Robot Simulator Game

CSCI 3081W – Program Design and Development

Software Design Document

Name: Saksham Goel
Lab Section: 012
Lab TA: Sam
Date: 11/28/2017

INTRODUCTION

Purpose

The purpose of this Design Document is to help understand the iteration 2 code for the CSCI 3081W Robot Simulator project. This design document will explain what iteration 2 requires, and explain the design that was implemented for various additional features that were added. The intended audience is anyone who has some knowledge about the code for iteration 1, because iteration 2 required adding new functionality and features on top of the iteration 1 code base.

Scope

Robot Simulator Game Project Iteration 2, consist of many different types of entities interacting in the arena (play area) until some win or lose condition is met for the user. The entities have their special characteristics, and these determine how it interacts with other entities. The objective of the game is to freeze all robots (Win!), before you run out of battery or all robots become superbots (Lost!).

Overview

As the game contains lots of different things (classes and their object instances) interacting with each other it is very important to understand what different type of classes are, and what they do. One important part of the whole game is the arena which controls everything and contains all the entities. Arena is responsible to update all the entities within it and check what is happening. Because the arena has all the information about every entity, it acts as a central point for every class. As we move forward in the design document the role of the arena will become much clearer. In the next section of the design document you can find the information about the different classes present as part of the project and a short summary about them along with the design of the project.

SYSTEM OVERVIEW

System Structure

Robot Simulation Game's system structure consist of many different classes, they are listed below with small summary stating what they are.

All the different type of entities in the arena are:

Entities

- ArenaEntity – An abstract class that provides a basic interface for all the different types of entities that can be present in the arena.
- ArenaImmobileEntity – An abstract class that provides a basic interface for all the immobile entities that can be present in the arena. Obstacle and RechargeStation classes inherit from this class.
- ArenaMobileEntity – An abstract class that provides a basic interface for all the mobile entities that can be present in the arena. Player, Robot, HomeBase, SuperBot classes inherit from this class.
- Player – A mobile entity which is controlled by the user through arrow key press handlers.
- Robot – A mobile entity which is autonomous, and tried to avoid all the other entities in the arena.
- SuperBot – A mobile entity which is autonomous, and tried to avoid all the other entities in the arena except the player.
- HomeBase – A mobile entity which is autonomous and has a random motion.
- RechargeStation – A immobile entity which acts as a recharge point for the battery of the player.
- Obstacle – A immobile entity that hinders the player.

The mobile entities of the arena contain sensors which transmit the information to the entity about its surroundings. These sensors play a vital role in determining how the entity should move itself in the next update cycle.

Sensor

- Sensor – An abstract class that provides a basic interface for all the different type of sensor classes.
- SensorTouch – A sensor that is present on the perimeter of the entity. This sensor is responsible to activate when some other entity collides with the entity containing the touch sensor.
- SensorProximity – A sensor which acts like a flashlight, and informs the entity about entities present in front of it, which are within its defined range and field of view.
- SensorDistress – A sensor which acts like a circular zone around the entity and checks whether there is some entity which is distressed near the entity containing the distress sensor.

- SensorEntityType – A sensor which acts like a distress sensor but instead of checking whether the entity is distressed or not, it informs the entity about the entity type of the nearby entity within its defined range.

These two things (Entities and Sensors) are the important part of the project beside arena because the entities lay out the functionality while the sensors help to implement these functionalities.

The sensors need information about different entities. This information is possessed by the Arena, hence the way which Arena transports this information to sensors is through different events. There are different types of events as listed below. The main task of all these events is to provide information about all entities present in the arena to sensors of different mobile entities present in the arena.

<u>Events</u>

- EventBaseClass – An abstract class that provides a basic interface for all the different type of event classes.
- EventCollission – Responsible to hold information about two entities that can collide.
- EventRecharge – Responsible to hold information about the player when it hits the recharge station, because the battery of the player needs to be recharged.
- EventCommand – Responsible to convert the user key press to commands that the player can understand.
- EventKeyPress – Responsible to handle the arrow key press from users and pass it to the player so that it can update itself.
- EventProximity – Responsible to hold information about two entities that may be in proximity.
- EventDistressCall – Responsible to hold information about two entities that may have a distress call between them.
- EventTypeEmit - Responsible to hold information about two entities that are nearby each other.

The entities get all the information about other entities through the interaction of sensors and the events. This information is then used by the motion handlers of the entity to update the motion state of the entity. Motion Handlers are another set of classes that help determine the motion of the entities.

