Design Document

Introduction

It is a Braitenberg Vehicles simulator. The robots simulate Braitenberg vehicles which sense light and food in arena.

There are four kinds of vehicle patterns: Coward, Love, Explore and Aggressive. In the stimulation, five robots exhibit "coward" towards light and five robots exhibit "explore" towards light. All the robots exhibit "aggressive" towards food. The robots have four status-not hungry, hungry, really hungry and starve. When the simulation starts and after a robot consuming food, the robot is not hungry. When the robot is not hungry, it moves according to light sensors. When the robot is hungry, it moves according to either food sensor or light sensor, depending on which reading is larger. In other words, it reacts to the closer stimuli. When the robot gets really hungry, it ignores the light and go aggressively towards food. The simulation terminates when any of the robots starve.

The food are immobile stimuli. All the other entities can pass through food.

The light are mobile stimuli. They can pass through everything except another light and the wall. We need to handle collision between stimuli itself and between stimuli and the wall.

Design Decision

Observer Pattern

One way to implement observer pattern is to take all stimuli (light and food) as subjects, and all sensors as observers. Each robot will have both light and food sensors at the same place in addition to touch sensors from iteration 1. When simulation starts, these sensors will register as observers in Arena for their respective stimuli. We have 10 robots, 4 foods and 4 lights, thus, 10 light sensors will register as a sensors' list in each four light and 10 food sensors will register as a sensors' list in each four foods.

When stimuli update its position, it would notify each sensors of the new position to update the readings. Each of the sensors accumulates the reading that is calculated by the position of the stimuli it is observing. The lights will change position with each time step, so the light sensors will be notified each timestep. Although the food is immobile stimuli, food sensor reading still need to be calculated each timestep because the sensor position changes. It is also convenient for future enhancement.

The sensor has two main functions, update the position of themselves and calculate

the reading. The sensor gets the position and radius of the stimuli from notification and get the position of the robot by calling the update in the robot class. The reading then will be got by the robot in order to determine its motion.

Another way to implement observer pattern is to take sensors as the observer and Arena itself as the subject. Arena keeps track of all entities. All information is measured and got in Arena. For example, the distance between vehicle and stimulus will be calculated in the class Arena. Arena will get the position of stimulus and measure the state of vehicle.

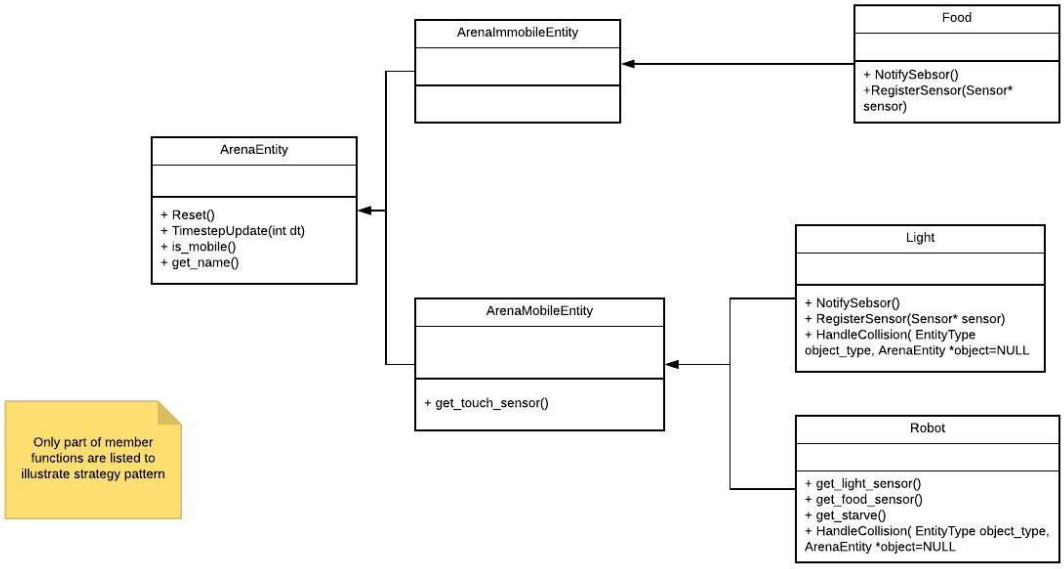
On each timestep in Arena, instead of in light and food, we check each entity in a loop. For every robot entity, if the second entity is light, the arena would calculate the light reading of this robot according to the distance between the robot's sensors and the light; If the second entity is food, the arena would calculate the food reading of this robot. Then Arena would pass the sensor the reading value which determine the motion of robot.

In this iteration, I used the first design. Although the data and stimuli are easy to get in Arena, we have to change much code in arena if more stimuli need to be added. However, in the first design, we can add more classes without changing the existing code. Besides, it makes more sense that the stimuli notify the sensor when its position changes.

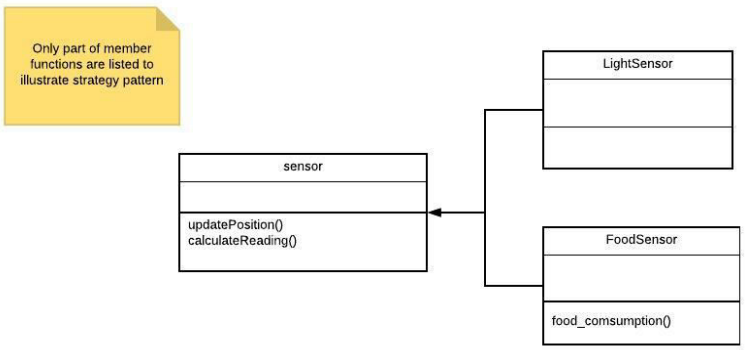
Strategy Pattern

Strategy pattern can be utilized to implement how arena entities handle their motion. We have light, robot, which is mobile entity and base, which is immobile. All of the entities have to be able to reset, update each timestep and get their names. These are the behaviors that stays the same. The other behaviors like handle collision, get starve will change from entity to entity.

