as well as actuators that turn the steering wheel. The wheels of the car could have sensors to detect the torque of the steering wheel, as well as the depression of each of the pedals. With the assumption that most reactions during times of stress (in this case, an accident) are reflex reactions, and therefore can be modeled, it is straight forward to create a virtual world which contains objects with the correct sensors and actuators to model the situation. In this case of an accident, it is often necessary to pinpoint specific points of time during the accident (for example, the time when the driver recognized the obstacle, the time when the driver stepped on the break, the point of impact, etc.) If a mathematical model of the accident existed, a time could be entered, and the simulator could calculate the position of all of the items in the world at that point in time. This operation could be done without running through the progression that led up to the point, making this method a non-linear way of seeking time-based information about the accident. The benefit of being able to do non-linear seeking in this case may outweigh the cost of building the complex equation that would allow the seeking to be done.

Acceleration Model. The acceleration model is perhaps the most accurate model for object movement. The next position for the object is based, not only on the location of the other items in the world, but also on the speed of the object at the previous position, and the length of time between the calculations. Agents gradually arrive at a speed over time instead of instantly traveling at the speed.

Using an acceleration model allows for some interesting things to be modeled. For example, one can model travel on a hilly surface, or on a surface with high friction by including how these factors affect the acceleration model for the world.

The architecture of Braitenberg-type vehicles does not allow for the vehicle to maintain any state information. Instead, it is supposed to react to the world around it based purely on what it senses. This architecture leads to a problem in representation for the acceleration model. The acceleration model requires that the Agent know how fast it was going at the previous step before it can assess the speed at which it will travel at this step.

Possible Uses

To implement the acceleration model requires extra calculations for each step of the movement. Some type of equation is needed to determine a new wheel value based on some acceleration value for the Agent and the previous wheel value. This equation and the acceleration value would need to be modeled to reflect real-world conditions.

The simulator built for this project has different goals for some of the sample applications described above. It is important for the user to be able to change the world in some way (for instance, move an object in the world), and quickly be able to see how this affects the movement of the Agent. In addition, the world may be changing while the agent is moving so the Agent truly needs to sense its surroundings while it is moving. I decided to use an incremental move model in which the Agent moves using the instantaneous model, but each movement is for a very short distance. Unless the world changes very quickly, the Agent appears to move at smoothly changing distances (velocities) for each step.

Possible Uses

There are many reasons why a simulator might be used rather than actual robots.

Portability. Computers are generally accessible anywhere. Transporting programs to other computers is easy, while transporting a robot to another location sometimes even requires taking the robot apart and reassembling it.

Reliability. Once a program is on a computer, the program will run the same way each time it is run. A robot is not always as reliable. Robot behavior can vary based on a number of factors including the floor covering, the room humidity, abundance of ambient light, etc. Since the simulated environment is more predictable and isolated, it is often easier to determine what factor is causing a particular behavior, and, in the situation where reproducing a simulation is important (in a proof of concept, for example), the reliability of the simulator can be very comforting.

Prototyping. Before building an actual robot, it is often desirable to prototype it in a less expensive environment. A simulator is an excellent way to prototype, allowing for quick and easy changes, and for testing in a variety of environments. In some situations, the prototype may provide sufficient information.

Scalability. Some experiments may require a large amount of space, or a facility that is not accessible to the researcher. A simulator allows the robots to explore anywhere that can be modeled. Some areas, such as inside a human vein, or on the ocean floor, may be impossible or too expensive to explore using actual robots.

Accessibility. If the desired task for the robot is to assist a user with computer-related tasks, the best place for that robot may be on the computer. In these types of situations, a simulator might easily be transformed to an actual "computer robot."

Future Work

There are many directions this project could take. This section briefly lists possible projects to continue this work. Each item listed here is described in detail in Appendix B.

