Introduction

The purpose of this design document is to describes two major design decisions that will be used in designing Braitenberg Vehicles Simulator. The first and second part of the document describe the design decisions related to using the Observer Pattern and the Strategy Pattern. For each design decision, two possible implementations are provided. The final design decision which is implemented within the finished simulation is described at the end of each design decision. The third part of the document provides a tutorial on how to add more stimuli to the simulator. There are four different scenarios in this simulator (Figure 1). Each robot has sensors and behaves differently depending on its scenario.

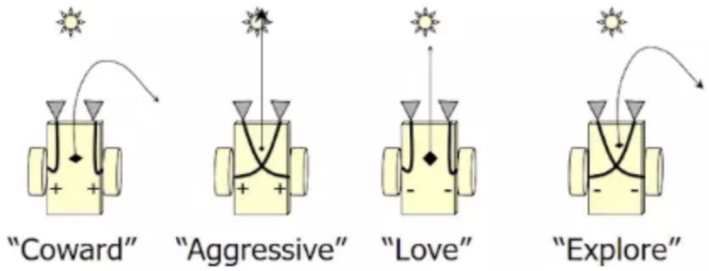


Figure 1: Four Different Behaviors for sensor

1 Observer Pattern

1.1 Introduction

This section includes the use of the Observer Pattern and two possible implementations relating to the Observer Pattern that works for the design of the simulation. The final design decision to be used out of the two possible implementations will be discussed in section 1.5.

1.2 Use of the Observer Pattern

The Observer Pattern is used within the simulation due to the one-to-many dependency between different sensors. This Pattern will address the problem on how the sensors receive information about different stimuli they are sensing. Information received by sensors will impact the wheels' motions of robots. The Observer and the Subject of the Observer Pattern will be discussed in 1.3 and 1.4.

1.3 First Implementation idea

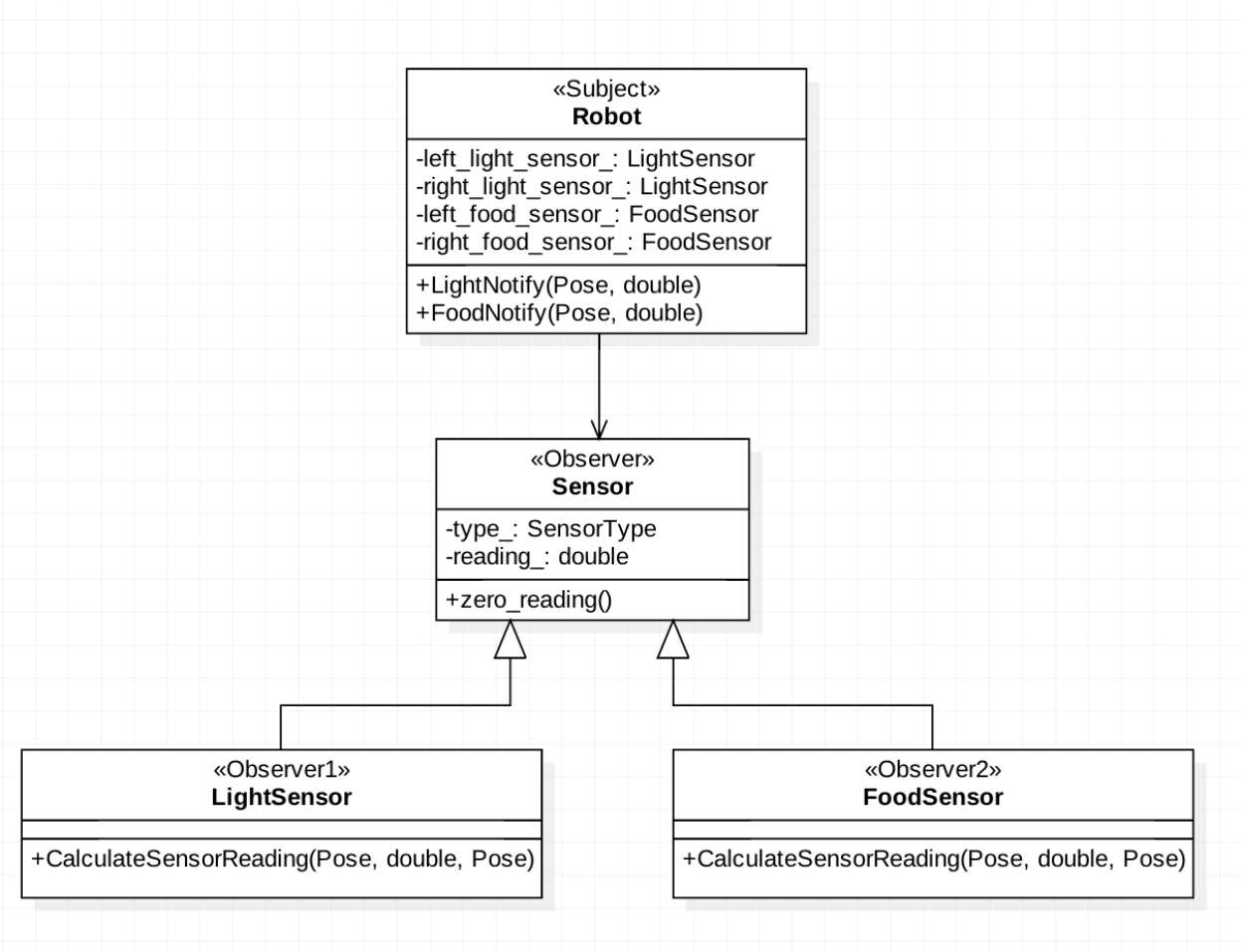
The first implementation idea is based on the idea that the Subject is the Robot Class, which serves to update robot's velocity and position and notify all the other sensors when an entity, such as food and light, has been updated for each time step in the arena. Observer is the main Sensor class. The Robot::LightNotify() and Robot::FoodNotify will be called whenever the observed object, such as food and light, is changed.

