CSCI3081 iteration 2
Software Design Document

Name : William Batu
Lab Section: 12:20 PM

Date: 11/29/2017

1. INTRODUCTION

1.1 Purpose

This software design document describes the architecture and system design of CSCI3081 project iteration2.

1.2 Scope

The goal of this software project is to develop 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 might be that 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.

1.3 Overview

In this iteration, the robot simulator is still like a video game, but moving towards autonomous, intelligent robot behavior. Users control a player with the arrow keys. 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.
If the Player freezes all regular robots, Player wins. If all robots turn in to SuperBots, Player loses.

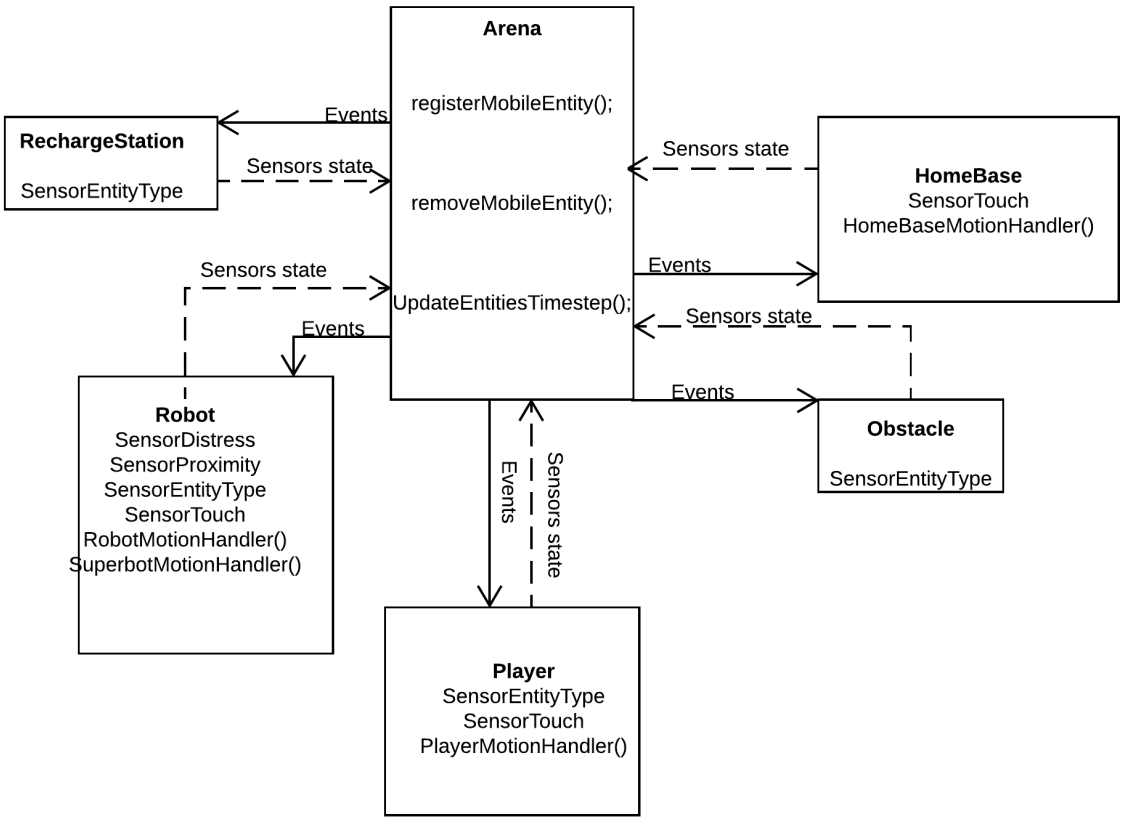
2. System Overview

Whole system using observer pattern to organize information flow and control flow. In this system arena is the subject because arena can access all the information of this system. Observer is the mobile entities because sensors of arena mobile entities are the one really do the update state computation.

Firstly Arena update the state of arena entity using current state of arena entity. Then check all the event happening in the area and create the specific event object for each event. After creating event objects, dispatch event objects to related arena mobile entity. Then mobile entity process event information using own handler and update the state.

