Software Design

Robot Simulation

by Zhongyi Sun

Date: November 28

1. INTRODUCTION

1.1 Purpose

This software design documents describes the architecture and system design of Robot Simulation with a video game's perspective. The intended audience is other programmers who need to understand class interfaces, function parameters, function behavior, code organization, class organization, and code logic. The purpose for this design is to help and guide others through the logic and class interfaces. In order to show the clear structure, and readable coding stuff for this design.

1.2 Scope

In this Robot Simulation software design, we will OOP language C++ to implement it. And this software project is to develop a rudimentary robot simulator in which robot behavior is visualized within a graphics window where there are many entities e.g. obstacles, automatic robot, homebase, recharge station and so on. This simulation can be used in many practical perspectives. For example, it might be a test environment for experimenting with autonomous robot control using sensor feedback to provide those collected information to scientists or statists support them to do their research, or it might be a visualization of real robots operating in a remote location for some real industries or some places where human can't arrive at right now. Overall, it can be used in many areas and aspects. For this design, it is used more like a videogame in which users control the robots and to see what will happen if those entities move and have collisions with each other in that graphics window.

The goal for this game software is to be published and to be provided to the customers in the future.

1.3 Documentation Overview

- Introduction
  - Purpose
  - Scope
  - Documentation Overview
  - Reference Material
- System Overview
- System Architecture
- Data Design
- Component Design
- Human Interface Design

1.4 Reference Material

ProjectClassOverview By CSCI 3081W

2. SYSTEM OVERVIEW

All of robots have autonomous, intelligent behavior; they can determine their behavior by detecting the different consitions. 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 player runs out of energy before let all robots become freeze, the player lose the game.

3. SYSTEM ARCHITECTURE

3.1 Architectural Design

For the module structure of this design, the main system is composed by several subsystems: Graphic Viewer, Arena, Entities, Events, Sensor, MotionHandler; First of all, Graphic Viewer is used to draw all of the entities as their states are changed, graphic window and GUI. Then, Arena keeps those the entities and check different conditions and provides the information to Events. After the Events are generated via arena at every timeupdate, they are passed and accepted by the entities, especially Robot. Moreover, entity will be accepted by related sensor. Last, Motionhandlers will react by updating their movement for the events sensors received.

3.2 Decomposition Description

1. **Graphic Viewer**
   Associated Files:
   Graphics_arena_viewer.cc
   graphics_arena_viewer.h
   This creates the graphics window and the GUI. It is controlling the timing of the screen update, performing the drawing, watching for key and mouse events, and it is the keeper of the arena.

2. **Arena**
   Associated Files:
   arena_params.h;
   arena.cc; arena.h
   Class contains all the entities within the arena:
   1.Player * player_
   2.Robot * robot_
   3.Recharge_station * recharge_station_

4.HomeBase * home_base_
5.vector< Robots * > robots_
6.vector< Obstacles * > obstacles_
7.All entities are stored in vector< ArenaEntity * > entities_
8.All mobile entities are stored in vector< ArenaMobileEntity * > mobile_entities_
Constructor creates all these entities.

3. **Entities**

Base Classes

arena_entity.h class ArenaEntity.
arena_immobile_entity.h class ImmobileArenaEntity : ArenaEntity
arena_mobile_entity.h class MobileArenaEntity : ArenaEntity
    arena_mobile_entity.cc
    arena_mobile_entity_params.h : struct used for initializing entity

Entities generally have:
color, position, size
if mobile, heading and velocity too
is_mobile() return True or False
reset()
TimeStepUpdate()

Home Base

Associated Files:
    home_base.cc
    home_base.h
    home_base_params.h : struct for initializing
Class contains sensors and motionhandler:
    SensorTouch
    HomeBaseMotionHandler
class Homebase : public MobileArenaEntity

This is the object that the robot can become superbot when colliding with it. The intent is to have this moving, and better yet, have it randomly change directions at unpredictable times to make the game harder.

Recharge Station

Associated Files:
    Recharge_station.h
class RechargeStation : public ImmobileArenaEntity

This is a stationary entity that has the ability to recharge the player’s battery. The player needs only collide with it to get a full charge to its battery.

