Da Song

CSCI 3081

November 28, 2017

Software Design Document

1. Introduction

1.1 Purpose

This software design document describes the architecture and system design of a rudimentary robot simulator in which robot behavior is visualized within a graphics window.

1.2 Scope

This project develops a rudimentary robot simulator in which robot behavior is visualized within a graphics window. The exact use of the application is yet to be determined. It is used more like a video game in which users control the robots. Alternatively, it might be a test environment for experimenting with autonomous robot control using sensor feedback, or it might be a visualization of real robots operating in a remote location. Robot will be moving towards autonomous and have intelligent behavior. And the user will control the player to freeze all robots at the same time before running out of energy to win the game.

1.3 Overview

This document will introduce the purpose of the project, system architecture, component design, user interface and the organization of the project.

1.4 Reference Material

https://github.umn.edu/umn-csci-watters-3081F17/public-class-repo/blob/master/project/Iteration2Requirements.md

1.5 Definitions and Acronyms

ArenaEntity: ArenaEntity includes the most basic attributes of entity such as radius, position and color.

ArenaMobileEntity: It contains moving entities such as Robot, SuperBot, Player and HomeBase.

ArenaImmobileEntity: it has fixed entities which cannot move like obstacles, recharge station.

arena_params: it includes Entities' parameters populating the arena. The entities' parameters are presented by struct. Following stuff are the elements of struct arena_params:

```
struct robot_params robots[MAX_ROBOTS];
struct player_params player;
struct arena_entity_params recharge_station;
struct home_base_params home_base;
struct arena_entity_params obstacles[MAX_OBSTACLES];
 size_t n_obstacles;
 size_t n_observers;
size_t n_robots;
uint x_dim;
uint y_dim;
```

event_xxx: xxx means various events' type like DistressCall, Proximity, Collision and so on. They represent those event which sensors need to accept.

sensor_xxx: xxx means various sensors' type reflecting to the above events.

motion_handler_xxx: xxx means different ArenaMobileEntities. MotionHandler's main job is to translate the directional commands from the user into the appropriate differential drive wheel speeds.

2. System Overview

In this project, there are a Player(blue), some autonomous Robots(Green), some SuperBot(black), a HomeBase(red), a RechargeStation(grey) and several obstacles(white) displayed in the graphic view. The objective of the game is for the player to freeze all robots at the same time before running out of energy. Energy is depleted constantly and is depleted even more when moving or when it bumps into obstacles, but it can renew its energy by going to the charging station. Autonomous robots that use sensors to avoid objects will move around the environment interfering with play. If the user-controlled robot collides with an autonomous robot, it will freeze (i.e. stop moving), but if another autonomous robot collides with a *frozen* robot, it will move again. A frozen robot will emit a distress call, which when detected by another autonomous robot, allows the moving robot to ignore its proximity sensor and bump into the frozen robot. Furthermore, SuperBots roam the arena and if they collide with the Player, the Player freezes for a fixed period of time. SuperBots become SuperBots by an ordinary Robot colliding

with home base. In this project, we write unit tests with Google Test, use the Google style guide, and maintain our UML and doxygen documentation.

3. System Architecture

In this project, there are three main modulars, Robot Arena, Graphic Environment and GUI and User Input.

- Robot Arena: it has a user-controlled player, autonomous robots, obstacles, home base, and a charging station. The intent is for the player to move around the space searching for robots to freeze. As it moves, its battery is depleted. The player can refill its battery by touching the recharge station. Autonomous robots have 2 proximity sensors, a distress call receiver, and entity type sensor, and a touch sensor. Plus, we need to obey the observer pattern. Arena is the subject in observer pattern. ArenaMobileEntity are the observers. To follow the design pattern, we build ArenaEntity first. It is obviously that there are two types of ArenaEntity. One is ArenaMobileEntity, the other one is ArenaImmobileEntity. Each of them is a smaller modular that constructs certain kind of object and its properties. ArenaMobileEntity contain MotionHandler and MotionBehavior to control and track their motion. They will also include some kinds of sensors as their components as needed. Additionally, SuperBot is Robot too. However, we use different MotionHandler to control them, which can help us implement the observer pattern. As far as Sensor, is consisted by general sensor, proximity sensor, touch sensor, distress sensor and entity type sensor. General sensor is an abstract class, Each sensor should Accept an event(any type of event) from

