Robot Simulation

Software Design Document

Name: Gregory Star

Lab Section: Friday at 2:30pm

Date: 11/29/2017

TABLE OF CONTENTS

1. INTRODUCTION

1.1 Purpose

This software design document describes Iteration2 of the Robot Simulation project. It is intended for both the author (myself) and also anybody who needs to inspect my code for grading purposes or otherwise.

1.2 Scope

This software is strictly for educational purposes, and is being constructed as part of an assigned project to help us learn about large code bases. It contains a fairly involved inheritance structure with hundreds (if not thousands) of lines of code.

1.3 Overview

This document explains the design of the project, and discusses certain problems and how they are solved. It also provides a description of the way in which components work together, and attempts to flesh out the structure of those components (inheritance).

1.4 Disclaimer

While this document attempts to give a good description of the program, it should be supplemented with the UML diagram and generated Doxygen files. The program is large and covering all aspects of it in a concise manner is impossible.

Also, this document describes the design I was following, but it does not necessarily describe the final product. I ran into many bugs, and time pressure forced some strange decisions. The BugReport contains more information about this.

2. SYSTEM OVERVIEW

2.1 Description of Operation

In Iteration2, Robot Simulation is designed as a game. There are circular entities within a rectangular arena. Some entities stay still, acting like obstacles, while other entities move around. Furthermore, some entities are autonomous whereas others are controlled by the player. Mobile entities include SuperBots, Robots, HomeBase and Player.

The game is won if Player freezes all Robots, and lost if all Robots become SuperBots or if the Player runs out of battery. How Robots freeze and become SuperBots, as well as how the player runs out of battery, will be explained below.

2.2 Arena and GraphicsArenaViewer

Arena is class which serves as a model for the GraphicsArenaViewer. All Entities reside in Arena, and their interactions are facilitated by Arena. GraphicsArenaViewer gets updates from Arena and draws the resulting information. To see the result of this drawing process, see **Section 5**.

2.3 The Entities in Arena

Robot: an autonomous entity that moves around the screen. Robots may become SuperBots.

SuperBot: an autonomous entity that bounces off obstacles much like a Robot, but with some slightly different properties.

Player: an entity whose movement is controlled by the user.

HomeBase: an autonomous entity which behaves much like a Robot, only its interaction with Player is different than that of Robot.

RechargeStation: an immobile entity which recharges the battery of the Robot.

Obstacle: an entity which is immobile and only serves to impede the movement of other entities.

Note that here all entities are treated like they are built from different classes, but this is not necessarily the case. This is elaborated on further in **Section 3**.

2.4 Sensors

Sensors are how Entities gain information about the Arena they live in, so that they know how to react. Since immobile entities can not react, they do not possess Sensors; instead, Sensors are attached to MobileEntities. The following is a list of the different kinds of Sensors which are instantiable.

SensorProximity: A cone-shaped sensor which outputs the distance from its owner to the object it detects.

SensorDistress: A Sensor which can sense distress calls from frozen Robots within a certain range, but cannot determine where they came from.

SensorTouch: A Sensor which can sense contact between itself and another entity. It can also determine the type of the entity.

SensorEntityType: A Sensor which can detect the type of an Entity from a distance.

Information is passed to Sensor via Events in Arena. The relevant events are EventDistressCall, EventProximity, EventTypeEmit, and EventCollision. Events are what hold the information necessary for a Sensor to process them.

As an example, Arena may create an EventCollision and populate it with information about the colliding Entity. Then that information is passed to Robot, which passes it on to its Sensor. The Sensor processes the Event and then Robot asks the Sensor for Output. More about this process is written in **Section 3**.

2.5 Some More Detail about Mobile Entities

With some knowledge of Sensors, it is now possible to give a more detailed description of the MobileEntities. All MobileEntities either contain a SensorTouch or can handle EventCollisions directly (Mobile Entities should never directly process EventCollisions, but refactoring this is impractical at the moment). However, in addition to a SensorTouch, Robot also contains a SensorDistress, SensorProximity, and SensorEntityType.

2.6 The Special Property of Player

Unlike other Entities, Player contains a Battery that depletes with movement. If the battery depletes to 0, the game is lost.