3. System Architecture

3.1 Architectural Design



System Architecture

(1) All entity registered in arena.

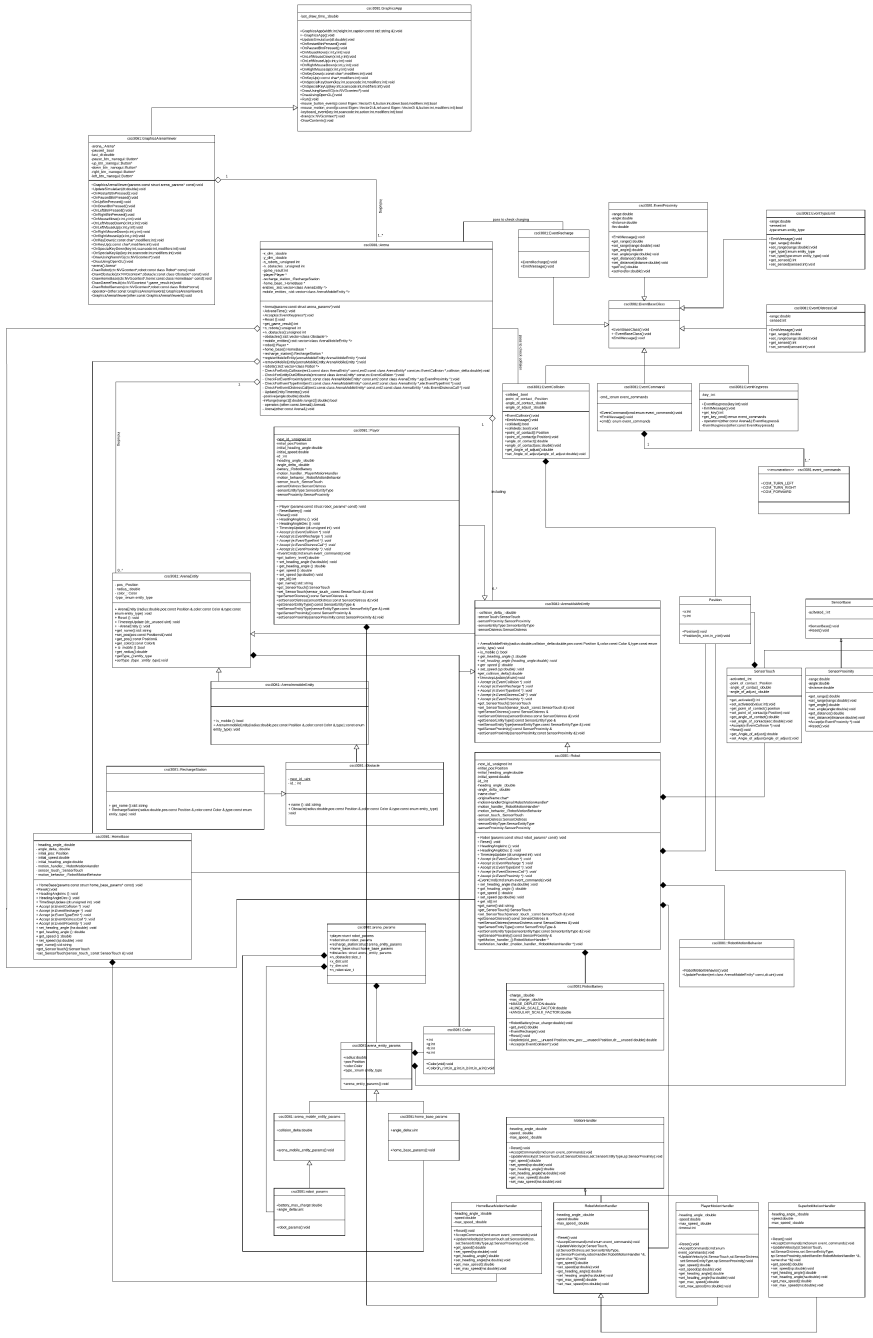
(2) After arena check all kinds of events based on sensors state of entities and state of entities.