The benefit of this implementation is that robot can directly notifies sensors and trigger motion handler to control the wheels at the same time, which is efficient. The drawback of this implementation is that the information that the sensor receives is not directly from the arena, which holds all the information of entities. This makes the process of notify observers (sensors) in the arena more complicated and difficult to understand.

1.3.1 Justification of the design

The justification for using this pattern is practical and reasonable. At each time step, different sensors of each robot in the arena will be notified by relative entities. For example, light sensors will be notified by the robot on the pose and the radius of the lights from arena and food sensors will be notified by robot on all the information about food from arena. This design ensures the arena and the robot to know the status of each entity at a given time step. Using this design the Robot class holds all the information of sensors.

1.3.2 UML Diagram Example



1.4 Alternate implementation

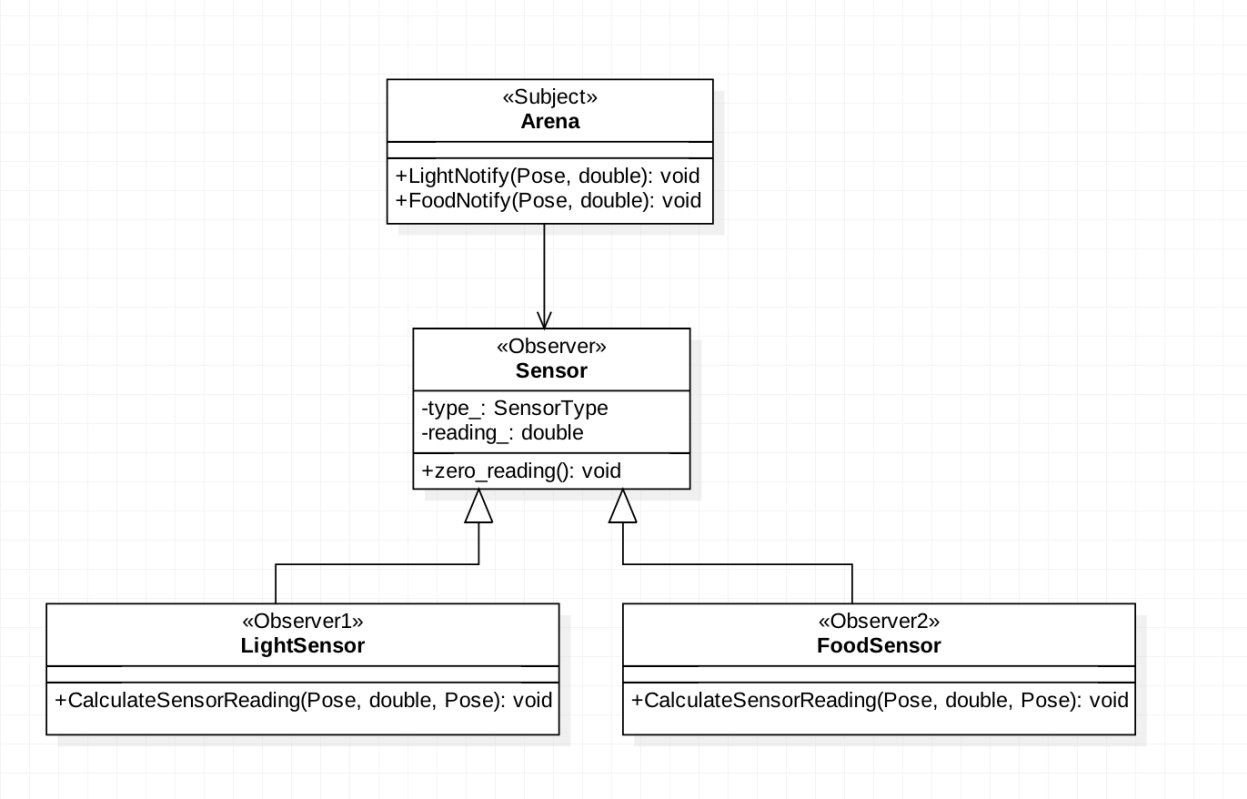
The alternate implementation is based on the idea that the Subject is the Arena Class, which serves to update each entity's velocity and position and notify all the other observers when a sensor has been updated for each time step. The Observer is the main Sensor class which is called whenever the status of observed object is changed.