- **Multiple Propagation Models.** Currently if there are multiple agents in the virtual world, they all have to use the same sensor-actuator function for their movement. Having multiple functions would allow agents in a simulation to behave differently.
- **Allow Agents to Move Independently.** All of the Agents in the world start to move when the Start Simulation button is pressed. There is no user interface for allowing the agents to move independently.
- **Make the Control Panel of the Application Work While the Simulation is Running.** User actions in the Control Panel are queued while a simulation is running. If the Panel worked during the simulation, changes could be made to the environment while the Agent is moving (for example, a light that the agent is approaching could be moved as the agent arrives.)
- **Change User Interface to a Drag and Drop Model.** To change the position of an item in the world, the user must know the new coordinates for the object, make sure that the correct object is shown on the control panel, and type in the new coordinates. A better approach would be to have the user click on the object to be moved, and drag it to its new position.
- **Change the Underlying Graphics Libraries.** The Brown Graphics Group intends to eventually stop supporting the WAV and OD libraries -- the two libraries necessary for most of the images seen during the simulation. The graphics in the simulator need to be changed to use libraries that will be supported.
- **Extend the Model to Three Dimensions.** Although all of the locations, directions, and graphical representations in the simulator are internally represented in three dimensions, the movement of the objects are restricted to two dimensions. Interesting results could be obtained from working in 3D.
- **Add Other Objects to the World, and Sensors to Detect Them.** The Agents can currently only detect Lights and Obstacles. Other objects such as colored lights, or goals can be added to the world along with sensors for the agent to detect the new objects.
- **Add the Ability to Change Sensor Direction.** The sensors on the Agents are hard coded to point in a particular direction relative to the Agent. Functions could be added to change the direction of the sensors.

References

- Braitenberg, V. (1984). *Vehicles: Experiments in Synthetic Psychology*. the MIT Press
- Conner, B (1993). *OD: Objects to Draw in WAV windows*. Department of Computer Science, Brown University.

References

Conner, B & Grimm, C. (1992). WAV:Classes Encapsulating Windows and Viewers for 3D Graphics - Draft [1.5]. Department of Computer Science, Brown University.

Foley, J. D., vanDam, A., Feiner, S. K., & Hughes, J. F., (1990). Computer Graphics: Principles and Practice, Second Edition. Addison-Wesley Publishing Company

Zeltzer, D & Johnson, M. B. (?). Virtual Actors and Virtual Environments: Defining, Modeling and Reasoning about Motor Skills. Interacting with Virtual Environment, L. MacDonald & J. Vince eds. John Wiley & Sons.

Zeltzer, D. & Johnson, M. B. (1991). Motor Planning: An Architecture for Specifying and Controlling the Behavior of Virtual Actors. The Journal of Visualization and Computer Animation, Vol 2, 74 - 80.

Appendix A. Class Structure

This section discusses both the coding conventions used for this project, and the class structure for the project.

Coding Conventions

TABLE 1. Variables

Example	Description
other_variable	All variables are in all lower case letters, with words separated by underscores.
member_variable_	Member variables of a class have the same format as other variables, but are followed by an underscore.
pPointer_variable	A pointer to a variable have the same format as other or member variables, but are preceded by a lower case p followed by a capital letter.

TABLE 2. Other Conventions

Example	Description
functionName()	Name of a function starts with a lower case letter with every new word starting with a capital letter.
IAVRClassName	Class names start with the program identifier IAVR, and have capital letters to start every new word.

Class structure

Alphabetical listing of the classes.

IAVRAgent

File Name(s): IAVRAgent.C IAVRAgent.H

Description: Models a virtual agent with an arbitrary number of sensors and motors

IAVRAgentItem

File Name(s): IAVRAgentItem.H

Description: Models an item in an agent list

IAVRAgentList

File Name(s): IAVRAgentList.C IAVRAgentList.H

Description: Models a list of agents for the virtual world

IAVRApplication

File Name(s): IAVRApplication.C IAVRApplication.H IAVRApplicationControls.C IAVRApplicationHandles.C

Description: creates a virtual environment and sets up the virtual agents for the application. This class handles all user input, and updates for the screen through motif calls.

IAVRLight

File Name(s): IAVRLight.C IAVRLight.H

Description: Models a light in the virtual world

IAVRLightItem

File Name(s): IAVRLightItem.H

Description: Models an item in a light list

IAVRLightList

File Name(s): IAVRLightList.C IAVRLightList.H

Description: Models a list of light sources for the virtual world.

