ITERATION 2 DESIGN DOCUMENT

Introduction:

The following design document contains a description of the two major design choices made related to using the Observer and the Strategy Pattern. The third part of this document provides a feature enhancement tutorial on how to add a new stimulus (water) and sensor to the simulator.

Observer Pattern:

Observer pattern is a software design pattern in which there is a subject that maintains a list of its dependents, called observers. Anytime there is a change in the state of the subject, it notifies the observer about it. In other words, subject is the part with changing state and observer is the part that needs to know about these changes.

This is the first part of the design document that discusses two possible implementations for handling how robots receive sensor information. The first implementation makes uses of Observer pattern whereas the other just treats and uses Arena as the central management system. At the end of this part of the document, there is a justification for the design choice made.

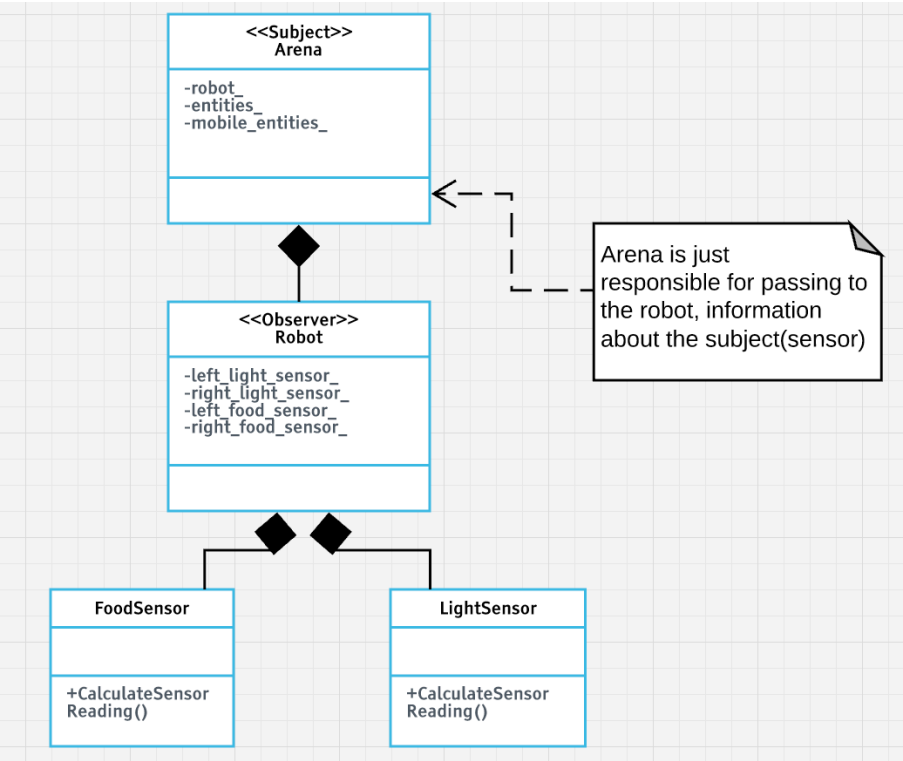
First Design Idea (uses Observer Pattern):

The first implementation idea is to treat the stimuli as the subject and the robot as the observer. At each time step, the robot receives information about every single stimulus in the arena from the Arena class since arena contains all the stimuli. (Even though the information is being passed through the arena, the robot observes and gets affected only by the stimuli.) Then based on whether the stimulus is light or food, robot uses its light or food sensors to calculate a sensor reading value pertaining to that particular stimulus to gauge how much is that stimuli affecting the robot. After which, depending on the value of the reading and its behavior towards that stimulus, the robot first decides whether its behavior needs to be changed and if so, it updates its motion and behavior accordingly.

Justification:

The use of this observer pattern idea is justified because arena contains all the entities (lights, food, robots) and so it can easily delegate the task of determining the impact of a stimulus on robot's behavior and motion to the robot itself. Furthermore, carrying out the observer pattern implementation this way allows us

to not let arena access any of the sensors of the robot and it intuitively makes more sense since it is robot's sensors and only robot should have access to it.



UML Showing the important classes and its attributes and functions necessary for Observer Pattern idea.

