Design Documentation of Robot Simulator

Vishnu Arun

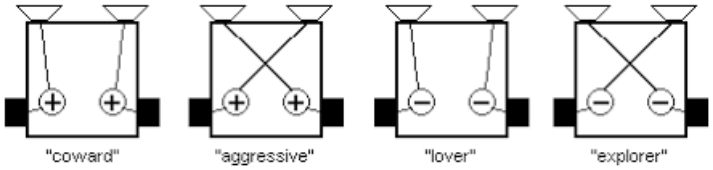
April 11, 2018

University of Minnesota

1 Background & Summary

This design document describes two majors design decisions associated with simulations: Observer Pattern and Strategy Pattern. For each design decision there will be two potential implementations to be used and a discussion on the chosen design. Our simulation consists of multiple autonomous Robots that follow the Braitenberg Vehicles concept. Each Robot will have sensors that react to stimuli placed within the environment as outlined in Figure 1. The last portion of this document will provide a tutorial on how to extend stimuli to the simulation.

Figure 1: Braitenberg Vehicles



2 Observer Pattern

2.1 Introduction

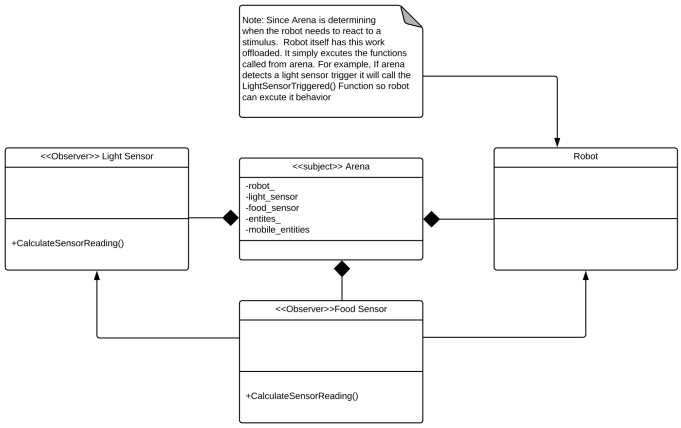
Observer Pattern is a software design pattern where a subject maintains a list of observers who notify the subject when there is any state change. This allows the observer to "subscribe" to a list of subjects without tightly coupling the two objects.

2.2 Original Implementation

Our original implementation consists of the Arena being the subject and the Robot be the observer This idea is similar to a user overseeing the entire environment and making a decision. Arena has access to all entities in the environment and can change state accordingly. Arena can keep track of these changes and, for example, inform the Robot if its sensors have been activated. This idea allows Arena to be a central management for all the entities.

This implementation is carried out by the UpdateTimeStep() function in Arena. At each time-step we loop through all robots and all other entities and determine whether a sensor has been activated. The arena will call CalculateSensorReading() on the sensors to determine reading values. The arena can then host the communication between the robot and the sensors. The robot will know which sensor was triggered and behave accordingly. This implementation is represented as a UML in figure 2.

Figure 2: UML Diagram for Original Implementation



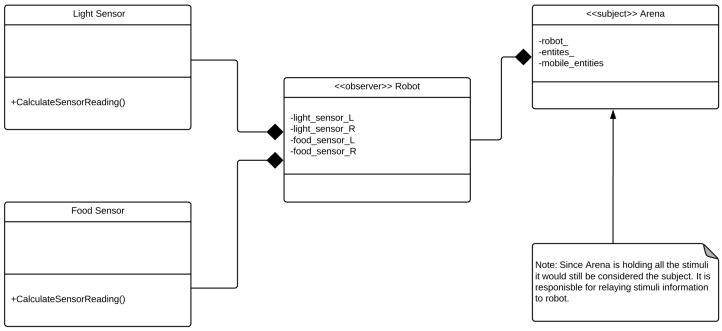
2.2.1 Justification of Design

This design is valid because Arena would be a good subject and robot would be a good observer. Arena has information on all its entities and can communicate effectively to the robot based on which sensor has been activated by the robot. The arena can also host the communication between sensor and robot. All sensors activities will thus pass exclusively through arena.