2.7 Collision Behavior Between Entities

Player/Robot: Player bounces off of Robot and Robot is frozen.
Player/SuperBot: The Player is frozen for a few seconds and the SuperBot bounces off.
Player/RechargeStation: Player's battery is refilled and then Player bounces off.
Player/Obstacle: Player bounces off, but its speed is reduced.
Player/HomeBase: Both Player and HomeBase bounce off of each other.
Robot/Robot: If one of the Robots is frozen, then that Robot is unfrozen. An unfrozen Robot in a Robot on Robot collision bounces.
Robot/SuperBot: Robot and SuperBot bounce off of each other.
Robot/Obstacle: If Robot collides with an Obstacle, the entity bounces off the Obstacle and the Obstacle remains still.
Robot/HomeBase: The Robot turns into a SuperBot. Both entities bounce off of each other.
Robot/RechargeStation: The Robot bounces off the RechargeStation.
HomeBase/RechargeStation: The HomeBase bounces off the RechargeStation.
MobileEntity/Wall: All MobileEntities bounce off walls when they collide with them. However, Player also loses some speed and battery charge.

3. SYSTEM ARCHITECTURE

3.1 High-Level Design

This program can be surmised by the interplay between Events, Entities, Sensors, and Arena. Arena is, as its name implies, is the container of all Entities. It's responsible for creating all Events and passing them down to their proper Entity. It was originally suggested that Arena should pass information to Sensors via an Observer/Subject relationship, but this is not what I decided on. I noticed that Arena already had a copy of all entities that would be displayed on screen, so it would be possible to pass Events directly to Entities. This makes sense because really, it is an Entity who is responsible for its Sensors, not Arena. With this in mind, the following chain of logic was constructed to react to Events created in Arena.

1. Arena creates an Event
2. Arena passes the Event to an Entity
3. Entity passes the Event to one of its Sensors
4. Sensor processes the Event and changes state
5. At some point Arena asks the Entity to update itself
6. Entity asks Sensor for its state
7. Entity changes state based on the Sensor's state

GraphicsArenaViewer is responsible for actually drawing the Arena and its entities. It does this by getting periodic updates of the state of Arena. Its implementation is kept separate from Arena, so that the visual and logic ends of the system interact only in controlled ways. So following the aforementioned chain of logic, there is technically an eight step where GraphicsArenaViewer asks Arena for its state.

3.2 The Inheritance Structure

Events: All Events inherit from EventBaseClass. This includes EventRecharge, EventKeypress, EventCollision, EventDistressCall, EventEntityType and EventProximity. EventBaseClass is a fairly barren class; the only thing all Events have in common is that they have an EmitMessage function (this is mainly used for debugging).

Sensors: All Sensors inherit from Sensor. Sensor provides a getter and setter for an "activated" attribute, as well as a Reset method. Furthermore, Sensor provides an Accept method for every type of Event that a Sensor may need, as well as a default implementation (essentially a "do nothing" implementation so children don't actually have to implement these methods). While I believe it'd be better to have one Accept method that takes an EventBaseClass, this would require casting in Sensor subclasses. My solution provides a convenient -- if not very clean -- answer.

ArenaEntities: ArenaEnitity is the base class of all Entities in Arena. All ArenaEntities have a position and color (and a few other properties).

From there, ArenaEntities are broken up into ArenaImmobileEntity and ArenaMobileEntity. As their names imply, this separates Entities into those that move and those that do not. Entities that move require some more properties, like a direction and a Sensor touch.

ArenaImmobileEntities are broken up into RechargeStation and Obstacle.

ArenaMobileEntities break up into Robot, HomeBase, and Player. SuperBot does **not** inherit from Player because its behavior is not a specialization of Player. Instead, Robot has a boolean variable called isSuperBot which keeps track of this. While I believe there are some serious problems with the Strategy design pattern, I still intended to use it for the implementation of Robot's behavior; Strategy would make changing between

SuperBot behavior and Robot behavior fairly straight-forward. However, I was unable to do this due to time constraints, so instead a flag was used.

MotionHandlers: MotionHandler is an Abstract class which provides some general methods related to movement and speed. The children of motion_handler are MotionHandlerPlayer and MotionHandlerRobot, which as the names imply, handle motion for Player and Robot respectively. There is some slight confusion because Player also has a PlayerMotionBehavior; I did not like this but did not want to touch the old code. As a compromise, I simply made the new code the way I wanted, so now Robot acts as its own MotionBehavior. Ideally, these decisions would be consistent with each other.

4. DATA THROUGH A FILE

Data is read in through a text file and then Main constructs an Arena with all of the specified Entities. ArenaEntities are stored in Arena as a list. Color is represented as a struct with four integer values between 0-255 (red, green, blue, alpha). Each Entity has a Color attribute.