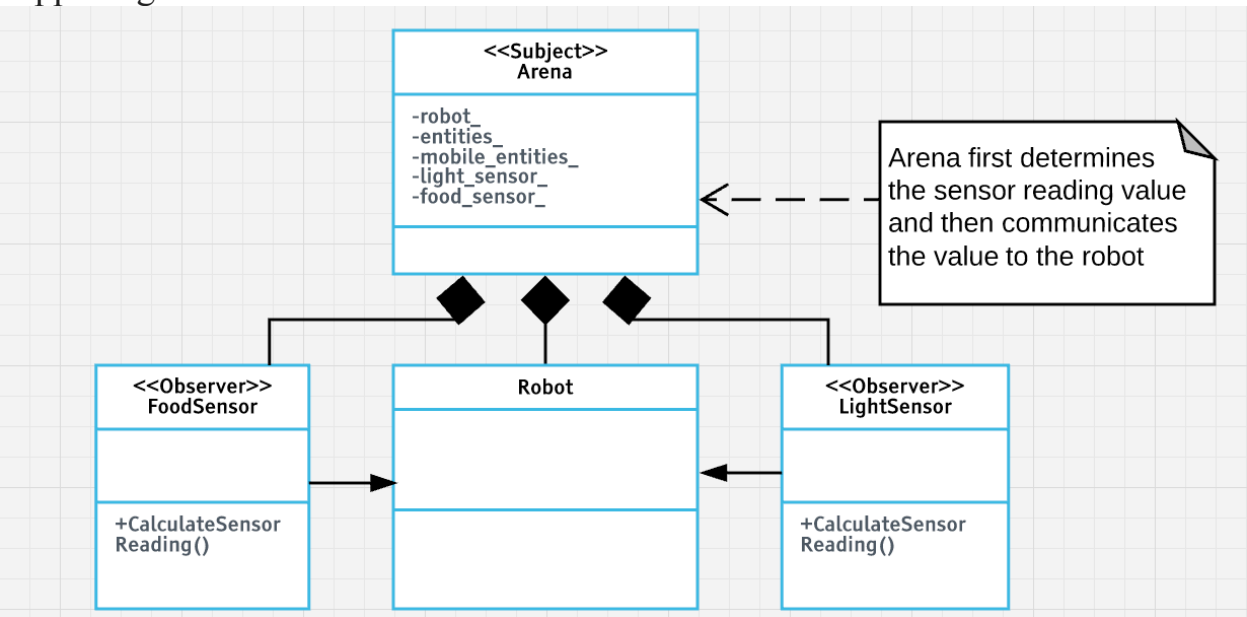
Second Design Idea:

The second implementation idea is to treat the Arena as the central management system for all actions and forms of communication pertaining to how robots receive sensor information. The arena has access to all the entities including the robots, lights and food. At every time step, arena checks for change in the state of a stimulus. When there is a change, arena notifies the robot sensors about this alteration and then the sensors calculate a sensor reading value. On update of the sensor reading, arena then takes this information and communicates it to the robot and the robot, based on its behavior, then updates its motion.

Justification:

The use of this idea is justified because Arena contains all the entities (lights, food, robots) and so it can keep track of what stimuli's state is causing. Once the arena has this information, it can then call the appropriate sensors and determine the reading value. Furthermore, the communication of the value between the sensor and the robot can be carried through the arena since arena has access to both the

robot and its sensors. Doing the implementation this way makes arena the center of everything and it makes logical sense as all the activities and changes are happening within the arena.



UML showing the second design choice for interaction between robot and its sensors.

Justification for choosing 1st choice:

For the purpose of this iteration, I am using the first idea for implementation. This is because in the first idea, the arena is responsible only for communicating the information about the stimuli to the robot and the robot decides whether its sensor has been activated or not and if it has been, the robot appropriately handles it by adjusting its motion and behavior. On the other hand, in the second idea, the arena is accessing the sensor of the robots, getting the sensor reading from the sensors and then communicating it to the robot. This results in unnecessary involvement of arena for communication between the robot and its sensors. This communication is avoided in the first implementation idea since it is done directly and so the first idea (the Observer Pattern) is the appropriate choice for implementing how robot sensors receive information.