2.3 Alternative Implementation

Our alternative implementation consists of the Robot being the observer and the stimulus being the subject through Arena. This idea is similar to how a physical robot would react to stimuli. At each time step the Robot will get readings from all stimuli in the environment then choose an action based on the reading values and stimuli type. If the readings are sufficient the Robot will change behaviors and portray a behavior based on the stimuli. The Robot will need to receive all stimuli from the Arena itself since Arena stores all entities ultimately causing Arena to be the subject.

Figure 3: UML Diagram for Alternative Implementation



2.3.1 Justification

This approach would be valid because Arena would not have access to the Robot sensors themselves allowing for looser coupling. This implementation is also intuitive because the Robot should have the task of determining the impact of a sensor reading and applying it to its own behavior. Arena simply relays the stimuli to the Robot itself for processing. This is detailed on figure 3.

2.4 Justification for choosing Alternative Implementation

The Alternative Implementation is more beneficial than the Original Implementation because the Arena should exhibit looser coupling to the Robot class. In the original design the Arena would do most of the processing work and relay a

behavior to the robot itself. This is not intuitive because the Robot is not fully deciding its own behavior. The Alternative Implementation allows the Arena to simply relay the stimuli to the Robot. The Robot would then calculate readings and behave accordingly. The Robot is therefore in charge of its own behavior and motions.

3 Strategy Pattern

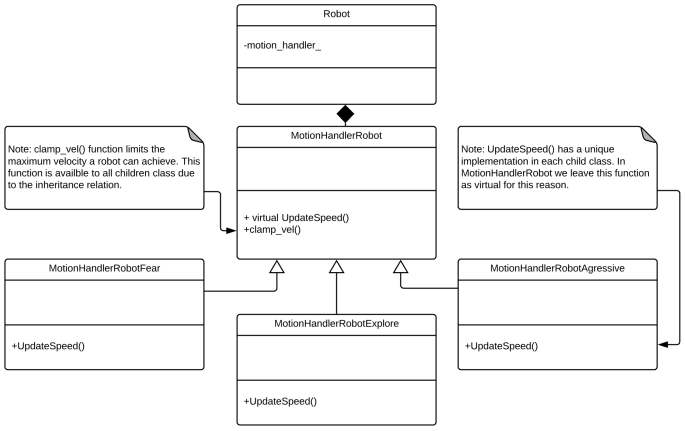
3.1 Introduction

Using Strategy Pattern, a class behavior or its algorithm can be changed at runtime. We create context object whose behavior varies. This behavior is relayed by objects that represent various strategies. The strategy object can change the executing algorithm based on the context. This concept is similar to a class having access to a family of algorithms that it can choose based on context. Each algorithm is encapsulated allowing easy switching between algorithms.

3.2 Original Design

In our Original Design we create separate classes for each robot behavior. These classes extend MotionHandlerRobot class by uniquely defining a a virtual function in each class. This Parent-Child Relationship allows us to use one instance of MotionHandlerRobot to call a specific behavior. This design therefore uses strategy pattern as detailed in Figure 4.