Obstacles

Associated Files:

- obstacle_params.h
- obstacle.cc
- Obstacle.h

class Obstacle : public ImmobileArenaEntity

Circular stationary objects that get in the way of the robot.

Robot

Associated File:

- robot_params.h
- robot.cc
- robot.h

class Robot : public MobileArenaEntity

Class contains sensors and motionhandler:

- SensorProximity
- SensorTypeEmitType
- SensorDistress
- SensorTouch
- RobotMotionHandler

Automated and intelligent, not controlled by user.

Superbot

Associated File:

- superbot.cc
- superbot.h

class Robot : public MobileArenaEntity

Class contains sensors and motionhandler:

- SensorProximity
- SensorTypeEmitType
- SensorDistress
- SensorTouch
- SuperbotMotionHandler

After Robot has collision with homebase, it change its behavior.

Player

Associated File:

- player_params.h
- player.cc
- player.h

class player : public MobileArenaEntity

Class contains sensors and motionhandler:

SensorTouch
PlayerMotionHandler
user-controlled Robot via input from keyboard.

4. **Event**
Base class:
Event_base_class.h EventBaseClass

It provides information regarding the entities involved and the location of the event, and it can be classified into many different events.

EventCollision:
Associated file:
Event_collision.h
Event_collision.cc
class EventCollision : public EventBaseClass
After Collisions happen and have been checked, arena will generate this event and pass it to the entity

EventDistressCall:
Associated file:
Event_distresscall.h
class EventDistressCall : public EventBaseClass
This event is generated when robot collides with player
When unfreeze robot has collision with robot, then
EventKeypress:
Associated file:
Event_keypress.h
Event_keypress.cc
class EventKeypress : public EventBaseClass
When user pressed the key from keyboard, this event is generated by arena and to control the movement of player

EventProximity:
Associated file:
Event_proximity.h
event_proximity.cc
class EventProximity : public EventBaseClass
If when collision intends to happen within a defined range, but not yet. In this case, this is a useful event that makes the robot change their direction

EventRecharge:

Associated file:

Event_recharge.h

Event_recharge.cc

class EventRecharge : public EventBaseClass

When player collided with recharge station, this event is generated.

EventTypeEmit:

Associated file:

Event_type_emit.h

event_type_emit.cc

class EventTypeEmit : public EventBaseClass

When collisions happen or intend to happen within a defined range, the mobile entity need to know what entity it will collided. This event is generated.

5. Sensor

Bass class

Sensor_base.h: Sensor

Those sensors are detectors is used by all mobile entities, but most of sensor is used by robot. The sensor determines if the event activates the sensor. and it can be classified as many different sensors.

SensorDistress:

Associated File:

Sensor_distress.h

sensor_distress.cc

class SensorDistress : public Sensor

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.

SensorEntityType:

Associated File:

Sensor_entity_type.h

sensor_entity_type.cc

class SensorEntityType : public Sensor

It is used to sense signal the type of the entity emitting when it is within a defined range, but the direction of the signal cannot be determined. Sensor output is the enumerated entity type.

SensorProximity:

Associated File:

Sensor_proximity.h

Sensor_proximity.cc

class SensorProximity : public Sensor

A proximity sensor should have a range and *field of view*, such that it has a limited cone in which it can sense objects. A single cone emanating from the center of the robot, split in two, signifies the two fields of view for the two sensors. The range and field of view, expressed as an angle, are attributes of the sensor. Sensor output is the distance to an obstacle. If there is no obstacle, it should output -1.

SensorTouch:

Associated File:

Sensor_touch.h

sensor_touch.cc

class SensorTouch : public Sensor

this one is inherited from sensor base, activate or signal when touched or bumped. Any time the robot collides with something (as determined by the Arena), an event is sent to the sensor to set its activation. Conversely, when collision is no longer detected, an event is sent to the sensor to deactivate it.

Motionhandler:

Base class:

Motion_handler.h : MotionHandler

For mobile entities, which This manages the setting of velocity for the player (or for any mobile arena entity). It can be classified as many various motion handler.