Arena directly. To following the observer pattern, we assume Arena is the subject, Robot is the observer. The only thing Arena knows about a Robot is that it implements a certain interface. And we can add new robots at any time. Thus, we ass three functions in Arena, RegisterRobot(Robot * r), RemoveRobot(Robot * r), NotifyRobot(Event * e). Proximity sensor can sense objects in its limited cone. Its output is the distance to an obstacle. Distress sensor senses the distress signal in its range. Sensor output is 1 for a sensed call and 0 for none. Entity type Sensor senses the type of the entity emitting the signal in its range. Sensor output is the enumerated entity type. Touch Sensor senses the collision. Sensor output is 1 for a collided and 0 for none. And I create a EventBaseClass. EventProximity, EventDistressCall, EventTypeEmit and EventCollision are all derived from EventBase Call. Each event class has a constructor which can construct an event object when this event happens. Finally, in Arena, it contains and controls all entities in this project. It will update the observers(all entities) directly by checking those events whether happened.

- Graphic Environment: The graphics environment consists primarily of a single window with robots, obstacles, home base, and charging station. All objects in the environment will be drawn as circles, which greatly simplifies collision detection. You are welcome to add graphics enhancements, such as color, text, or decorations provided it does not interfere with required functionality.
- GUI and User Input: In iteration1, we use hard code to write a main method. Each time we want to make some changes of numbers of entities, radius or positions, we had to each the code and recompile the program. But now, we can write those data in a file and use I/O method to read the file and create the view. Just changed the

data in that file, we can change numbers of entities, radius or positions. Plus, we create UI buttons like Restart and Pause and we can control play by pressing arrow keys.

When the program starts running, Arena will be initialized with all of entities contained in the input file and registered those to observers. ArenaViewer would draw those stuffs. While there are some new events happened, Arena will notify the observers to updated. The observers will accept the information and update themselves.

3.2 Decomposition Description

This design document described all requirements in Iteration2. Following things are the details of each class with some questions can explain more specifically.

Sensors:

Q1: How does the sensor information effect entity behavior?

Q2: Where is that information used to effect entity behavior?

Q3: What information should the sensor have for the entity to appropriately react?

Touch Sensor:

Q1:

- Robot/Robot: If the output is 1 and one of the Robot is frozen, unfreezes that Robot and the other Robot would bounce off.

- Robot/SuperBot: If the output is 1 and the Robot is frozen, unfreezes that Robot and the SuperBot would bounce off.
- Robot/HomeBase: If the output is 1 and the Robot is not frozen, Robot will become SuperBot then both SuperBot and HomeBase bounced away. Or if the output is 1 and the Robot is frozen, Robot will unfreeze and become SuperBot then both SuperBot and HomeBase bounced away.
- Robot/Player: If the output is 1 and the Robot is not frozen, Robot will be frozen and Player will bounce away.
- Robot/RechargeStation: If the output is 1, Robot simply bounce away.
- Robot/Obstacles: If the output is 1, Robot simply bounce away.
- Robot/Walls: If the output is 1, Robot simply bounce away.
- Player/SuperBot: If the output is 1, Player will freeze 3 seconds. SuperBot will bounce away.
- Player/HomeBase: If the output is 1, Player simply bounce away.
- Player/RechargeStation: If the output is 1, Player simply bounce away and charge becomes full.
- Player/Obstacles: If the output is 1, Player simply bounce away.
- Player/Walls: If the output is 1, Player simply bounce away.
- SuperBot/RechargeStation: If the output is 1, SuperBot simply bounce away.
- SuperBot/Obstacles: If the output is 1, SuperBot simply bounce away.
- SuperBot/Walls: If the output is 1, SuperBot simply bounce away.

Q2:

It can check whether or not the collision event happened. If happened, then told motion handler that the entity collided and changing heading_angle_.