Strategy Pattern:

Strategy Pattern is a behavioral software design pattern that allows to select an algorithm at runtime. Instead of implementing a single algorithm directly, the code receives information as to which in a family of algorithms to use. In other words,

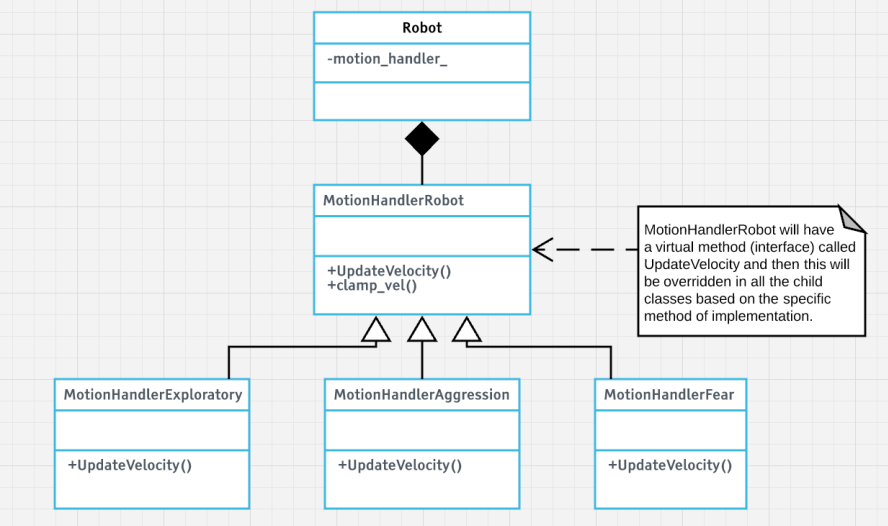
define a family of algorithms, encapsulate each one and make them interchangeable.
This is the second part of the design document that discusses two possible implementations for handling robot motion. The first implementation makes uses of Strategy pattern whereas the other one just creates instances of each kind of motion within the robot. At the end of this part of the document, there is a justification for the design choice made.

First Design Idea (uses Strategy Pattern):

The first implementation idea is to define separate classes for each type of robot motion and have each of them be child class of MotionHandlerRobot. The rationale for creating separate classes is that based on the robot's nature and behavior, its motion can vary so there are separate classes for dealing with each specific motion. Within the robot class, however, there will be an instance of only MotionHandlerRobot to handle all kinds of robot motion. Since MotionHandlerRobot is the parent class of all the different types of robot motion, we can use the instance of MotionHandlerRobot itself, defined within robot, to call the class of the specific motion (which will be one of the child classes of MotionHandlerRobot) that is required, rather than creating instances of all possible motions within robot and then using those instances to call the required class.

Justification:

The use of strategy pattern is justified because a robot has more than one motion and based on robot's behavior and the stimulus, the robot decides which motion to implement and this decision is made at runtime. Since the decision is made at runtime, having just an instance of MotionHandlerRobot and then using this instance to call the specific motion as and when the situation is encountered allows to make use of polymorphism. Hence, due to polymorphism, there is no need to create separate instances for each kind of motion within the robot.



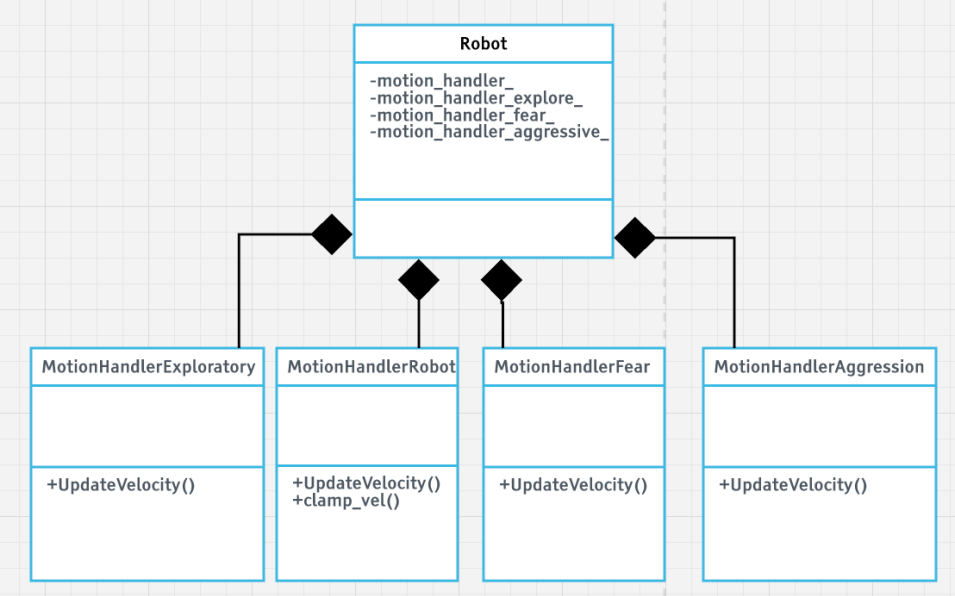
UML Showing the important classes and its attributes and functions necessary for Strategy Pattern idea.