(3) then dispatch event using timeStepUpdate() to all registered entities then sensors accept the event.

(4) Sensors update internal states based on accepted events.

(5) Arena entity based on state of sensors to handle own state.

3.2 Decomposition Description



Above diagram is the UML diagram for the system which includes inheritance/aggregation/ composition relation.

3.3 Design Rationale

This system using observer pattern to organize information flow and control flow. In this system arena is the subject because arena can access all the information of this system. Observer is the mobile entities because sensors of arena mobile entities are the one really do the update state computation.

3.3.1 Why observer pattern?

Organize in this way, the whole systems information flow is very clear. Arena dispatch the event information to the arena mobile entities then sensors of mobile entities process the event then update the state.

(1) The reason choosing arena mobile entities to be the observers is easy to implement.
If we use sensors as the observers, then there is a lot of duplicate code in arena and arena entities. Because we need to register in the arena and arena entities and when dispatch the events there is large number of sensors to write.

(2) Another reason choosing arena mobile entities to be the observers is easy to scale. When adding new arena entities, we just register in arena and dispatch event to it.

3.3.2 Why super bot motion handler is inherit from robot motion handler?

(1) super bot is one kind of robot, so its handler should be inherit from robot motion handler.

(2) using inheritance is the way to use polymorphism. If we declare robot motion handler in the Robot class, then we can change it to super bot motion handler at runtime.

3.3.3 Why add type to the arena entity class?

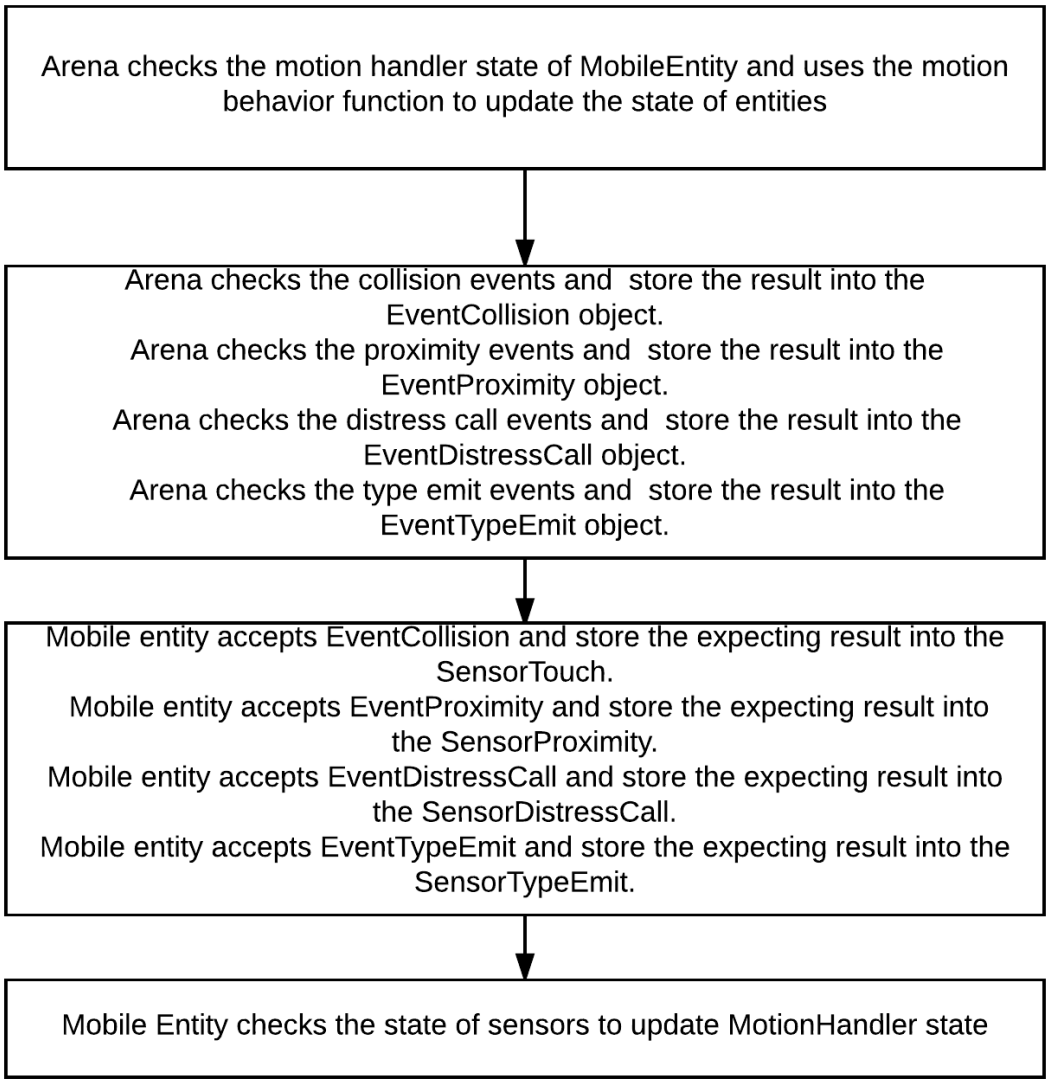
When determine the EventTypeEmit state, we need type information of arena entity.