<u>MotionHandlers</u>

- MotionHandler – An abstract class that provides a basic interface for all the different types of motion handler classes.
- MotionHandlerPlayer – A class that defines the behavior of motion for the player. Contains information about the heading angle, speed, max_speed of the player. Is also responsible to translate the user key press handler command information into correct state change for the entity.
- MotionHandlerHomeBase - A class that defines the behavior of motion for the homebase. Contains information about the heading angle, speed, max_speed of the homebase.
- MotionHandlerRobot - A class that defines the behavior of motion for the robot. Contains information about the heading angle, speed, max_speed of the robot. Also is

responsible to update itself accordingly by using the information from the different type of sensors of Robot.
- MotionHandlerSuperbot - A class that defines the behavior of motion for the superbot. Contains information about the heading angle, speed, max_speed of the superbot. Also is responsible to update itself accordingly by using the information from the different type of sensors of Superbot.

All of these classes act as the pillars for the whole game and make it work as they play the most important role in the whole simulation. However, there are many other small classes that abstract other work through its member variables and functions as follows :-

<u>Miscellaneous</u>

- MotionBehavior – This class is responsible to get data from the entity and compute its next position using its current position, heading angle, current speed.
- GraphicsArenaViewer – The main class that is responsible to draw all the graphics for the game. This includes drawing all the entities of the arena, sensors for robot and superbot, battery for the robot, control panel, win and lose status.
- RobotBattery – Class that implements the battery behavior for the player. Is responsible to update the charge of the battery based on the movement of the player and recharge event.
- Color – A structure that helps to store the colors for different types of entities of the arena.
- Position – A structure that helps to store the coordinates of the entity in the arena plane.
- EntityType – An enum that defines the different entity types that an entity can possibly have.
- RobotParams – A structure that helps to store the initial parameters of the robot and superbot entity.
- PlayerParams – A structure that helps to store the initial parameters of the player entity.
- HomeBaseParams – A structure that helps to store the initial parameters of the homebase entity.
- ArenaEntityParams – A structure that helps to store the initial parameters of the generic arena entities like Obstacles and RechargeStation.
- ArenaMobileEntityParams – A structure that helps to store the initial parameters of the generic arena mobile entities.
- ArenaParams – A structure that helps to store the initial parameters of different entities present in the arena as one cohesive group comprising of different type params structures for entities.
- ReadParams – A class that is responsible to collect information from a file at run time and use those parameters to initialize the arena and its entities.

We have discussed about different classes that together make the iteration2. It is also important to discuss about the different interactions of entities in the arena to understand why these classes interact they way as given in the section below.

Interactions

- Player with:
    - Robot – Robot stops/freezes.
    - SuperBot – Player Stops.
    - HomeBase – Simple collision.
    - RechargeStation – Battery for the player gets recharged and a simple collision.
    - Obstacle – Reduce the speed and decrease the battery.
- SuperBot with:
    - Any entity except player – a simple collision event.
    - Player – Player Stops.
    - If any entity in proximity and the entity is not in distress – Avoid it.
- Robot with:
    - Any immobile entity – a simple collision event.
    - Player – Robot stops.
    - SuperBot – If frozen, unfreeze robot.
    - Robot – If frozen, unfreeze robot.
    - HomeBase – Convert to SuperBot
    - If any entity in proximity except for home base and the entity is not in distress – Avoid it.
- HomeBase – simple collision event.

This is all the details about the structure of the project. The next section discusses about the interaction of different classes to make the game work and also the design decision that was taken to complete iteration 2.

Architectural Design

Design is an integral part of making iteration 2 successful. Many different design decisions were considered while implementing the functionality and working on the code. This section will discuss the how the whole code is working together and how the classes are interacting with each other. Also, this section will provide information about the design decisions that were considered which design decision was implemented and how.

To discuss how everything is working together, A layout of how the control switches when one time frame updates is been tracked.