RobotMotionHandler:

Associated file:

robot_motion_handler.h

Robot_motion_handler.cc

To handle robots' motion

SuperbotMotionHandler:

Associated file:

superbot_motion_handler.h

superbot_motion_handler.cc

To handle superbot's motion

PlayerMotionHandler:

Associated file:

player_motion_handler.h

Player_motion_handler.cc

To handle player's motion

HomeBaseMotionHandler:

Associated file:

home_base_motion_handler.h
home_base_motion_handler.cc
To handle homebase's motion

3.3 Design Rationale

The Rationale for this design is using Observer pattern and Strategy pattern to fulfil the functionalities. Robots are good candidates for observers since they have all sensors, so they are able to collect the information from their sensor in the arena. It is also easy to consider sensors as observers since sensors deal with those events eventually, but I think the sensors are parts of robots and robots keep updating and accepting event from the arena. Therefore, robots are more reasonable to be considered as observers. As for the strategy pattern, because we know that the superbots and robots have same sensors but different behaviors, so I can keep the robot object, but changing its behavior, make its behavior look like a superbot. That is easier to think and implement.

4. DATA DESIGN

4.1 Data Description

A lot of Objects in this design: Graphic Viewer, Arena, ArenaEntity, ArenaImMobileEntity, ArenaMobileEntity, Homebase, RecharStation, Obstacles, Robot, Superbot, Player, EventCollision, EventDistressCall, EventBaseClass, EventKeyPress, EventProximity, EventRecharge, EventTypeEmit, SensorDistress, SensorEntityType, SensorProximity

Using vectors to store them, such as entities vector, mobile entities vector, robot entities vector, obstacles vector in the arena. When initializing the those objects, we store those objects via pushing back to that vector. since if we want to keep checking the different conditions, we should use vectors to loop each element to find the result.

4.2 Data Dictionary

Graphic Viwer:

Methods and members:

```
Public:
void UpdateSimulation(double)
void OnRestartBtnPressed();
void OnPauseBtnPressed();
void OnMouseMove(int, int)
void OnLeftMouseDown(int,int)
void OnLeftMouseUp(int,int)
void OnRightMouseDown(int,int)
void OnRightMouseUp(int,int)
void OnKeyDown(const char *,int)
void OnKeyUp(const char *,int)
void OnSpecialDown(int,int,int)
void OnSpecialUp(int,int,int)
void DrawUsingNanoVG(NVGcontext*)
Arena* arena(void) const
Private:
DrawRobot(NVGcontext *, const class Robot* const)
DrawObstacle(NVGcontext *, const class Obstacle* const)
DrawHomeBase(NVGcontext *, const class HomeBase* const)
```

DrawPlayer(NVGcontext *, const class Player* const)
Arena *arena_;
bool paused_;
double last_dt = 0.;
nanogui::Button *pause_btn_;
GraphicsArenaViewer& operator=(const GraphicsArenaViewer& other) = delete;
GraphicsArenaViewer(const GraphicsArenaViewer& other) = delete;

Arena:

Public:
explicit Arena(const struct arena_params * const params);
Void AdvanceTime(void)
Void Accept(EventKeypress *)
Void Reset(void)
Unsigned int n_robot(void)
Unsigned int n_obstacles(void)
std::vector<class Obstacle*> obstacles(void)
std::vector<class Robot*> robots(void)
std::vector<class ArenaMobileEntity*> mobile_entities(void)
Player* player(void)
Robot* robot(void)
HomeBase* homebase(void)
void CheckForEntityCollision(const class ArenaMobileEntity* const ent1,const class ArenaEntity* const ent2,EventCollision * ec,double collision_delta);
Void CheckForEntityOutOfBounds(const class ArenaMobileEntity* const, EventCollision *)
Void UpdateEntitiesTimestep(void)
double x_dim_;
double y_dim_;
unsigned int n_robots_;
unsigned int n_obstacles_;
Robot * robot_;
Player* player_;
RechargeStation * recharge_station_;
HomeBase * home_base_;
ArenaEntity * entity_;
ArenaMobileEntity * mobilentity_;
std::vector<class Robot*> robots_;
std::vector<class ArenaEntity*> entities_;
std::vector<class ArenaMobileEntity*> mobile_entities_;

ArenaEntity (Abstract):

Public methods:

ArenaEntity(double radius, const Position& pos,const Color& color)
Virtual void TimeStepUpdate(_unused uint)
Virtual void Reset()
Virtual std::string name() const = 0
Void set_pos(const Position&)
Const Position& get_pos()
Const Colo& get_color()
Void set_color(const Color& color)
Virtual bool is_mobile() = 0
Double get_radius() const
Private:

double radius_;
Position pos_;
Color color_;

ArenaImMobileEntity :

Public :
ArenaImmobileEntity(double radius, const Position& pos,const Color& color)

bool is_mobile(void)

ArenaMobileEntity Abstract

Public:
ArenaMobileEntity(double radius, double collision_delta,const Position& pos, const Color& color)
bool is_mobile(void)
virtual double get_heading_angle(void) const = 0;
virtual void set_heading_angle(double heading_angle) = 0;
virtual double get_speed(void) = 0;
virtual void set_speed(double sp) = 0;
double get_collision_delta(void) const { return collision_delta_; }
void TimestepUpdate(uint dt);
virtual void Accept(EventCollision * e) = 0;
virtual void Accept(EventRecharge * e) = 0;
virtual void set_name(std::string s) = 0;
Private:
double collision_delta_;

RecharStation:

Public:
RechargeStation(double radius, const Position& pos,const Color& color)
std::string name(void)

Obstacles:

Public:
Obstacle(double radius, const Position& pos,const Color& color);
std::string name(void)
Private:
static unsigned int next_id_;
int id_;

HomeBase

public:
explicit HomeBase(const struct home_base_params* const params)
std::string name(void)
double get_heading_angle(void) const
void set_heading_angle(double ha)
double get_speed(void)
void set_speed(double sp)
void TimestepUpdate(uint dt)
void Accept(EventCollision * e)

void Accept(EventRecharge * e)
void Accept(EventTypeEmit * e)
void Reset(void)

Private:

Double speed_
double heading_angle_;
double angle_delta_;
double change_angle;
HomeBaseMotionHandler motion_handler_;
RobotMotionBehavior motion_behavior_;
SensorTouch sensor_touch_;

Robot

Public:

explicit Robot(const struct robot_params* const params);
void Reset(void);
double get_heading_angle(void) const
void set_heading_angle(double ha)
double get_speed(void)
void set_speed(double sp)
void TimestepUpdate(uint dt)
void Accept(EventCollision * e)
void Accept(EventRecharge * e)
void Accept(EventTypeEmit * e)
void Accept(EventProximity * e)
void Accept(EventDistressCall * e)
void Reset(void)
int id(void)
std::string name(void)
void set_name(std::string s)

Private:

static unsigned int next_id_;
int id_;
std::string name_
Double speed_
double heading_angle_;
double angle_delta_;
RobotBattery battery_;
RobotMotionHandler motion_handler_;
RobotMotionBehavior motion_behavior_;
SensorTouch sensor_touch_;
SensorEntityType sensor_entity_type_
SensorProximity sensor_proximity_
SensorDistress sensot_distress

Superbot

Public:

explicit Robot(const struct robot_params* const params);
void Reset(void);
double get_heading_angle(void) const
void set_heading_angle(double ha)

double get_speed(void)
void set_speed(double sp)
void TimestepUpdate(uint dt)
void EventCmd(enum event_commands cmd
void Accept(EventCollision * e)
void Accept(EventRecharge * e)
void Accept(EventTypeEmit * e)
void Accept(EventProximity * e)
void Accept(EventDistressCall * e)
void Reset(void)
int id(void)
std::string name(void)
void set_name(std::string s)
Private:
static unsigned int next_id_;
int id_;
Double speed_
std::string name_
double heading_angle_;
double angle_delta_;
RobotBattery battery_;
SuperbotMotionHandler motion_handler_;
RobotMotionBehavior motion_behavior_;
SensorTouch sensor_touch_;
SensorEntityType sensor_entity_type_
SensorProximity sensor_proximity_
SensorDistress sensot_distress