3.3.4 Why check all the events in the Arena functions?

Because the one and only object that know all the information of the system is Arena. If we check events in other object then when we need new event the most probably need to redesign data flow. But using Arena is the most secure way to get biggest scalability because Arena hold all the information data.

4. Data Design

4.1 Data Description



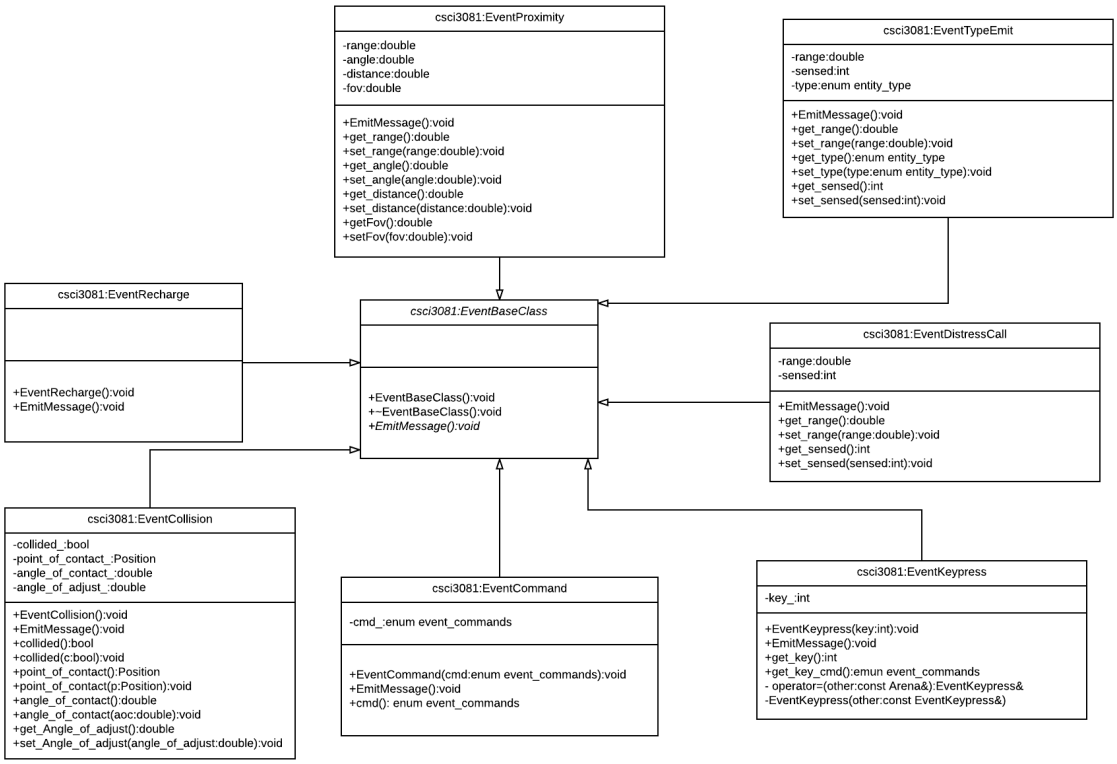
Data is flowing through from top to bottom then go back to top.