IAVRLightSensor

File Name(s): IAVRLightSensor.C IAVRLightSensor.H

Description: Models a light sensor from the parent class sensor

IAVRLocation

File Name(s): IAVRLocation.C IAVRLocation.H

Description: Models and x, y, z location and allows for arithmetic

IAVRMotor

File Name(s): IAVRMotor.C IAVRMotor.H

Description: Models a motor for a virtual agent

IAVRObsSensor

File Name(s): IAVRObsSensor.C IAVRObsSensor.H

Description: Models an obstacle sensor from the parent class sensor

IAVRPtObs

File Name(s): IAVRPtObs.C IAVRPtObs.H

Description: Models a point obstacle

IAVRPtObsItem

File Name(s): IAVRPtObs.H

Description: Models an item in a point obstacle list

IAVRPtObsList

File Name(s): IAVRPtObsList.C IAVRPtObsList.H

Description: Models a list of point obstacles for the virtual world

IAVRSensor

File Name(s): IAVRSensor.C IAVRSensor.H

Description: Models a generic sensor and is the parent class for other sensors

IAVRWall

File Name(s): IAVRWall.C IAVRWall.H

Description: Models a wall in the virtual world

IAVRWallItem

File Name(s): IAVRWallItem.C IAVRWallItem.H

Description: Models an item in a wall list

IAVRWallList

File Name(s): IAVRWallList.C IAVRWallList.H

Description: Models a list of walls for the virtual world.

Appendix B. Future Projects

This appendix describes the projects described in the Future Work section of this paper. For each project, a description, suggested procedure, and estimated time is included.

Multiple Propagation Models

Description: Currently if there are multiple agents in the virtual world, they all have to use the same sensor-actuator function for their movement. Having multiple functions would allow Agents in a simulation to behave differently from each other.

Estimated Time: 40 - 80+ hours depending on the person's familiarity with the IAVR code, and with function lists. A knowledge of C++ is essential, and a familiarity with Makefiles will be helpful. Motif experience will be necessary to build a user interface for this new function.

Getting Started: This project consists of two parts: determining the best procedure for implementation, and actually implementing the new model.

Currently the propagation code for the agents is stored in a global function called IAVR-Propagate(). This function is called by IAVRAgent::incrementalMove(int, int, int, int) which is located in the file iAVRAgent.C. The Application stores a list of agents in its member variable IAVRApplication::agent_list_.

What needs to be done? You will need to create a list of function pointers for the IAVRPropagate code (this list would ideally be a dynamic list). You will also need to devise a scheme for attaching particular Propagate code to different Agents. Each Agent should know which Propagate code it uses. You will also need to build some sort of user interface to allow the user to specify which propagate code will be used for each agent in the virtual world. It would be nice if the different versions of the Propagate code were easily recognizable from a users point of view (maybe the name of the "brain" is the name of its function?)

Allow Agents to Move Independently

Description: All of the Agents in the world start to move when the Start Simulation button is pressed. There is no user interface for allowing the Agents to move independently.

Estimated Time: 10 + hours depending on the person's knowledge of motif and this project. Some knowledge of motif would be helpful, or at least a strong desire to learn motif.

Getting Started: Code for moving an Agent independently is located in `Agent::moveToLight(const IAVRLightList&, const IAVRPtObsList&)`. Calling this function causes that Agent to move by calling its `propagate()` code. [NOTE: the name `MoveToLight` is named as such due to historical reasons.]

What Needs to be Done? You need to create a motif button and its callback function(s). You should familiarize and understand all of the code in the file `IAVRApplicationControls.C`, and understand how the callbacks are defined in the `IAVRApplication.H` file. You can look at the callbacks `IAVRApplication::addAgent` and `IAVRApplication::agentXChanged` located in the file `IAVRApplicationHandles.C` for ideas on how to implement your callback

Make the control Panel of the Application Work While the Simulation is Running

Description: User actions in the Control Panel are queued while a simulation is running. If the Panel worked during the simulation, changes could be made to the environment while the Agent is moving (for example, a light that the agent is approaching could be moved as the agent arrives.)

Estimated Time: 20 + hours depending on the person's knowledge of motif.

Getting Started: Although I don't have extensive knowledge of motif, I believe that there should be some way to get user events while you are doing other code. For example, when a user starts a simulation, the callback function `IAVRApplication::handleMove` is called (located in `IAVRApplicationHandles.C`). This function runs through a loop and move the agent incrementally for each iteration. If you could check for user events, and handle the ones you want from within this loop, you should be able to dynamically change the world while the simulation is running.