We use arena mobile entity and arena immobile entity as an interface rather than implementation directly in light and robot. To create a specific implementation of an entity, we will extend the Arena Entity class and the Arena Immobile or Mobile Entity class, and in this way have access to the is_mobile, get_touch_sensor functions. This successfully separates the different behavior of arena entities. The diagram below gives an overhead view of how we implement the strategy pattern.



Another place to implement strategy pattern is on sensor. We can create a sensor interface from which light sensor and food sensor composed. The parent sensor class has the function update() and calculateReading() which is shared by light sensor and food sensor. food consumption is the different behavior from light sensor. The diagram below gives an overhead view of how we implement the strategy pattern on sensors.



I choose the first implement which use strategy pattern to implement arena entity. In this way we I didn't use strategy pattern to implement sensor because the behavior of the light sensor and food sensor is very similar, I can just put them in one sensor class where have all the functions they need.

Feature Enhancement

To Add a New Stimulus to the Arena

- Add a new class for the stimulus.
- Add motion handler for this stimulus if it is mobile.
- Add functions that can notify and register sensor in the new stimuli class.

```
void Light::NotifySensor() {
  for ( auto &s : sensors_ ) {
    s->calculateReading(get_pose(), get_radius());
  }
}

void Light::RegisterSensor(Sensor *sensor) {
  sensors_.push_back(sensor);
}
```

- Add parameters for this stimulus.

```
// food
#define FOOD_RADIUS 20
#define FOOD_COLLISION_DELTA 1
#define FOOD_INIT_POS \
  { 400, 400 }
#define FOOD_COLOR \
  { 153, 230, 0 }
```

- Update entityfactory to handle stimulus.

```
ArenaEntity* EntityFactory::CreateEntity(EntityType etype) {
  switch (etype) {
    case (kLight):
      return CreateLight();
      break;
    case (kFood):
      return CreateFood();
      break;
    default:
      std::cout << "FATAL: Bad entity type on creation\n";
      assert(false);
  }
  return nullptr;
}
```

```
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;
}
```

- Add the stimulus to the Arena's constructor.

```
    AddEntity(kFood, 4);
```

- Register the sensor as an observer.

```
  // register sensor
  for (auto ent1 : robot_) {
    for (auto ent2 : entities_) {
      if (ent2->get_type() == kLight)
        dynamic_cast<Light*>(ent2)->RegisterSensor(ent1->get_light_sensor());
      if (ent2->get_type() == kFood)
        dynamic_cast<Food*>(ent2)->RegisterSensor(ent1->get_food_sensor());
        // dynamic_cast<Food*>(ent2)->NotifySensor();
    }
  }
}
```

- Add a sensor type in sensor to handle extra requirement.

```
// The stimuli calls this function to push notifications to its sensors
void Sensor::calculateReading(Pose p, double stimuliraius) {
  if (stimuli_type_ == kFood) {
    double dis = sqrt((p.x - robot_pose_.x) * (p.x - robot_pose_.x) +
     (p.y - robot_pose_.y)*(p.y - robot_pose_.y)) - stimuliraius;
    bool food_consumption_temp = false;
    if (dis <= 5)
      food_consumption_temp = true;
    food_consumption_ = food_consumption_ || food_consumption_temp;
  }
```

- Add sensor to robot in the robot class.

```
  private:
   // Manages pose and wheel velocities that change with time and collisions.
   MotionHandlerRobot motion_handler_;
   // Calculates changes in pose foodd on elapsed time and wheel velocities.
   MotionBehaviorDifferential motion_behavior_;
   // Lives are decremented when the robot collides with anything.
   // When all the lives are gone, the game is lost.
   int lives_;

   Sensor* light_sensor_;
   Sensor* food_sensor_;

```

- reset the reading calculated by this stimulus in each timestep of robot.

```
  food_sensor_->set_left_reading(0);
  food_sensor_->set_food_consumption(false);
  food_sensor_->set_right_reading(0);
  food_sensor_->update(get_pose());
} /* TimestepUpdate() */
```

- Include the reading of this sensor to handle robot motion in each timestep.

```
void Robot::TimestepUpdate(unsigned int dt) {
  time_count_++;

  // set status of robot
  if ( time_count_ > 300 )
    hungry_ = true;
  if ( time_count_ > 1200 )
    really_hungry_ = true;
  if ( time_count_ > 1500 )
    starve_ = true;
  if ( food_sensor_->get_food_consumption() ) {
    hungry_ = false;
    really_hungry_ = false;
    starve_ = false;
    time_count_ = 0;
  }

  if (starve_) {
    set_color({128, 0, 0});
  } else if (really_hungry_) {
    set_color({255, 51, 51});
    motion_handler_.UpdateVelocitybySensor(food_sensor_);
  } else if (hungry_) {
    set_color({255, 128, 128});
    if (food_sensor_->get_left_reading() > light_sensor_->get_left_reading()
     ||food_sensor_->get_right_reading() > light_sensor_->get_right_reading()) {
         motion_handler_.UpdateVelocitybySensor(food_sensor_);
    } else {
        motion_handler_.UpdateVelocitybySensor(light_sensor_);
    }
  } else {
    set_color(ROBOT_COLOR);
    motion_handler_.UpdateVelocitybySensor(light_sensor_);
  }
```