Figure 4: UML Diagram for Original Design



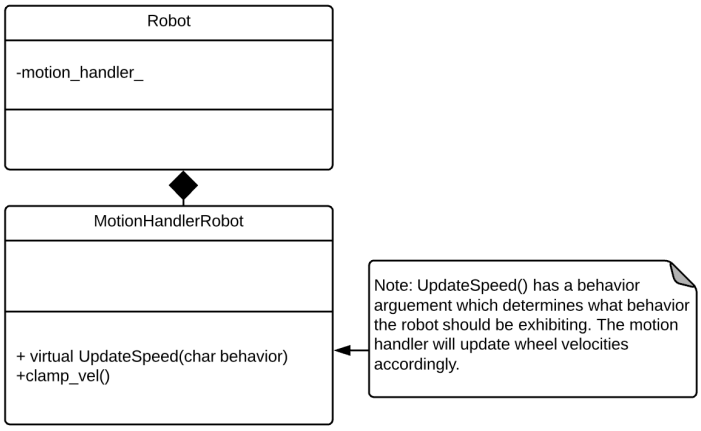
3.2.1 Justification

This approach would be valid because we have multiple behaviors the Robot would need to exhibit based on stimuli. Our approach extends one instance of MotionHandlerRobot thus adding a new behavior would require less code change (we detail this process in the tutorial section). This concept exhibits polymorphism since the decision on behavior is made at runtime and separate class instance for each behavior are not required.

3.3 Alternative Design

Our alternative design requires MotionHandlerRobot to implement UpdateVelocity() with an additional flag variable. This flag would be a char corresponding to fear ('F'), explore ('E'), etc. The UpdateVelocity() method would have each behavior hard-coded into it with a switch statement depending on the flag passed in. The flag would be determined by the Robot stimuli and ID. This process is detailed in figure 5.

Figure 5: UML Diagram for Alternative Design



3.3.1 Justification

This design would centralize all motion behaviors to one class. The robot would simply access this single instance of MotionHandlerRobot and pass in a specified behavior. All motion updates would happen exclusively through this class allowing the code for this behavior to be easily accessible.

3.4 Justification for Choosing the Original Design

Our Original Design uses the Strategy Pattern concept. This allows behaviors to be derived from a single instance of MotionHandlerRobot class. We can eas-

ily add a new behavior by adding a new behavior class and properly initializing. The Alternative Design requires a large amount of hard coding into the same function. If we were to remove a behavior in the future it would require refactoring a large amount of code, only a portion of which pertains to the original behavior. If we wanted to add a large number of behaviors in our alternative design we would need to document our flag variables for code readability. Our original design also takes advantage of polymorphism where multiple instance of behaviors will not be created at run time. This approach is highly memory efficient. We select the Original Design for these reasons.

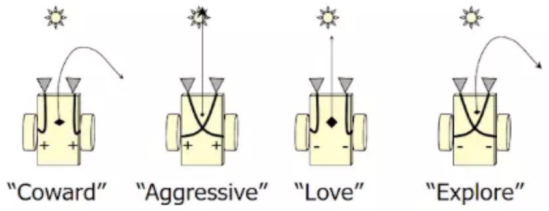
4 Tutorial: Feature Enhancement

4.1 Introduction

In this section we consider a scenario where a developer needs to add a new stimuli (e.g. water) and corresponding sensor. We will present code blocks and descriptions of the scenarios.

4.2 Scenario

Figure 6: Our Braitenberg Vehicles exhibit 4 behaviors based on stimuli.



1. Coward
    - The sensor-wheel relationship is correlated and the entity moves away from stimuli in attempt to avoid.
2. Aggressive
    - The sensor-wheel relationship is crossed and the entity attempts to move towards the stimuli.
3. Love
    - The sensor-wheel connection makes the entity follow the stimuli and stay close to it.
4. Explore

- The sensor-wheel connection makes the entity roam freely. It roams for stimulus and allows vehicles to follow.

4.3 Tutorial

In this tutorial we detail the changes necessary to implement a new stimulus and sensor. We will provide code examples for food stimuli and context.