1. When we start the game the main function in the main.cc file is called which calls the UpdateSimulation() function which is responsible for updating the entities in the arena by calling the AdvanceTime() method of Arena and DrawUsingNanoVg() which is responsible to draw all the entities.
2. The control has shifted to Arena now. Next step is the AdvanceTime() function of the arena. This function checks whether the current game is paused or not. If not then the function calls UpdateEntitiesTimestep(). In the UpdateEntitiesTimestep() function, because the arena contains all the entities, the arena iterates over all the entities and updates them.
    a. First the arena checks whether the arena has met a win or lose condition which means
        i. Win – All the robots have been frozen by the player

        ii. Lose – All the robots have become SuperBots or The player has run out of battery.

    b. Then, because of added functionality the arena only collects the information about all the entities and then pass it to the sensors. This is achieved by iterating through all the mobile entities and for every mobile entity iterating over all other entities in the arena, making different types of events as described in the earlier section and passing it to Accept() function for all different types of sensor so that they can reach an updated state. Once all different types of sensors have updated themselves to some new state the arena calls the TimestepUpdate() function of all the mobile entities, and then resets the sensors.
        i. The above part of the code was implemented using the observer pattern design pattern. Here the subject is Arena because it has all the data about all the entities, while the sensors are the observers. Another design in which entities act as an observer were considered, however it was dropped because entity should not contain information about other entities. The design decision with sensors as observer was preferred because sensors are allowed to interact with other entities and use that information to update themselves, and also because an entity contains sensor, it makes sense to update them and then get information from them to determine the motion of the entity. The composition relationship between entities and sensors prompted the use of observer pattern with sensors as observers.
        ii. Arena is able to get access to all the different type of sensors for the entities through getters which return the pointer to these different types of sensors of the entity. This pointer to the sensors helps to update the right sensor and also helps to save data so that it can used afterwards when updating the entity.
        iii. As discussed earlier, the arena passes the information about different entities from itself to sensors using the entities and passing them to the corresponding sensor type.
        iv. Resetting the sensors is important because we don't need to have their previous set state interfere with the new updated state because sensors rely on their previous information when updating themselves. There is this dependency because they don't want to overwrite data when they don't sense anything.
    c. After this it checks for the collisions of different entities with entities and wall and then explicitly change their configuration accordingly.

3. Now the control has shifted to each individual mobile entity to update itself. Next function being called is TimestepUpdate() in the mobile entities. This function varies depending on the entity type, however one thing that is same is updating the position of the entity using the current speed and heading angle. This part is completed through the UpdatePosition() function of the robot_motion_behavior class. This function just uses math to find the next position and sets the position of the entity to the new position. Because we are calculating the new position based on the current angle and speed of the entity, it is essential that the entity first changes its state using the data from sensors. This is accomplished differently as follows:
    a. Player – check if the player has collided with superbots to determine whether it freezes or not. Also check whether the player should remain frozen considering the time that