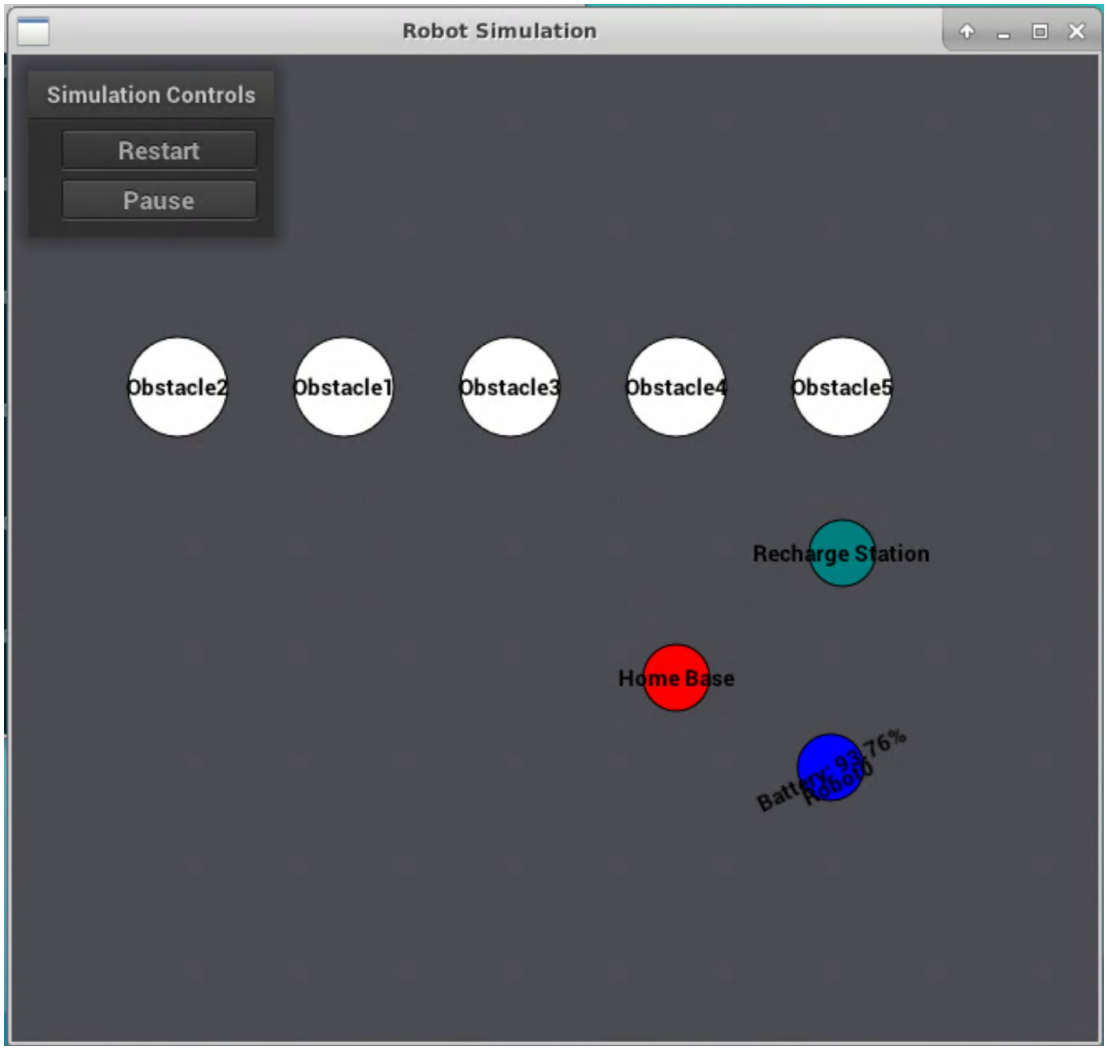
Information for the constructors of Entities is passed in via structs. There is a struct for most major classes because otherwise the number of arguments to the constructor would be unmanageable. That being said, relatively simple classes like Sensor do not have such structs.

5. HUMAN INTERFACE DESIGN

5.1 Overview of User Interface

The user interface consists of a window with a grey background. There is a translucent panel on the top left of the window that contains two buttons. The buttons are labeled “Pause” and “Restart”, which pause and restart the program respectively (when clicked). There are circles of various sizes and colors, which all have text labels. These circles represent the various entities. Some of them move, and others don’t. The user can move the Player circle with the arrow keys.

5.2 Screen Images



A screenshot of the program as it was at the end of Iteration1. Entities are labeled with text and have different colors to help distinguish them. A control panel is in the top-left corner of the screen. In the final version, there will be many different kinds of entities on screen interacting in more complicated ways.

The program looks somewhat different in Iteration2, but I am unable to take a screenshot at this time.