Q3:

To sensed the collision and told motion handler to handle this collision, sensor_touch need to contain the point of contact and the angle of contact. Once sensor_touch obtain the information from the event_collision, motion handler can get the informations from sensor_touch.

Entity Type Sensor:

Q1:

- Robot: If the output is kRobot and that robot is frozen, proximity sensor will not be activated. Otherwise, it will not affect.
- SuperBot: If the output is kRobot and that robot is frozen, proximity sensor will not be activated. If the output is kPlayer, proximity sensor will not be activated.
- Player: Nothing happened.
- HomeBase: Nothing happened.
- Recharge Station: Nothing happened.

Q2:

Information of entity type sensor will be used inside functions of proximity sensor. If there is something in the range of proximity sensor, check the type of object using entity type sensor.

Q3:

To sensed the entity type, this Sensor need to know the radius and positions of sensed entity and sensor entity. Because we need to determine whether the former one in the later one's range

Proximity Sensor:

Q1:

- Robot: If SensorReading does not return -1 and the Robot do not have Distress-Call signal, then Robot's proximity would be activated. After it is activated, the output is the distance between them. And they will avoid to hit each other. If another entity is in the left sensor, the robot will turn right. Else if another entity is in its right sensor, robot will turn left. Else if the entity in both proximities, robot will turn around. Otherwise, it will not be activated.
- SuperBot: Same to Robot. If SensorReading does not return -1 and the SuperBot do not have DistressCall signal, then SuperBot's proximity would be activated. After it is activated, the output is the distance between them. And they will avoid to hit each other. If another entity is in the left sensor, the SuperBot will turn right. Else if another entity is in its right sensor, robot will turn left. Else if the entity in both proximities, SuperBot will turn around. Otherwise, it will not be activated.
- Player: If player is in Robot/SuperBot range and SensorReading is not -1, proximity would not be activated. The Robot/SuperBot's behavior will not be affected.
- HomeBase: Same to Robot. If SensorReading does not return -1 and the Robot do not have DistressCall signal, then Robot's proximity would be activated. After it is activated, the output is the distance between them. And they will avoid to hit each

other. If another entity is in the left sensor, the robot will turn right. Else if another entity is in its right sensor, robot will turn left. Else if the entity in both proximities, robot will turn around. Otherwise, it will not be activated.

- RechargeStation: Same to Robot. If SensorReading does not return -1 and the Robot do not have DistressCall signal, then Robot's proximity would be activated. After it is activated, the output is the distance between them. And they will avoid to hit each other. If another entity is in the left sensor, the robot will turn right. Else if another entity is in its right sensor, robot will turn left. Else if the entity in both proximities, robot will turn around. Otherwise, it will not be activated.
-

Q2:

In MotionHandler, if the sensor is activated, motion handler will change its heading angle to a specific number.

Q3:

The sensor needs to have the position of its own robot and the position of the distressed robot in order to calculate the distance between its own robot and sensed object. The position of the distressed robot will be passed by EventProximity. The position of its own robot can be obtained from the link of robot the sensor saved.

Distress Sensor

Q1:

- Robot: If the output of distress sensor is 1, it shows that the robot can be collided. Other Robot or SuperBot would not be avoid to hit it.

- SuperBot: If the output of distress sensor is 1, it shows that the SuperBot can be collided. Other Robot or SuperBot would not be avoid to hit it.
- Player: Nothing happen.
- HomeBase: Nothing happen.
- RechargeStation: Nothing happen.
- Obstacles: Nothing happen.

Q2:

Distress Sensor can affect entity behavior in Arena. The distress signal can be sensed when it is within a defined range, but the direction of the signal cannot be determined. Sensor output is 1 for a sensed call and 0 for none. In MotionHandler, if the output is 1 and the distress sensor is activated, the heading angle would not be changed when proximity sensor want to change the Robot or SuperBot's heading_angle.

Q3:

The sensor needs to have the position of its own robot and the position of the distressed robot in order to calculate the correct heading angle of motion handler. The position of the distressed robot will be passed by EventDistressCall. The position of its own robot can be obtained from the link of robot the sensor saved.