Second Design Idea:

The second implementation idea is to define separate classes for each type of robot motion and then create an instance of each motion within the robot. Then depending on what motion the robot wants to implement based on its behavior, we can use the instance of that particular motion defined within robot directly to carry out the required update in robot's velocity.

Justification:

This idea is justified because all the motions are very specific to robots and robots have the possibility of moving in any of these motions so it is intuitively reasonable to have all instances defined within the robot and then use these instances to carry out the update of motion as and when it is required. Moreover, this method is easy from both the implementation and understanding standpoint as well.



UML showing the second design choice for different kinds of robot motion.

Justification for choosing 1st choice:

For the purpose of this iteration, I am using the first idea for implementation. This is because in the first idea, the robot has only one instance of MotionHandlerRobot and using this instance, any of the robot motions can be implemented. This allows to make effective use of polymorphism (as and when required) and at the same time is less costly on memory. On the other hand, with the second idea for implementing robot motion, creating an instance for each kind of motion can become very costly and memory ineffective as the number of possible motions of a robot increases. For example, if the robot can move a total of 10 different ways but one particular instance of a robot can exhibit only 3 different kinds of motion, using the second design idea will require to create 10 instances for motion within robot but only 3 will be used. But this problem does not arise because of the use of strategy pattern and polymorphism. Hence for this iteration, proceeding with the first idea of implementation makes more logical sense.

Tutorial:

This tutorial details the changes needed to add a new stimulus (water) and a sensor. There are code snippets of iteration 2 implementation provided throughout the

course of this tutorial that details the changes necessary to implement a new stimulus and sensor.

Adding the stimulus (water):

Make the header and source files for water:

- Make the source and header file for water which will be very similar to the current implementation of food. Water like food will be an Immobile Entity, so it will inherit from ArenaImmobileEntity class. In the header file, there will be a constructor and there also will be the headers for all the methods defined in the source file of water.

```
class Food : public ArenaImmobileEntity {
 public:
  /**
   * @brief Constructor.
   *
   * @param params A food_params passed down from main.cc for the
   * initialization of the Food.
   */
  Food();

  /**
   * @brief Reset the Food using the initialization parameters received
   * by the constructor.
   */
  void Reset() override;
```

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

/*****************************************************************
 * Member Functions
 *****************************************************************/
void Food::Reset() {
  set_pose(SetPoseRandomly());
  set_type(kFood);
  set_color(FOOD_COLOR);
  set_radius(FOOD_RADIUS);
  set_captured(false);
} /* Reset */
```

- In particular, the source file for water should have a Reset function that will randomly set the position of the water entity to ensure that at every new game, water is positioned randomly within the arena. Water, like food, is an infinite source and will never get exhausted.

Create the water entity in entity factory class:

- Include the header file for water in the entity factory's header file and also declare the header of the method that adds water entities to the arena. This method's header will be exactly like the method that creates food entities. Also in this class, declare and initialize a private variable "water_count_{0}" that counts the number of water entities in the arena.

```
Food* CreateFood();

/**
* @brief An attempt to not overlap any of the newly constructed entities.
*/
Pose SetPoseRandomly();
```

- In the source file for entity factory, provide the implementation for the method that creates the water entities. This method will again be very similar to the implementation of the CreateFood() method. This class also uses the SetPoseRandomly() method to set randomly the positions of the water entity within arena.

```
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;
}

Pose EntityFactory::SetPoseRandomly() {
  // Dividing arena into 19x14 grid. Each grid square is 50x50
  return {static_cast<double>((30 + (random() % 19) * 50)),
        static_cast<double>((30 + (random() % 14) * 50))};
}
```

Add water entity in the Arena Class:

- Similar to the way entities like food, light and robot are added to the arena, add water entities. This is done in the constructor of the arena class. Also the header for the method that is adding water to the arena must be declared in the arena's header file.

```
Arena::Arena(const struct arena_params *const params)
    : x_dim_(params->x_dim),
      y_dim_(params->y_dim),
      factory_(new EntityFactory),
      robot_(),
      entities_(),
      mobile_entities_(),
      game_status_(PLAYING) {
  AddRobot(kRobot, MAX_ROBOTS);
  AddFood(kFood, N_FOOD);
  AddLight(kLight, params->n_lights);
}
```

- Like food, water is stationary and a robot should pass through water on collision. Furthermore, if the robot is within 5 pixels of the water, water is considered to be consumed.