The benefit of this implementation is the accessibility of the Arena class because the arena always holds all the information of the sensors. The drawback is that this implementation separates the robot and its sensors in the way that robot not knowing the information of other entities that its sensor knows, which may impact the consistency and dependency between robot and sensors.

1.4.1 Justification of the design

Whenever the entity updates, the Arena can notify the relative sensor depending on its sensor type and the entity type. This design ensures the arena to know all the information of sensors and all the sensors can update at each given time step.

1.4.2 UML Diagram Example



1.5 Design to be used

The first implementation in 1.3 is the design to be used for this simulation.

1.5.1 Brief explanation why

In the Robot-Sensor Observer pattern, Robots are observers of stimuli, in that a Robot would register to receive information from the Arena about stimuli that it is actively sensing. The reason why Robot-Sensor Observer pattern is better than Arena-Sensor is because of the dependency between robot and sensor. Since all the sensors are parts of the robot, sensors will be more accessible in the Robot than in the Arena. Thus, it is safer and simpler to notify sensors in the Robot than in the Arena.

2 Strategy Pattern

2.1 Introduction

This section includes the use of the Strategy Pattern and two possible implementations relating to the Strategy Pattern that works for the design of the simulation. The final design decision to be used out of the two possible implementations will be discussed in section 2.5.

2.2 Use of the Strategy Pattern

The Strategy Pattern is used within the simulation due to the behaviors vary independently from the robot of different specifications (ie. coward, explore) that performs it. This pattern will address the problem on how robots of different specifications will behave according to the sensor readings. The Context and the Strategy of the Strategy Pattern will be discussed in 2.3 and 2.4.

2.3 First implementation idea

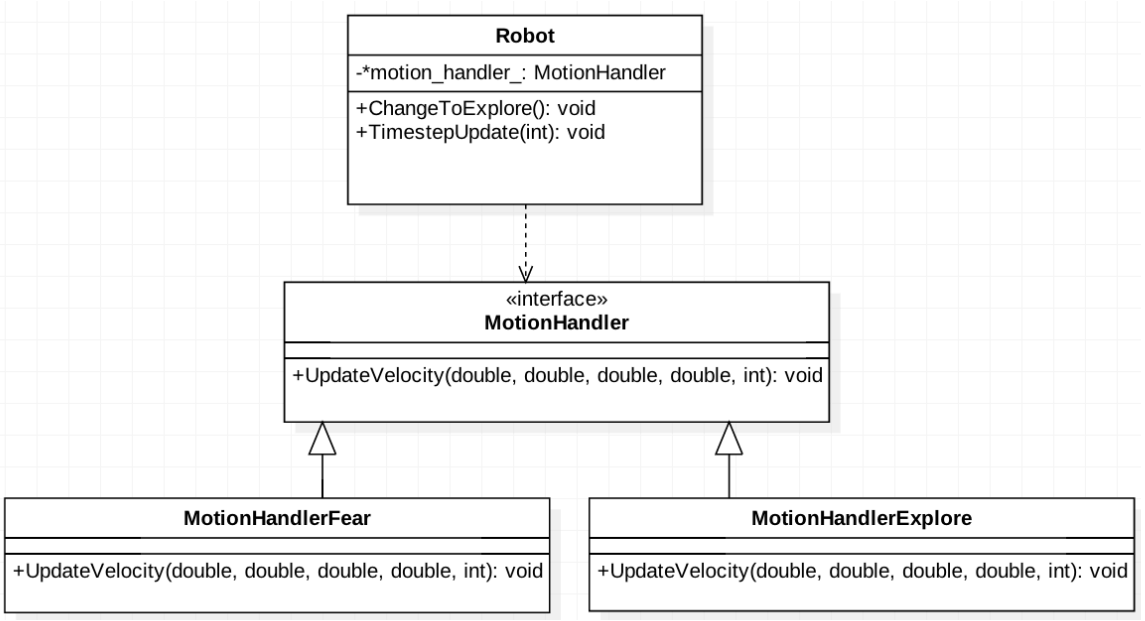
The first implementation is based on the idea that the context will be the Robot class and the interface is the MotionHandler class. Two strategies will be MotionHandlerFear and MotionHandlerExplore. The robot of fear (coward) will use MotionHandlerFear as its motion handler and robot of explore will use MotionHandlerExplore. Aggressive behavior will be realized in both MotionHandlerExplore and MotionHandlerFear and will be triggered when the robot is really hungry.