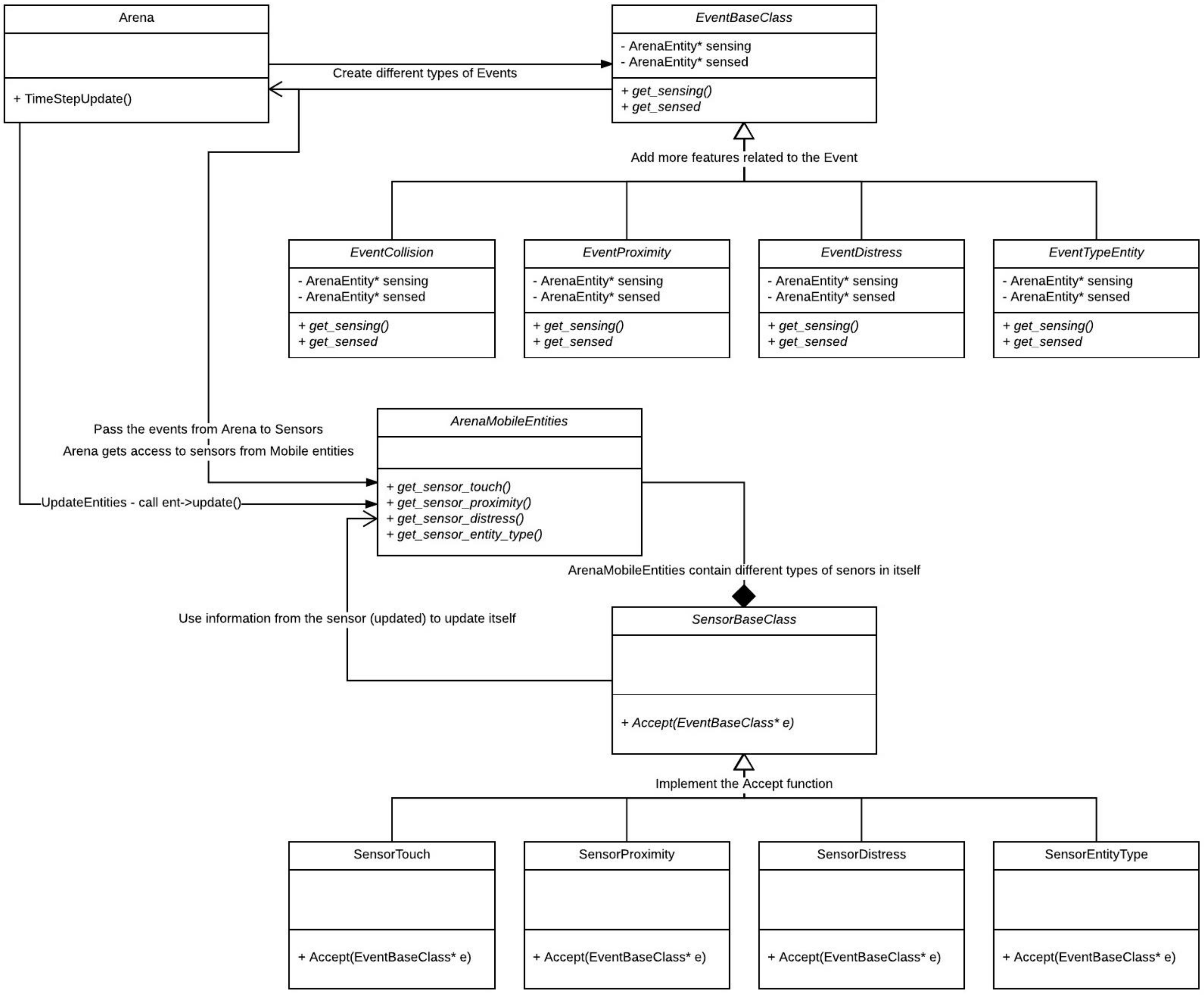
has passed since it froze. Then use the data from the SensorTouch and pass it to the motion handler (of type MotionHandlerPlayer) to update the heading angle. Once the position of the player has been updated, deplete the battery for the player accordingly.

   b. HomeBase – Use a random number generator to randomly change the direction of the homebase and then update the position.
   c. Robot – Pass the information from SensorTouch, SensorProximity, SensorDistress, SensorEntityType to the motion handler (of type MotionHandlerRobot) through the function UpdateAll() and let the motion handler update the heading angle and speed. The function basically checks for different types of interactions that a robot can have and updates the member variables according to those cases as mention before. Important design decision considering the case where the robot gets converted into Superbot. I used the strategy pattern design to change the behavior of robot. The strategy pattern uses polymorphism to its advantage. I accomplish this by changing the motion handler of robot from type of MotionHandlerRobot to type of MotionHandlerSuperbot because both of these inherit from MotionHandler base class.
   d. SuperBot - Pass the information from SensorTouch, SensorProximity, SensorDistress, SensorEntityType to the motion handler (of type MotionHandlerSuperbot) through the function UpdateAll() and let the motion handler update the heading angle and speed. The function basically checks for different types of interactions that a robot can have and updates the member variables according to those cases as mention before.

4. Now the control is shifted back to the main loop as most of the update has been done and it is time to repeat the whole process again.

One important part of the design for the whole project is that I used polymorphism heavily and it proved to be very advantageous because it helps me access overloaded function definition during the run time which makes it easy to implement different functionality.

A sample UML diagram which just outlines a basic observer pattern design that is implemented between arena and the sensors is as follows:


Arena
+ TimeStepUpdate()
EventBaseClass
- ArenaEntity* sensing
- ArenaEntity* sensed
+ get_sensing()
+ get_sensed
Create different types of Events
Add more features related to the Event
EventCollision
EventProximity
EventDistress
EventTypeEntity
- ArenaEntity* sensing
- ArenaEntity* sensed
+ get_sensing()
+ get_sensed
Pass the events from Arena to Sensors
Arena gets access to sensors from Mobile entities
UpdateEntities - call ent->update()
ArenaMobileEntities
+ get_sensor_touch()
+ get_sensor_proximity()
+ get_sensor_distress()
+ get_sensor_entity_type()
ArenaMobileEntities contain different types of senors in itself
Use information from the sensor (updated) to update itself
SensorBaseClass
+ Accept(EventBaseClass* e)
Implement the Accept function
SensorTouch
SensorProximity
SensorDistress
SensorEntityType
+ Accept(EventBaseClass* e)