4.3.1 Adding New Stimulus (Water)

1. Make header and source file for water.

   - Our Header File will create constructors and methods related to water. Water will inherit from ArenaImmobileEntity class. We detail some example methods such as IsCaptured() which indicates whether the Stimili has been captured.

```
class Food : public ArenaImmobileEntity {
#ifndef SRC_FOOD_H_
#define SRC_FOOD_H_


#include "src/arena_immobile_entity.h"

 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;

    /**
   * @brief Getter for captured_, which is the state of
   * the food
   *
   * @return true if captured.
   */
  bool IsCaptured() const { return captured_; }

  /**
   * @brief Setter for captured_, which is the state of
   * the food
   */
  void set_captured(bool state) { captured_ = state; }
  }
```

```
#endif  // SRC_FOOD_H_
```

- In the source file we will have a reset method to reset all attribute upon simulator restart. We must also ensure the stimuli restarts in a random position so we use a function SetPoseRandomly() that achieves this. This function generates random valid positions within the Arena.

```
#include "src/food.h"
#include "src/params.h"

/*****************************************************
 * Constructors/Destructor
 *****************************************************/
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_color(FOOD_COLOR);
  set_radius(FOOD_RADIUS);
  set_captured(false);
} /* Reset */
```

2. Create Water entity within Entity Factory Class

- Include header file of water class and make private variable to create water. Our EntityFactory class also includes the SetPoseRandomly() function as detailed previously.

```
/**
  * @brief CreateFood called from within CreateEntity.
  */
  Food* CreateFood();

  Pose SetPoseRandomly();
```

- In the source file of Entity Factory, we need to create a water object with the necessary attributes such as entity count. This count will allow the simulation to understand how many entities are created in total. For example, in Arena's UpdateTimestep() function we loop through all entities to determine whether the Robot is colliding

```
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. Add Water entity to Arena Class

   - Add amount of water entities in constructor.

```
Arena::Arena(const struct arena_params *const params)
    : x_dim_(params->x_dim),
      y_dim_(params->y_dim),
      factory_(new EntityFactory),
      entities_(),
      mobile_entities_(),
      robot_(),
      game_status_(PLAYING) {
  AddRobot(N_ROBOTS);
  AddEntity(kFood, 3);
  AddLight(params->n_lights);
}
```

   - We must declare the Add method in the header file of Arena in order to properly use this method in the Arena source code.

```
void AddLight(int quantity);
```

   - We can also declare methods in robot to check if water is consumed. This is similar to food consumed. In our food implementation we use a consumed_food_ flag to dictate how the robot should behave. The flag can be passed into timestep update to remove the robots ability to exhibit certain behaviors, etc.

```
void Robot::CheckHungry(Pose ent) {
  food_sensor_L.CalculateReading(ent);
  food_sensor_R.CalculateReading(ent);
  double fsr_L = food_sensor_L.get_reading();
  double fsr_R = food_sensor_R.get_reading();
  if (((fsr_L > 500) | (fsr_R > 500))) {
    consumed_food_ = true;
  }
}
```

4. Add functionality to draw entity in GraphicsArenaViewer Class.

   - We use NVG methods to draw water entity and add color. This physically draws the entity in the graphics of this simulation.

```
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);
}
```

- The size and colors derive from the Params.h file where other default settings are present

```
// light
#define LIGHT_POSITION \
  { 200, 200 }
#define LIGHT_RADIUS 30.0
#define LIGHT_MIN_RADIUS 10
#define LIGHT_MAX_RADIUS 50
#define LIGHT_COLOR \
  { 255, 255, 255 }
```

4.3.2 Adding New Sensors (Water Detection sensors)

1. Make Source and Header File for Water Sensors

   - Our headers file will follow the same concept as food sensor. This class would inherit from Sensor class. Our header file will include a constructor related to the sensor initialization and methods specific to the sensor.

```
#ifndef SRC_FOOD_SENSOR_H_
#define SRC_FOOD_SENSOR_H_

/*Includes
*/
#include "src/sensor.h"

class FoodSensor : public Sensor {
   public:
      /**
      *@brief FoodSensor constructor
      */

      explicit FoodSensor(Pose pose) : Sensor(pose) {}
```

```
    virtual ~FoodSensor() = default;
    // @brief Calculate Reading function using entity
    void CalculateReading(Pose ent) override;
};

#endif  // SRC_FOOD_SENSOR_H_
```

- Our Water Sensor source file will need an implementation of CalculateReading() unique to the properties of the water entity. This function will calculate the distance between the sensor and the water entity in order to generate a reading. It takes in the Pose of the entity as an argument in order to calculate the reading. We can use FoodSensor implementation of this function as an example. In FoodSensor we calculate the reading based on the Euclidean distance of the sensor and the stimuli.