The benefit of this implementation is that when the robot is not hungry after eating food, the robot does not need to change motion handler, which cause less time and space. However, as for the audience/ new developer, it can be not clear enough for them that the robot has aggressive behavior towards food because the aggressive behavior is hidden/handled in the MotionHandlerFear and MotionHandlerExplore.

2.3.1 Justification of the design

This design is simple and practical. Firstly, the motion handler is declared as a MotionHandlerFear pointer (as default) in Robot constructor. When a robot specifying in explore is created in the arena, Robot::ChangeToExplore() will be called to make sure the robot has the correct behavior towards its type. When the robot is hungry, robot will behave aggressively towards food no matter what type the robot originally has. Thus, there will not be an independent MotionHandlerAggressive class in this simulation. Instead, both MotionHandlerFear and MotionHandlerExplore can perform aggressive behavior when the robot is very hungry.

2.3.2 UML Diagram Example



2.4 Alternate implementation

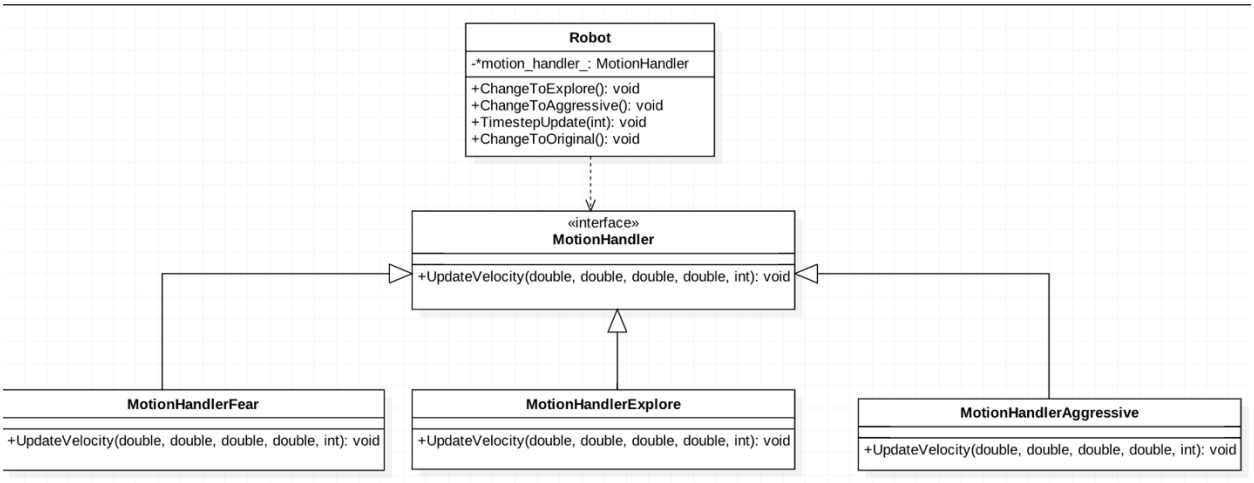
The Alternative implementation has similar idea as the first implementation idea. The context will be the Robot class and the interface is the MotionHandler class. The only difference here is this implementation has three strategies towards MotionHandler, including MotionHandlerFear, MotionHandlerExplore and MotionHandlerAggressive. The MotionHandlerAggressive will be used by robot when the robot is very hungry.