```
/* Calculates the distance between food and robot to check if robot
* is going to consume that food entity or not.
*/
bool Robot::IsFoodConsumed(ArenaEntity * const other_e) {
    double delta_x = other_e->get_pose().x - (this)->get_pose().x;
    double delta_y = other_e->get_pose().y - (this)->get_pose().y;
    double distance_between_ = sqrt(delta_x*delta_x + delta_y*delta_y);
    return
    (distance_between_ <= ((this)->get_radius() +
                                            other_e->get_radius()+5.0));
}
```

- If the water has the same properties as food in the sense that robot's behavior changes if it goes without water for some time, then the implementation to check for these conditions and ensuring that the robot is behaving appropriately is going to be very similar to the implementation of robot when it is really hungry/starving.

Visual Representation of the Water Entities:

- To show the visual presence of water in the arena, it has to be graphically drawn and this is handled in the DrawEntity method of the GraphicsArenaViewer class. This method is the same that draws the other entities – lights and food – in the arena.

```
void GraphicsArenaViewer::DrawEntity(NVGcontext *ctx,
                                     const ArenaEntity *const entity) {
  // light's circle
  nvgBeginPath(ctx);
  nvgCircle(ctx,
            static_cast<float>(entity->get_pose().x),
            static_cast<float>(entity->get_pose().y),
            static_cast<float>(entity->get_radius()));
  nvgFillColor(ctx,
               nvgRGBA(entity->get_color().r, entity->get_color().g,
                       entity->get_color().b, 255));
  nvgFill(ctx);
  nvgStrokeColor(ctx, nvgRGBA(0, 0, 0, 255));
  nvgStroke(ctx);

  // light id text label
  nvgFillColor(ctx, nvgRGBA(0, 0, 0, 255));
  nvgText(ctx,
          static_cast<float>(entity->get_pose().x),
          static_cast<float>(entity->get_pose().y),
          entity->get_name().c_str(), nullptr);
}
```

- Lastly all the macros that are being used for the water stimulus must be defined in the params.h file and the type kWater must be added to the file entity type file.

```
// food
#define FOOD_RADIUS 20
#define N_FOOD 4
#define FOOD_COLLISION_DELTA 1
#define FOOD_INIT_POS \
  { 400, 400 }
#define FOOD_COLOR \
  { 255, 0, 0 }
#define CHANGED_FOOD_COLOR \
  { 255, 255, 0 }
```

Adding the sensors for water:

Make the header and source files for water sensor:

- Make the source file and header for water sensor which will be very similar to the current implementation of food sensors. Water sensors like food inherit from the Sensor class. In the header file for water sensors. there will be a constructor and also there will be the header for all the methods defined in the source file of water sensor.

```
class FoodSensor : public Sensor {
 public:
  /**
 * @brief FoodSensor constructor initialized with default values from params.h
 */
  explicit FoodSensor(Pose pose) : Sensor(pose) {}

  virtual ~FoodSensor() = default;
  /**
  * @brief Calculates sensor reading value based on the position of the robot
  * from the food.
  */
  void CalculateSensorReading(Pose location_entity) override;
};
```

- The water sensor source file in particular must have a method defined as CalculateSensorReading() which overrides this same method declared in the Sensor class. CalculateSensorReading() takes the position of the water entity as an input and then calculates a sensor reading value based on the position of the entity and the sensors. This sensor value is then used by the robot to implement motion that is a result of the robot being thirsty.

```
void FoodSensor::CalculateSensorReading(Pose location_entity) {
  /* Calculates the food sensor reading based on the distance of the robot
  * from the food entity.
  */
  double reading_ = get_sensor_reading();
  double distance_x_ = pow(get_sensor_position().x - location_entity.x, 2);
  double distance_y_ = pow(get_sensor_position().y - location_entity.y, 2);
  double distance_ = pow(distance_x_ + distance_y_, 0.5) - FOOD_RADIUS;
  reading_ += 1200/(pow(1.08, distance_));
  set_sensor_reading(reading_);
  if (reading_ > 1000)
    set_sensor_reading(1000);
}
```

Creating left and right water sensors:

- The robot needs to have 2 water sensors – left water sensor, right water sensor. Hence in the robot header file, two private variables are declared – left water sensor and right water sensor.