What Needs to be Done? You need to investigate the motif books and look for a way to query for user events. You will need to handle these events from within the `handleMove` function. When you are successful with this, you can add a "Stop Simulation" button (or change the Start Simulation button to a Stop button when the simulation is proceeding), and remove the 100 step limit in this function. This will eliminate the possibility of the user having to hit the Start Simulation button more than once to see an entire simulation.

Change User Interface to a Drag and Drop Model

Description: To change the position of an item in the world, the user must know the new coordinates for the object, make sure that the correct object is shown on the control panel, and type in the new coordinates. A better approach would be to have the user click on the object to be moved, and drag it to its new position.

Estimated Time: Unknown. This project should only be attempted after the underlying graphics libraries (WAV and OD) are replaced. Some experience with the graphics group libraries would be helpful.

Getting Started: If you are not familiar with the Graphics Group libraries for WAV and OD, you can find user manuals for them in `/pro/uga/doc/wav/wav.dvi` and `/pro/uga/doc/od/od.dvi`.

What Needs to be Done? This would be determined by the replacement graphics library.

Change the Underlying Graphics Libraries

Description: The Brown Graphics Group intends to eventually stop supporting WAV and OD libraries -- the two libraries necessary for most of the images seen during the simulation. The graphics in the simulator need to be changed to use libraries that will be supported.

Estimated Time: Unknown. The time required depends on the new graphic libraries. Some familiarity with transformation matrices would be extremely helpful, but can be learned by reading pages 201 - 226 of *Computer Graphics: Principles and Practice*, Second Edition (Foley, et. al.)

Getting Started: If you are not familiar with the Graphics Group libraries for WAV and OD, you can find user manuals for them in `/pro/uga/doc/wav/wav.dvi` and `/pro/uga/doc/od/od.dvi`.

What Needs to be Done? You will need to talk to people in the Graphics Group to learn what they suggest to replace the graphics libraries WAV and OD. The WAV code is used in `IAVRMain.C` and in `IAVRApplication::IAVRApplication()`. You will also need to replace the `createPicture()` code for the Light, Agent and Wall.

Extend the Model to Three Dimensions

Description: Although all of the location, directions, and graphical representations in the simulator are internally represented in three dimensions, the movement of the objects are restricted to two dimensions. Interesting results could be obtained from working in 3D.

Estimated Time: This project could be another masters project.

Getting Started: You should familiarize yourself with the class IAVRLocation, and with the paper Braitenberg Vehicles in a Virtual Environment.

What Needs to be Done? You will minimally need to determine equations for determining the distance travelled in each incremental step. You will have to determine how the Agents move in three dimensions, and possibly create other types of actuators in addition to the current wheels. You will also need to add user interface elements.

Add Other Objects to the World and Sensors to Detect Them

Description: The Agents can currently only detect Lights and Obstacles. Other objects such as colored lights or goals can be added to the world along with sensors for the agent to detect the new objects.

Estimated Time: 60 - 80 + hours for each object/sensor combination. Some experience in C++ would be helpful.

Getting Started: You should look at either the Wall series or the Light series of code. The series of code consists of the object (IAVRWall or IAVRLight), the object's list item (IAVRWallItem or IAVRLightItem), and the object's list (IAVRWallList or IAVRLightList). You will also need to create a sensor to detect the new object. For this, you should look at the class IAVRLightSensor.

What Needs to be Done? Using the classes suggested above, you should create a new object and a sensor to detect it. You will need to add the sensors to the Agent member list, and add a new incremental move function to IAVRAgent which takes a list of the new object (as well as the other things you want to detect) to get the sensor readings. You will need to create a member list of the object in the class IAVRApplication.

Add Ability to Change Sensor Direction

Description: The sensors on the Agents are hard coded to point in a particular direction relative to the Agent. Functions could be added to change the direction of the sensors.

Estimated Time: 80 - 130 + hours depending on knowledge of this code.

Getting Started: In the file `IAVRAgent::Agent()` you will notice that, when the sensors are created, they are given the direction that the agent is pointing by default. The direction of the sensor is one of its properties (see `IAVRSensor.H`), but there is currently no method to change this direction.