As far as the motion handler, I create a base motion handler. MotionHandlerHomeBase. MotionHandlerPlayer, MotionHandlerRobot, MotionHandlerSuperBot are all derived from MotionHandler. All the MotionHandlers can reset and update themselves.

Strategy pattern:

Additionally, the converting between Robot and SuperBot is a interesting point. I would like to use the strategy pattern like the duck exercise we did before. In duck exercise, there are two class which named QuackBehavior and FlyBehavior. And it contains fly() and quack(). We can only see the QuackBehavior because quack() is a pure virtual function. After class QuackBehavior, there are several class which are the subclass of QuackBehavior. They will perform differently because QuackBehavior class is abstract. As the Requirement said, we are not allowed to use a simply flag to distinct the Robot and SuperBot. Following this example and the requirement, we can convert Robot and SuperBot by change their MotionHandler. After all, since all different and complex attributes in Robot class, creating a new SuperBot behavior to apply strategy is really complicated. Thus, by simply applying strategy on MotionHandler, we can easily transfer Robot to SuperBot. Firstly I create a MotionHandler * motion_handler; in robot.h file. Thus in robot.cc, I can let motion_handler = new MotionHandlerRobot() or MotionHandlerSuperBot() when we need to. In constructor I construct all robot to Robot. When a Robot hit HomeBase, a event I created before named EventTrans happened and robot will accept it in arena. It can pass a bool value to robot.cc. If the bool value is true, then motion_handler = new MotionHandlerSuperBot(). It is a easy and convenient way.

Observer pattern:

Arena is the subject and mobile entities are observers. When there is new information in Arena, Arena will notify all the observers by sending information to all. Observers will accept information by Accept(EventBaseClass* e). They will update themselves if they

are capable to this information.

3.3 Design Rationale

The reason of using observer pattern is that we can make sure all the entities can accept the updated information which made in Arena. All observers can be notified a basic changed information and process the information by themselves. If we change or update the entities attributes in Arena directly, then it violets the observer pattern. Strategy pattern an help Robot convert to SuperBot easily, meanwhile, we can distinct the Robot and SuperBot by not only a simply flag but the motion_handler. This strategy pattern can help us add more complex difference between Robot and SuperBot in the future. Because we can add more different behaviors without changing the existing codes. Other kinds of objects can also reuse these behaviors since they are no longer hidden from Robot and SuperBot.

4. Data Design

In iteration1, we use hard code to write a main method. Each time we want to make some changes of numbers of entities, radius or positions, we had to each the code and recompile the program. But now, we can write those data in a file and use I/O method to read the file and create the view. Just changed the data in that file, we can change numbers of entities, radius or positions.

Here is the Input file:

```
robotnum 6
Obstacle 200 200 30.0 255 255 255 255
Obstacle 100 200 30.0 255 255 255 255
Obstacle 300 200 30.0 255 255 255 255
Obstacle 400 200 30.0 255 255 255 255
Obstacle 500 200 30.0 255 255 255 255
Player 100 400 25.0 100 10 1 0 0 255 255
HomeBase 350 400 20.0 10 1 255 0 0 255
RechargeStation 500 300 20.0 123 128 128 255
Robot 250 250 15.0 10 1 0 255 0 255
Robot 150 300 15.0 10 1 0 255 0 255
Robot 350 600 15.0 10 1 0 255 0 255
Robot 300 550 15.0 10 1 0 255 0 255
Robot 600 550 15.0 10 1 0 255 0 255
Robot 700 550 15.0 10 1 0 255 0 255
```

Those numbers have a input pattern which is :

Obstacle: x, y, rad, r, g, b, a

Player: x, y, rad, battery_max_charge, angle_delta, collision_delta, r, g, b, a

HomeBase: x, y, rad, angle_delta, collision_delta, r, g, b, a

RechargeStation: x, y, rad, r, g, b, a

Robot: x, y, rad, angle_delta, collision_delta, r, g, b, a

robotnum: the number of robots.

5. Interface Design:

I have 5 obstacles(White), 6 robots(Green), 1 homebase(Red), 1 player(Blue), 1 rechargestation(Grey) and 2 UI buttons. After Robot converts to SuperBot, it would become Black.

5.1 ScreenShot