4.2 Data Dictionary

Look up the system UML provided above.

5. Component Design

5.1 Events



event UML

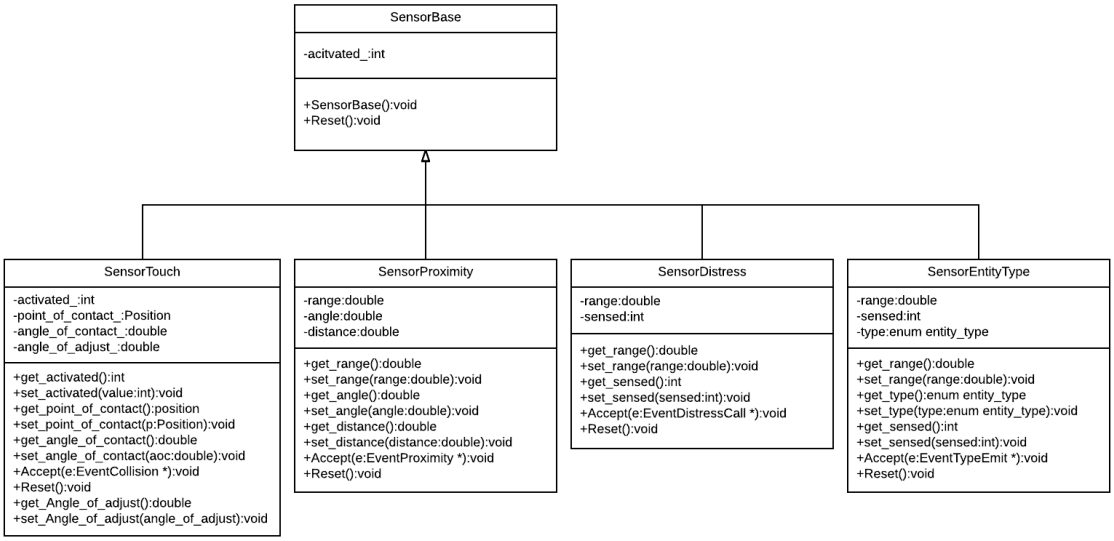
(1) local date
—EventCollision member variable collided is true means collision event happened.
—EventCollision member variable point_of_contact means collision point position.
—EventCollision member variable point_of_angle means collision angle.

—EventDistressCall member variable sensed is “1” means distress call event happened.
—EventDistressCall member variable range means distress call sensing range.

—EventProximity member variable distance is “-1” means proximity event not happened.
—EventProximity member variable range means proximity sensor sensing distance.
—EventProximity member variable angle means proximity sensor sensing cone angle.

—EventTypeEmit member variable range means type sensor sensing range.
—EventTypeEmit member variable sensed is “1” means type is sensed.
—EventTypeEmit member variable type means sensed type.
(2) getter and setter function for the member variables.