```
void FoodSensor::CalculateReading(Pose ent) {
  double reading = get_reading();
  reading = (1200.0 / pow(1.08,
  (pow((pow((get_position().x - ent.x), 2.0) +
   pow((get_position().y - ent.y), 2.0)), 0.5) -
   FOOD_RADIUS)));
  if (reading > 1000.0) {
    set_reading(1000.0);
  } else {
    set_reading(reading);
  }
}
```

2. Create Sensor Instances in robot and configure behaviors

- Each robot needs a left and right sensor placed on either side in order to conform to Braitenberg vehicles behaviors concept. We declare these sensor instances as private in the robot header file.

```
FoodSensor food_sensor_R;

FoodSensor food_sensor_L;
```

- These sensors must be initialized in robots constructor. We pass in a UpdateSensorPosition() method which calculates the correct position of the sensor depending whether it is left or right.

```
Robot::Robot() :
    motion_handler_(new MotionHandlerRobot(this)),
    motion_behavior_(this),
    lives_(9),
    arc_reverse_(false),
    arc_time_(0),
    consumed_food_(false),
    hungry_timer_(HUNGRY_TIME),
    light_sensor_R(UpdateSensorPosition('R')),
    light_sensor_L(UpdateSensorPosition('L')),
    food_sensor_R(UpdateSensorPosition('R')),
```

```
    food_sensor_L(UpdateSensorPosition('L')) {
  set_type(kRobot);
  set_color(ROBOT_COLOR);
  set_pose(ROBOT_INIT_POS);
  set_radius(ROBOT_RADIUS);
```

- Our Update Sensor position method currently places the sensors 40 degrees to either side of the robots headings. It decides the side based on the argument passed in. If the incorrect argument is passed in we will get an exception.

```
Pose UpdateSensorPosition(char type) const {
    double pi = atan(1)*4;
    if (type == 'R') {
      double thetaL_ = (this)->get_heading() +
      (-40.0*pi/180.0);
      double R_x_ = (this)->get_radius()*cos(thetaL_) +
      (this)->get_pose().x;
      double R_y_ = (this)->get_radius()*sin(thetaL_) +
      (this)->get_pose().y;
      return {R_x_, R_y_};

    } else if ( type == 'L' ) {
      double thetaR_ = (this)->get_heading() +
      (40.0*pi/180.0);
      double L_x_ = (this)->get_radius()*cos(thetaR_) +
      (this)->get_pose().x;
      double L_y_ = (this)->get_radius()*sin(thetaR_) +
      (this)->get_pose().y;
      return {L_x_, L_y_};
    } else {
      throw std::invalid_argument( "Received␣Incorrect
␣␣␣␣␣␣Char" );
      return {0.0, 0.0};
    }
  }
```

- Our Robots Reset() function must reset all sensor positions and readings on restart. We call Reset() on each sensor in order to accomplish this.

```
void Robot::Reset() {
  set_pose(SetPoseRandomly());
  motion_handler_->set_max_speed(ROBOT_INIT_SPEED);
  motion_handler_->set_max_angle(ROBOT_INIT_SPEED);
  sensor_touch_->Reset();
  set_lives(9);
  ArcTime_ = 0;
  ArcReverse_ = false;
  light_sensor_L.Reset();
  light_sensor_R.Reset();
  food_sensor_R.Reset();
  food_sensor_L.Reset();
  set_color(ROBOT_COLOR);
  ConsumedFood_ = false;
  HungryTimer_ = HUNGRY_TIME;
} /* Reset() ;*/
```

- In Our CheckStimuli() Function we decide which behavior to execute based on stimuli type and robot sensor readings. We will use the existing instance of MotionHandlerRobot and extend the class to the correct subclass (e.g. MotionHandlerRobotAggresive()) based on the stimuli. Due to the implementation of Strategy pattern, we can easily change the behaviors by extending other subclasses.