What Needs to be Done? You will need to create a method in the class `IAVRSensor` that changes the direction of the sensor. You will need to call that function whenever the direction of the sensor changes. Because of the nature of the agent, you probably want to save the direction as an offset to the direction that the agent is facing (this is a similar model as what is done with the sensor locations). You will need to add some sort of user interface to allow users to access the new functionality.

Braitenberg Vehicles in a Virtual Environment: Users Guide

By Laura Dorival Paglione

This document describes how to use the simulator VRAgents. For information about the internals of this project, see the companion paper: Braitenberg Vehicles in a Virtual Environment.

Valentino Braitenberg, in his book *Vehicles: Experiments in Synthetic Psychology*, describes a series of hypothetical, self-operating machines which he calls "vehicles." All of the vehicles have a very simple internal structure, and are discussed as though they exhibit some sort of civilized "behavior." His vehicle's behaviors are borne out of a direct connection of input sensors to output actuators. This connection allows the vehicles to be able to directly translate what they sense into how they react.

In this paper I discuss how to use this simulator for Braitenberg-type vehicles. This paper also serves as a guide to interface this project with others (such as a compiler which will automatically set up, or link the conditions of a simulation.)

What is in this guide?

This guide is divided into two sections:

- Section 1: Guide for using the simulator - describes what the buttons do, how to set up the environment for a simulation, and how to run the program. This section is intended for people actually running the simulations.
- Section 2: Guide for interfacing with the simulator - describes how to link different Propagate() code to the simulator, and how to set up various structures for expected behavior. This section is written for a project coordinator, or person responsible for linking Rex or C code with the simulator.

This paper assumes that the user is familiar with Braitenberg-type vehicles, and is familiar with Rex programming models.

Section 1. Guide for using the simulator

Before using this guide you will need to review some information given to you by the project coordinator. This should include:

1. **Location of the program VRAgents.** The name of the simulator program is VRAgents. This is the name that you will type at the UNIX prompt to start the program.
2. **Description of how to link your sensor-actuator code.** The simulator can use code that you wrote as the “brains” for the vehicles. Your project coordinator will give you instructions on how to compile and link your code so that it will be usable by the simulator.

Getting Started

The simulator uses special libraries written by the Brown Graphics Group which access Silicon Graphics International’s XGL libraries. These libraries will only work if you are using Open Windows (instead of X Windows, for example). When you log on to the computer, be sure that you choose Open Windows as your window environment. In addition, the simulator takes full advantage to any graphics hardware that is in the computer. For the best results, use one of the computers that are located in the Sun Lab. These computers are equipped with Leo cards which will accelerate the graphic redrawing, and will cause the pictures to look and run better.

Once you have linked your “brain” code with the simulator, type the following at the UNIX prompt to start the program:

```
> VRAgents
```

This will start the simulator.

You will see two new windows open on your screen. The larger window is the Viewer. This window is where you will see the movement of the vehicles (which are called Agents in this program). The smaller window is a Control Panel. This window is where you will make the adjustments in the virtual world to achieve the simulation that you want.

The Viewer Window does not have any drag or drop capabilities, nor does it detect any mouse or keyboard commands. This window is only for viewing your simulation.

The Control Panel

The Control Panel has:

- a menu bar with two sub-menus: file and delete
- a start simulation button
- a quit button
- three sections: Agent, Light, and Wall, all of which contain a drop-down list box to switch between objects, a button to add an object, and two text boxes to change the x and y coordinates of the object. The Agent section also contains two text boxes to change the direction of the Agent.

When positioning objects, you should note that the Viewer Window is set up in Cartesian Coordinates. The center of the Viewer Window is the point (0, 0), and the corners of the Window are (-45, -45) [lower left corner], (45, -45) [lower right corner], (45, 45) [upper right corner], and (-45, 45) [upper left corner]. If you try to position an object off of the screen, the program will force the object back on the screen. For example, if you tried to put an object at the point (3, 70), the program would actually put the object at the point (3, 45).

The Menu Bar

The menu bar of the control panel has two sub menus: File and Delete.

The File Sub Menu. The File sub menu has three choices:

- Start Simulation - This option has the same effect as when you press the Start Simulation button that is on the Control Panel. Choose this option to start your simulation (see also "Start Simulation Button" below for more information.)
- Reset - This option moves the agent to (0, 0). [This option should be removed. It no longer has much usefulness]
- Quit - Choose this option to quit the program.