5.2 Sensors



Sensor Architecture

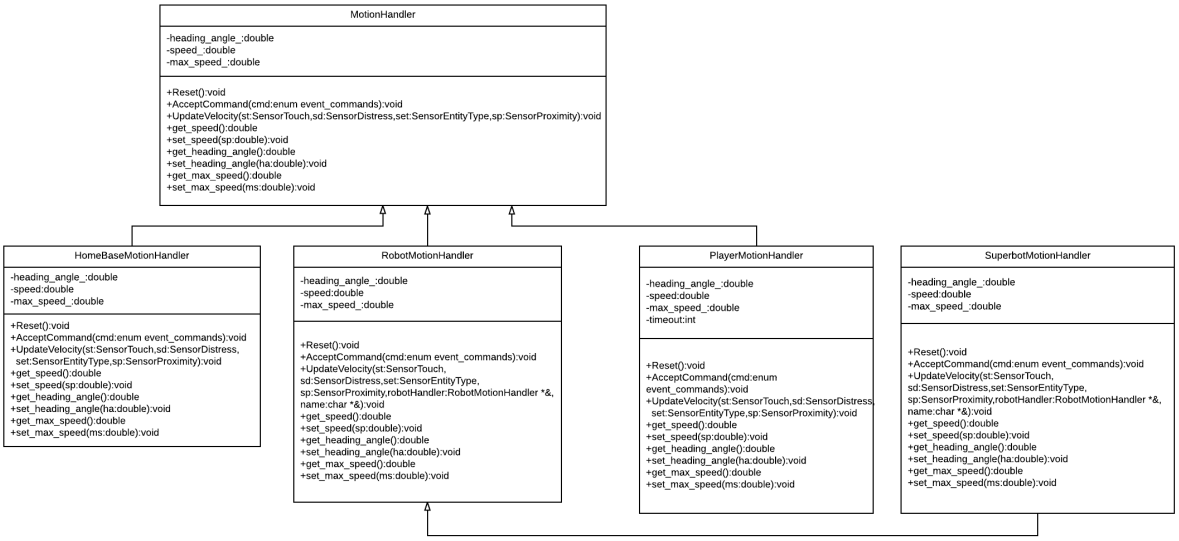
(1) Sensors update internal states based on accepted specific event.

```
Sensor::Accept(Event* e){
        if (e->get_state()==expected value){
        Sensor::set_state(e->get_value_of_expecting_member);
        }
}
```

(2)Sensors can reset all state to refresh beginning state.

```
Sensor::Reset(){
        member_variable=initial_value;
}
```

5.3 MotionHandlers



MotionHandler Architecture

(1) Every arena entity has MotionHandler to handle the motion.

(2) MotionHandler accept the sensors to update states.

```
MotionHandler::Accept(sensors){
        if(sensors.get_value()==expected_value){
                speed = num1;
                heading_angle=num2;
        }
}
```

(3) Different combination of sensors state will cause different result state of Motion handler.

```
MotionHandler::Accept(SensorProximity sp, SensorTouch st, SensorDistressCall
                        sdc, SensorTypeEmit ste){
if(sp.get_distance()!=-1) && (ste.get_type!= player) && (sdc.get_sensed==0){
heading_angle+=180; //escape from obstacle
} else {
heading_angle+=0; //remain the angle
}
}
```

5.4 Arena

GraphicAppViewer calls updateEntitiesTimestep() 60 times per second to proceeding the game.

The functions pseudo code:

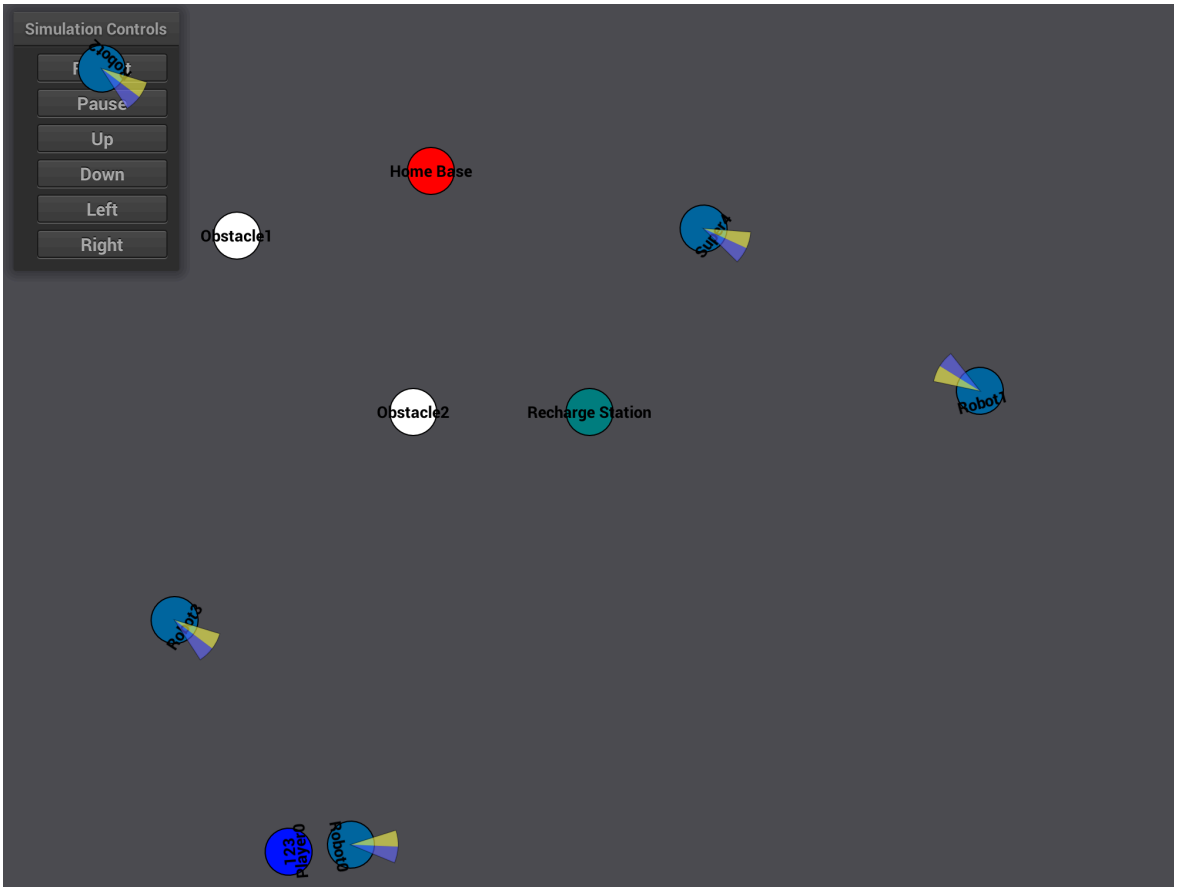
```
Arena::UpdateEntitiesTimestep(){
        for(auto ent : entities_) {    ent->TimestepUpdate(1);   }
        for(auto ent : entities_) {
                for(int i=0;i<entities_.size();i++){
                        CheckEventCollision(ent, entities_[i], &eventCollision);
                        if (eventCollision.collided()){
                                break;
                        }
                }
                ent.Accept(&eventCollision);
                for(int i=0;i<entities_.size();i++){
                        CheckEventProximity(ent, entities_[i], &eventProximity);
                        if (eventProximity.distance()!=-1){
                                break;
                        }
                }
                ent.Accept(&eventProximity);
                for(int i=0;i<entities_.size();i++){
                        CheckEventDistressCall(ent, entities_[i], &eventDistressCall);
                        if (eventDistressCall.sensed()){
                                break;
                        }
                }
                ent.Accept(&eventDistressCall);
                for(int i=0;i<entities_.size();i++){
                        CheckEventTypeEmit(ent, entities_[i], &eventTypeEmit);
                        if (eventTypeEmit.sensed()){
                                break;
                        }
                }
                ent.Accept(&eventTypeEmit);



        }
}
```

6. Human Interface Design

6.1 Overview of User Interface

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.

6.2 Screen Images



6.3 Screen Objects and Actions

- With UI buttons: Restart and Pause
- Restart button: reset all the arena entities to the new beginning state.
- Pause button: stop all the updating action
- With left and right arrow keys, change the direction of the robot
- With up and down arrow keys, change the speed of the robot (no reverse)

7. Requirements Matrix

Priority 1	SensorProximity	SensorTypeEmit	SensorDistressCall	SensorTouch	
Priority 1	MotionHandler	RobotMotionHandler	PlayerMotionHandler	HomeBaseMotionHandler	SuperbotMotionHandler
Priority 1	Player				
Priority 1	EventProximity	EventDistressCall	EventTypeEmit	EventCollision	
Priority 2	SensorProximity	SensorTypeEmit	SensorDistressCall	SensorTouch	
Priority 2	Arena				
Priority 3	Player	Robot	HomeBase	Arena	