```
if (type == (kFood) {
  food_sensor_L.CalculateReading(ent);
  food_sensor_R.CalculateReading(ent);
  double fsr_L = food_sensor_L.get_reading();
  double fsr_R = food_sensor_R.get_reading();
  if (((fsr_L > 500) | (fsr_R > 500))) {
    ConsumedFood_ = true;
  }
  if (HungryTimer_ < 0) {
    MotionHandlerRobot *RobotAggressive =
    new MotionHandlerRobotAggressive(this);
    RobotAggressive->UpdateSpeed(fsr_L, fsr_R);
  }
  }
```

- Our Header file for the MotionHandlerRobotAggressive class defines an aggressive behavior that inherits from MotionHandlerRobot. This Behavior is unique due to its UpdateSpeed() function which defines the behavior itself.

```
#ifndef SRC_MOTION_HANDLER_ROBOT_AGGRESSIVE_H_
#define SRC_MOTION_HANDLER_ROBOT_AGGRESSIVE_H_

#include "src/motion_handler_robot.h"

class MotionHandlerRobotAggressive :
public MotionHandlerRobot {
 public:
  explicit MotionHandlerRobotAggressive(
  ArenaMobileEntity * ent)
      : MotionHandlerRobot(ent) {}

  MotionHandlerRobotAggressive(const
  MotionHandlerRobotAggressive& other) =
  default;
  MotionHandlerRobotAggressive& operator=
  (const MotionHandlerRobotAggressive& other)
  = default;



Pose UpdateSpeed(double lsr_L, double lsr_R) override;
};


#endif  // SRC_MOTION_HANDLER_ROBOT_AGGRESSIVE_H_
```

- Our source code for the MotionHadlerRobotAggressive class contains the specific implementation of UpdateSpeed().

```
Pose MotionHandlerRobotAggressive::UpdateSpeed(
double lsr_L, double lsr_R) {
  double constant_ = 0.001 * ((lsr_R > lsr_L) ? lsr_R :
  lsr_L);
  return Pose(clamp_vel(5 + constant_*lsr_R), clamp_vel(5 +
  constant_*lsr_L));
  }
```

3. Draw sensors on robot using NVG

   - We use GraphicsArenaViewer class using DrawRobot() method to draw sensors visually on robot. Since all sensors are superimposed on robot, we only draw one pair of sensors to represent all three sensors. We can set the size and colors directly here or use the Params.h file. These sensors get redrawn every time the robot gets redrawn. We use UpdateSensorPosition() to calculate the correct position for the sensors.

```
// draw left sensors
 nvgBeginPath(ctx);
 nvgCircle(ctx, static_cast<float>(
 robot->UpdateSensorPosition('L').x),
 static_cast<float>(robot->UpdateSensorPosition('L').y),
 static_cast<float>(SENSOR_GRAPHICS_RADIUS));

 nvgFillColor(ctx, nvgRGBA(100, 100, 100, 255));
 nvgFill(ctx);

 nvgStrokeColor(ctx, nvgRGBA(0, 0, 0, 255));
 nvgStroke(ctx);

 // draw right sensors
 nvgBeginPath(ctx);
 nvgCircle(ctx, static_cast<float>(
 robot->UpdateSensorPosition('R').x),
 static_cast<float>(robot->UpdateSensorPosition('R').y),
 static_cast<float>(SENSOR_GRAPHICS_RADIUS));

 nvgFillColor(ctx, nvgRGBA(100, 100, 100, 255));
 nvgFill(ctx);

 nvgStrokeColor(ctx, nvgRGBA(0, 0, 0, 255));
 nvgStroke(ctx);
```