The Delete Sub Menu. The Delete Sub Menu has options to delete objects in the virtual world. At the time of this writing, none of the options are programmed to do anything interesting.

The Start Simulation Button

When you press the Start Simulation button, all of the Agents in the world begin to move according to the "brain" code that you programmed in Rex or another language. You should click this button after you have set up the positions of all of the objects in the world.

While a simulation is running, any changes you make or buttons you press will be queued until the simulation has stopped running. It can become quite frustrating if you press the start button by mistake, and a long (or possibly even an infinite) simulation occurs. To help make mistakes like these less disastrous, the program runs the simulation until it stops, or until the agent has move 100 "steps," whichever comes first. (The agents move through a series of incremental steps. Every time that you see the Agent make a small movement during a simulation, it has taken a step.) Using this method, the simulation will only run a maximum of 100 Agent steps after the Start Button is pressed. One downfall of this approach is that, when you press the Start Simulation Button, the Agent may not move far enough to complete the simulation. As a result, you may have to hit the button more than once to run the entire simulation.

The Quit Button

When you are finished running your simulation, press the quit button to terminate the program.

The Sections

There are three sections (Agent, Light and Wall) on the control panel which allow you to make changes in the virtual world. All of the sections consist of at least the following things:

1. The name of the object for the section

2. A drop-down list box for changing the current object for the section
3. A button to add a new object
4. Two text boxes to change the x and y location of the current object for the section

1. The name of the object for the section. There are three types of objects that exist in the world: Agents, Lights, and Walls. To manipulate each of the objects, there is a section for handling each type of object.

2. Changing the current object for the section. Each of these sections has a concept of the “current” object for which all of the information is displayed (for example, the location text boxes show the location of the current object). The first box at the top of each section is a drop-down list box. On the right side of the drop-down box is a control which causes the list to show itself when it is clicked. If you select an object from this list, the item becomes the “current” object, and its information is shown in the other parts of the section (for example the position information is changed to reflect the new current object).

3. Adding new objects. Click the Add object button, and a new object of the section’s type will be added to the world. This object is also added to the drop-down list described in 2 above. If you want to change the position of an object you’ve just added, you will have to use the drop-down list box to change the new object to the current object. Objects are numbered sequentially in the order they were created, so the object that you just added will always be the highest numbered object on the list.

The program places new objects in default positions in the Viewer Window which may be the same position as another object in the world. If you click the Add object button, but don’t see a change in the Viewer Window, it may be because the new object was added where an object already exists. Try changing the object you just added to the current object, and change its location.

4. Changing the x and y locations of the current object. For each section, the program shows the location of the current object for that section. To change the location of the current object, type in the new x and y locations in the text boxed provided. The change you type will not be in effect until the text box loses focus. You can make your changes take effect by pressing the tab or return keys, or by clicking the mouse on another control.

Agents

Agents look like spheres in the Viewer Window. The control panel allows you to change the agents’ position and direction, and allow you to add agents to the world. All of these items work as described in “The Sections” above, with the exception of changing the direction of the Agent, which is not described above.

Changing Agent direction. As with changing the location of an object, you can change the direction of an Agent by typing the new x and y directions in the text boxes provided for the direction. To determine what to type for the new direction:

1. Imagine that the Agent is at the coordinates (0, 0);
2. Pick a point on the screen that the agent should face;

Section 1. Guide for using the simulator

3. Estimate the coordinates of the point determined in 2 given that the origin of the coordinate system is at the center of the agent;
4. Type in the value of the point the agent should face (from the estimation made in 3.)

It is not necessary to know the scale of the coordinate system when you make your estimation because the program is only interested in the ratio of the numbers you type for the x and y directions, not the actual values.

Lights

Lights look like orange cubes in the Viewer Window. The control panel allows you to change the Lights' position and allows you to add move Lights.

Walls

Walls look like yellow cubes in the Viewer Window. The control panel allows you to change the Wall's position and allows you to add move Walls.

Mini-Tutorial

Using the default brains for the agents, try changing the simulation. The default brains make the agents move toward lights and avoid obstacles.

Move into the directory where the simulator is by using the UNIX command `cd`.

```
> cd [the directory]
```

Start the simulator by typing the following at the UNIX prompt.

```
> VRagents
```

The simulator program should start. You are now ready to try out your first simulations.