```
// Manages the left and right light sensors.
LightSensor right_light_sensor_;
LightSensor left_light_sensor_;
// Manages the left and right food senors.
FoodSensor right_food_sensor_;
FoodSensor left_food_sensor_;
```

- Both the left and the right water sensor variables are initialized in the robot constructor. Left and right sensor positions are calculated and then these positions are used for initializing the left and right sensors respectively.

```
Robot::Robot(int behavior) :
    motion_handler_(),
    motion_behavior_(this),
    lives_(9),
    time_counter_(0),
    collision_tracker_(false),
    right_light_sensor_(PoseRightSensor()),
    left_light_sensor_(PoseLeftSensor()),
    right_food_sensor_(PoseRightSensor()),
    left_food_sensor_(PoseLeftSensor()),
```

- In the Reset() function within robot, the sensor reading and position has to be reset to prepare for a new simulation and this is done in exactly the same fashion in which reset for the food and light sensors is done.
- In the RobotDecideBehavior() method, after setting the left and right sensors to the correct position, we check if the update is being caused by water, light or food. The implementation of change in motion due to water will be very similar to the change in motion due to food. This is because of the similarity in the impact that these 2 entities have on robot. However, the motion that the robot has when it is thirsty could be something other than aggression, say MotionHandlerThirsty - which could be implemented as Negative, Direct Cross connection of the wheels. In that case MotionHandlerThirsty will have to be called whenever the robot is thirsty. This is done in a similar to fashion to how MotinHandlerAggressive is called when the robot is hungry.

```
case (kFood) : if (IsFoodConsumed(ent)) {  // checking if the food has been
// consumed.
hunger_tracker_ = false;
time_counter_ = 0;
set_color(ROBOT_COLOR);
motion_handler_->set_velocity(5, 5);
} else if (hunger_tracker_) {  // checking if the robot needs to be aggressive
// or not.
left_food_sensor_.CalculateSensorReading(ent->get_pose());
right_food_sensor_.CalculateSensorReading(ent->get_pose());
double l = left_food_sensor_.get_sensor_reading();
double r = right_food_sensor_.get_sensor_reading();
motion_handler_ = new MotionHandlerAggression(this);
motion_handler_->UpdateVelocity(l, r);
}
break;
```

- MotionHandlerThirsty's implementation will be on the same lines as the implementation of motion handler aggression except the formula used to calculate the velocity which will be based on negative direct cross connection. It will be a child class of MotionHandlerRobot and so it will have to override the UpdateVelocity() method defined in the parent class.

```
class MotionHandlerAggression : public MotionHandlerRobot {
 public:
  explicit MotionHandlerAggression(ArenaMobileEntity * ent)
       : MotionHandlerRobot(ent) {}
  MotionHandlerAggression(const MotionHandlerAggression& other) = default;
  MotionHandlerAggression& operator=(const MotionHandlerAggression& other)
                                                              = default;

  /**
  * @brief Aggression behavior of robot is shown when the robot is hungry and
  * Aggression is implemented with positive, crossed sensor-motor connections.
  *
  * This means that there is a positive correlation between the right sensor and
  * left wheel, and between the left sensor and right wheel.
  * Hence based on the left and right sensor reading, right wheel and left wheel
  * velocity is calculated.
  * @param lsreading the Left sensor reading.
  * @param rsreading the right sensor reading.
  */

  void UpdateVelocity(double lsreading, double rsreading)override;
};


 void MotionHandlerAggression::UpdateVelocity(double lsreading,
                                              double rsreading) {
   set_velocity(clamp_vel(ROBOT_MAX_SPEED  * 100 * rsreading/0.5),
                clamp_vel(ROBOT_MAX_SPEED * 100 * lsreading/0.5));
 }
```

Visual Representation of the Water Sensors:

- To show the visual presence of water sensors on the robot in the arena, it has to be graphically drawn and this is handled in the DrawRobot() method of the GraphicsArenaViewer class. All the sensors of the robot are at the same spot so water sensors don't have to be drawn separately because the sensors for light and food are already drawn at the correct positions.

In conclusion, through the course of the first 2 sections of this design document, important design choices have been highlighted and what drove those decisions. In the final part of the document, a feature enhancement tutorial has been provided

that discusses the important steps that need to be carried out in order to add a new stimulus (water) and sensors to the arena.