The benefit of this design is that it is clear for the audience that robot will have three different behavior towards different entity. The drawback is that when the robot is not hungry, the robot should change back its motion handler from aggressive to the original, which is more time-consuming and space-consuming for the whole program.

2.4.1 Justification of the design

This design is simple and reasonable. This design is simple and practical. Firstly, the motion handler is declared as a MotionHandlerFear pointer (as default) in Robot constructor. When a robot specifying in explore is created in the arena, Robot::ChangeToExplore() will be called to make sure the robot has the correct behavior towards its type. When the robot is very hungry, the motion handler will use aggressive strategy via calling Robot::ChangeToAggressive. When the robot is already eaten at a very hungry status, the motion handler will use its original strategy by calling Robot::ChangeToOrigin.

2.4.2 UML Diagram Example



2.5 Design to be used

The first implementation in 2.3 is the design to be used for this simulation.

2.5.1 Brief explanation why

The reason why this design is more beneficial than the other design is because of the time and space that this design can save. If we use the alternate implementation, the modification of motion handler in Robot class will happen whenever the hungry level changes between not hungry and very hungry. Since no matter the robot specifies in explore or coward towards light, robot should always perform aggressive towards food when it is very hungry, handling aggressive behavior in MotionHandlerFear and MotionHandlerExplore is simpler.

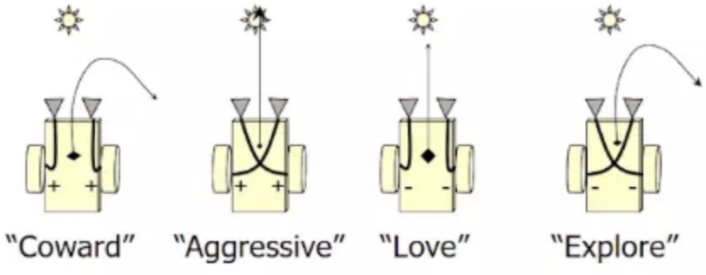
3 Feature Enhancement

3.1 Introduction

In this section, we will consider the scenario in which a new developer needs to add a new stimulus (e.g. water) and sensor to the simulator based on the current simulation. The description of scenario is at section 3.2 and a tutorial with snapshots of code is provided in section 3.3.

3.2 Scenario

There are four different sensor-wheel connections in this simulation, including Coward, Aggressive, Love and Explore. In this section, we will describe how the motion of robots in different specification will differed.



3.2.1 Coward

This vehicle spends more time in the places with less stimulation and speeds up when exposed to more stimulation. If the stimulation is directly ahead, the vehicle may hit the source. Otherwise, it will tend to turn away from the stimulation. The coward sensor-wheel connection is positive.

3.2.2 Aggressive

Since the sensor-wheel is crossed, aggressive behavior is the opposite of coward ‘s behavior. If the stimulation is directly ahead, the vehicle moves directly towards it. But, if the stimulation is to one side, the vehicle will tend to veer towards it with increasing speed.

3.2.3 Love

This vehicle has negative sensor-wheel connection. When approaching the stimulation, “Love” will orient towards it and finally face it.

3.2.4 Explore

This vehicle has negative and crossed sensor-wheel connection. When approaching the stimulation, “Explore” will orient towards it as Love will but come to rest facing away from the stimulation.

3.3 Tutorial

In this tutorial, we will add a new stimulus, water, and the water sensor to the simulator. Code examples are provided to provide an idea of how the stimulus and sensor will be implemented given the current code from current simulation. We will use the food and food sensor as the code to implement our new code.

3.3.1 Adding a new stimulus (water)

1) Make the header and source file for water.
    - Water will inherit from the ArenaImmobileEntity class.
    - Create constructors and methods related to water and make sure the position will be set randomly and the radius will be set randomly as needed.

```
Food::Food() : ArenaImmobileEntity(), captured_(false) {
  set_type(kFood);
  set_color(FOOD_COLOR);
  set_pose(FOOD_INIT_POS);
  set_radius(FOOD_RADIUS);
```

    - In the source file, there will be a reset function to reset the attributes and the position of the water object when the Reset game function is called.

```
1  void Food::Reset() {
2    Pose new_position = {static_cast<double>((30+(random()%19)*50)),
3         static_cast<double>((30+(random()%14)*50))};
4    set_pose(new_position);
5    set_color(FOOD_COLOR);
6    set_radius(FOOD_RADIUS);
7    set_captured(false);
8  } /* Reset */
```