Simulation 1. For the first simulation, try leaving all of the objects where they are. Run the simulation by pressing the Start Simulation button, or by choosing Start Simulation from the File menu.

You will see the Agent move to the light. In this simulation, the obstacle is not in the way, so the Agent doesn't need to avoid it.

Simulation2. Move the agent back to where it started. Its starting position was (0, -40).

- Double click in the box for Agent Location X. Type the number 0, and hit the tab key
- Motif controls have similar navigation commands to emacs. Hit C-A to move the cursor to the beginning of the Y Location text box. Hit C-D a few times to erase the number that is there, and type in -40. Hit the tab key to make the change take effect.

Change the direction back to the direction it was pointing originally by changing the direction to (0, 1). (Do this the same way that you changed the agent's direction)

Change the location of the light to (40, 10)

Change the location of the wall to (5, 0)

Section 1. Guide for using the simulator

Add a Light by pressing the Add Light button. (this adds a new light -- Light 2 to the center of the Viewer Window)

Switch to the new light by selecting it from the Light drop-down list box. You can do this two ways: (1) click on the small rectangle next to the label "Light 1" and choose Light 2 from the list, or (2) When the Control Window panel has the focus, press M-L. The drop-down list will expand. Now you can type the number 2 or click on the label "Light 2."

Change the location of Light 2 to (-40, 10)

Start the simulation by pressing the Start Simulation button

Simulation 3. You can try out your own environments or press the Quit button to quit the simulator.

Section 2. Guide for interfacing with the simulator

Linking user defined code is fairly simple, but a few rules must be followed:

1. The purpose of the user defined code is to specify the values for the wheels of the agent given the values of each of the sensors of the agent
2. The code must be written in C
3. The code consists of at least one global C-function called `IAVRPropagate()`. This is the only function that is called by the simulator.
4. The code uses the structure `IAVR_Input` to get the values for the sensors
5. The code puts the value of the motors in the structure `IAVR_Output`

Input and Output Structures

The two structures `IAVR_Input` and `IAVR_Output` are located in the file `IAVR RexInterface.H`. Below are the definitions of the structures:

```
struct Input
{
    int left_light_sensor;    // value between 0 and 100
    int right_light_sensor;   // value between 0 and 100

    int left_obstacle_sensor; // value between 0 and 100
    int right_obstacle_sensor; // value between 0 and 100
} IAVR_Input;

struct Output
{
    int left_wheel;           // value between 0 and 100
    int right_wheel;          // value between 0 and 100
} IAVR_Output;
```

You will need to include `IAVR RexInterface.H` in your file with the `IAVRPropagate()` function to use these structures.

IAVRPropagate() function

The function `IAVRPropagate()` has no parameters, and returns no value. It can do any sort of calculation or function calling allowable by C, but you should keep in mind that this code is called many times during a simulation, and the less computationally complex that you keep the function, the better the simulation will run. Also keep in mind that integer calculations are faster than floating point ones.

Sample Code

Your file should end up looking something like:

```
#include IAVRRexInterface.H

void IAVRPropagate()
{
    IAVR_Output.left_wheel =
        [some function of
            IAVRInput.left_light_sensor,
            IAVRInput.right_light_sensor,
            IAVRInput.left_obstacle_sensor,
            IAVRInput.right_obstacle_sensor];

    IAVR_Output.right_wheel =
        [some function of
            IAVRInput.left_light_sensor,
            IAVRInput.right_light_sensor,
            IAVRInput.left_obstacle_sensor,
            IAVRInput.right_obstacle_sensor];

}
```

You can name the file anything that you want.

Editing the Makefile

In the Makefile, you will see a keyword

```
REX_FILE      =
```

Type in the name of your file (that contains the IAVRPropagate() code) after the “=” sign, save the Makefile, and type make at the UNIX prompt.

```
> make
```

When the files have finished linking, you will be able to simply type VRAgents at the UNIX prompt to run the simulation. (You should note that this Makefile requires and objs/ director, and a dependent Makefile. See someone with more experience with Makefiles if you have further questions about this.)

[NOTE: a more skilled Makefile writer could probably make this a slightly less painful process by allowing the name of the file that you want to link be a command line variable to the make command.]