2) Create the water entity in the entity factory class
    - Include the header file of the water class and make a private variable to create water

```
Food* EntityFactory::CreateFood() {
  auto* food = new Food;
  food->set_type(kFood);
  food->set_color(FOOD_COLOR);
  food->set_pose(SetPoseRandomly());
  food->set_radius(FOOD_RADIUS);
  ++entity_count_;
  ++food_count_;
  food->set_id(food_count_);
  return food;
}
```

3) Create a water type kWater in the entity type class

```
enum EntityType {
  kRobot, kLight, kFood, kEntity,
  kRightWall, kLeftWall, kTopWall, kBottomWall,
  kUndefined
};
```

4) Add water Entity in the Arena class
   - Define the amount of water entities in the params class

```
// arena
#define N_LIGHTS 4
#define N_ROBOTS 10
#define N_FOOD 4
```

   - Add the amount of water entities in the constructor of Arena

```
Arena::Arena(const struct arena_params *const params)
    : x_dim_(params->x_dim),
      y_dim_(params->y_dim),
      factory_(new EntityFactory),
      entities_(),
      mobile_entities_(),
      game_status_(PLAYING),
      game_paused_(false) {
  AddRobot(params->n_robots);
  AddEntity(kFood, 4);
  AddLights(params->n_lights);
}
```

3.3.2 Adding a new sensor

1) Make header and source file for sensor

   - The sensor will inherit from the Sensor class.
   - Add new sensor type to SensorType class

```
enum SensorType {
  kLightSensor, kFoodSensor
};
```

2) Create water sensors in the robot class'

   - Include the header file of the water sensor class and make two private variables to create left water sensor and right water sensor.

```
// Left Food Sensor
FoodSensor food_sensor_left_;
// Right Food Sensor
FoodSensor food_sensor_right_;
```

3) Add notify functionality in the robot class

```
void Robot::FoodNotify(Pose food_pose, double food_radius) {
  food_sensor_left_.CalculateSensorReading(food_pose, food_radius,
    get_sensor_position(LEFT_SENSOR));
  food_sensor_right_.CalculateSensorReading(food_pose, food_radius,
    get_sensor_position(RIGHT_SENSOR));
}
```

4) Add reset water sensor reading functionality in robot class

```
void reset_sensor_reading() {
  light_sensor_left_.zero_reading();
  light_sensor_right_.zero_reading();
  food_sensor_left_.zero_reading();
  food_sensor_right_.zero_reading();
}
```

3.3.3 Pass new sensor reading to motion handler in robot

1) Pass the left and right water sensor reading into the UpdateVelocity function in MotionHandler class.

```
virtual void UpdateVelocity(__unused double lt_left_reading,
  __unused double lt_right_reading, __unused double fd_left_reading,
  __unused double fd_right_reading, __unused int hungry_level) {}
```

2) Modify the UpdateVelocity function in MotionHandlerFear class and MotionHandlerExplore class as above. Make sure to relate the speed to both newly added readings.

3.3.4 Get updated from Arena

1) Enable the robot to notify the water sensor in arena.
   - In Arena:: UpdateEntitiesTimestep(), we add a new if statement to enable the robot to notify water sensor on the updates of water. Follow the pattern that how food sensor be notified.

```
for (auto ent : entities_) {
  /* For non-robot entities, update their velocity and position, and no
   * each robot's sensors their position
   */
   if (kRobot != ent->get_type()) {
    ent->TimestepUpdate(1);
    //  Notify the light sensors of each robot about each light's posit
    //  radius
    if (kLight == ent->get_type()) {
      for (int i = 0; i < N_ROBOTS; i++) {
        robot_[i] ->LightNotify(ent->get_pose(), ent->get_radius());
      }
    }
    //  Notify the food sensors of each robot about each food's positio
    //  radius
    if (kFood == ent->get_type()) {
      for (int i = 0; i < N_ROBOTS; i++) {
        robot_[i]->FoodNotify(ent->get_pose(), ent->get_radius());
```

3.3.5 Compile and Finish

1) Run “make” command in the src file in the terminal.
2) Run “ ./../build/bin/arenaviewer” in the terminal to check your new simulation.
3) Wonderful! You made it!

This concludes the tutorial of adding a new stimuli and sensor within the simulation.