Player

Public:
explicit Player(const struct player_params* const params);
void ResetBattery(void);
void Reset(void);
void HeadingAngleInc(void)
void HeadingAngleDec(void)
double battery_level(void)
double get_heading_angle(void) const
void set_heading_angle(double ha)
double get_speed(void)
void set_speed(double sp)
void TimestepUpdate(uint dt)
void EventCmd(enum event_commands cmd
void Accept(EventCollision * e)
void Accept(EventRecharge * e)
void Accept(EventTypeEmit * e)
void Reset(void)
int id(void)
std::string name(void)
Private:
Double speed_
static unsigned int next_id_;
int id_;

double heading_angle_;
double angle_delta_;
RobotBattery battery_;
PlayerMotionHandler motion_handler_;
RobotMotionBehavior motion_behavior_;
SensorTouch sensor_touch_;

EventBaseClass:
Public:
EventBaseClass(void)
virtual ~EventBaseClass(void)
virtual void EmitMessage(void) = 0;

EventCollision
Public:
EventCollision()
void EmitMessage(void)
bool get_collided()
void set_collided(bool c)
Position get_point_of_contact()
void set_point_of_contact(Position p)
double get_angle_of_contact()
void set_angle_of_contact(double aoc)
Private:
bool collided_;
Position point_of_contact_;
double angle_of_contact_;

EventDistressCall
Public:
EventDistressCall()
void EmitMessage(void)
bool get_distressed()
void set_distressed(bool c)
Private:
bool distressed_;

EventKeyPress:
Public:
explicit EventKeypress(int key)
void EmitMessage(void)
int get_key(void) const
enum event_commands get_key_cmd()
Private:
int key_;
EventKeypress& operator=(const EventKeypress& other) = delete;
EventKeypress(const EventKeypress& other) = delete;

EventProximity
Public:
EventProximity()
void EmitMessage(void)
Position get_posiiton_of_sensor(void)

```
        void set_position_of_sensor(Position a)
        Position get_position_of_sensed(void)
        void set_position_of_sensed(Position p)
        double get_heading_angle_of_sensor(void
        void set_heading_angle_of_sensor(double a)
        double get_radius_of_sensed(void)
        void set_radius_of_sensed(double a)
        Private:
        Position p1, p2;
        double radius_, heading_angle_;
EventRecharge
        public:
        EventRecharge(void)
        void EmitMessage(void)
EventTypeEmit
        public:
        EventTypeEmit()
        void EmitMessage(void)
        enum entity_type get_entity_type()
        void set_entity_type(enum entity_type c)
        private:
        enum entity_type type_;
Sensor:
        Public:
        Sensor()
        int get_activated()
        void set_activated(int value)
        virtual void Reset(void) = 0;
        virtual void Accept(EventCollision * e) = 0
        virtual void Accept(EventProximity * e) = 0;
        virtual void Accept(EventDistressCall * e) = 0;
        virtual void Accept(EventTypeEmit * e) = 0;
        Private:
         bool activated_;
SensorDistress:
        Public:
        Sensordistress()
        int get_activated()
        void set_activated(int value)
        void Reset(void);
        void Accept(EventDistressCall * e);
        Private:
        int activated_;
SensorEntityType:
        Public:
        SensorEntityType()
        void Reset(void);
        void Accept(EventTypeEmit * e)
        entity_type get_entity_type(void)
        void set_entity_type(entity_type e)
        Private:
```

entity_type type_;

SensorProximity:

Public:
SensorEntityType()
void set_range(double r)
void set_fov(double f)
double get_fov()
double get_range(void)
double get_distance()
void set_distance(double d)
double sensor_reading(Position p1, Position p2, double ha, double r)
bool in_range(double lower1, double upper1,double lower2, double upper2)
void Reset(void);
void Accept(EventProximity * e);
Private:
double range_, fov_, distance_;
Position p1, p2;
double heading_angle_, radius_;
ArenaEntity* sensed_;
double speed_;

SensorTouch

Public:
SensorTouch()
void set_activated(int value)
int get_activated()
void Reset(void);
Position get_point_of_contact()
void set_point_of_contact(Position p)
double get_angle_of_contact(void)
void set_angle_of_contact(double aoc)
void Accept(EventCollision * e);
Private:
bool activated_;
Position point_of_contact_;
double angle_of_contact_;
double distance_;
double speed_;

MotionHandler:

Public:
MotionHandler()
void Reset(void)
void UpdateVelocity(SensorTouch st);
double get_speed()
void set_speed(double sp)
void set_angle_delta(double ab)
double get_heading_angle() const
void set_heading_angle(double ha)
double get_max_speed()
void set_max_speed(double ms)
Private:
double heading_angle_

```
        double speed_
        double max_speed_
        double angle_delta_
RobotMotionHandler:
        Public:
        RobotMotionHandler()
        void Reset(void)
        void UpdateVelocityt(SensorEntityType se, SensorTypeEmit st, SensorProximity sp, SensorDistress sd);
        double get_speed()
        void set_speed(double sp)
        void set_angle_delta(double ab)
        void HeadingAngleInc(double ag)
        void HeadingAngleDec(double ag)
        double get_heading_angle() const
        void set_heading_angle(double ha)
        double get_max_speed()
        void set_max_speed(double ms)
        void set_type(entity_type t)
        std::string get_name(void)
        void set_name(std::string s)
        Private:
        std::string name_
        entity_type t_
        double heading_angle_
        double speed_
        double max_speed_
        double angle_delta_
SuperbotMotionHandler:
        Public:
        SuperbotMotionHandler()
        void Reset(void)
        void UpdateVelocityt(SensorEntityType se, SensorTypeEmit st, SensorProximity sp, SensorDistress sd);
        double get_speed()
        void set_speed(double sp)
        void set_angle_delta(double ab)
        void HeadingAngleInc(double ag)
        void HeadingAngleDec(double ag)
        double get_heading_angle() const
        void set_heading_angle(double ha)
        double get_max_speed()
        void set_max_speed(double ms)
        void set_type(entity_type t)
        std::string get_name(void)
        void set_name(std::string s)
        Private:
        std::string name_
        entity_type t_
        double heading_angle_
        double speed_
double max_speed_
double angle_delta_
```

PlayerMotionHandler:

```
Public:
PlayerMotionHandler()
void Reset(void)
void AcceptCommand(enum event_commands cmd);
void UpdateVelocity(SensorTouch st);
double get_speed()
void set_speed(double sp)
void set_angle_delta(double ab)
void HeadingAngleInc(double ag)
void HeadingAngleDec(double ag)
double get_heading_angle() const
void set_heading_angle(double ha)
double get_max_speed()
void set_max_speed(double ms)
void set_type(entity_type t)
Private:
entity_type t_
double heading_angle_
double speed_
double max_speed_
double angle_delta_
```

HomeBaseMotionHandler:

```
Public:
HomeBaseMotionHandler()
void Reset(void)
void UpdateVelocity(SensorTouch st);
double get_speed()
void set_speed(double sp)
void set_angle_delta(double ab)
void HeadingAngleInc(double ag)
void HeadingAngleDec(double ag)
double get_heading_angle() const
void set_heading_angle(double ha)
double get_max_speed()
void set_max_speed(double ms)
void set_type(entity_type t)

Private:
entity_type t_
double heading_angle_
double speed_
double max_speed_
double angle_delta_
```

5. COMPONENT DESIGN

Graphic Viwer:

MermberFunction: UpdateSimulation sends a message to the arena arena_->advanceTime(1). It has methods (DrawRobot, DrawObstacle, and so on) to draw a robot, obstacle, and the homebase, player. It holds all the On<key/mouse>Event() methods. It also contains a boolean paused_ (which is associated with the button member which is nanogui::Button *pause_btn_.)

Arena:

Call to AdvanceTime(dt) makes things happen. This sends message to each entity to update time by 1 for each dt with UpdateEntitiesTimeStep(), which in turn for each entity calls entity->TimeStepUpdate(1). After all entities updated for the 1 time step, checks for events by CheckForEntityOutOfBounds and CheckForEntityCollision:

player at charging station
robot at home base
robot with player
robot with freezed robot
superbot with player
collisions with obstacles
collisions with walls

And when users press the keyboard, call Accept(EventKeypress*) to receive information and change player's velocity and angle.

ArenaEntity:

A base class which can be derived, and provides the basic information for all entities, such as name, radius, position, color. All of those data can be accessed and changed via getters and setters function. And it will also update along with time step by call TimeStepUpdated(dt) function. Reset() which set the default data to this entity is called when game restarts. And is_mobile function is to differentiate the mobile entity and immobile entity.

ArenaImMobileEntity :

is_mobile() will return false, and because this is immobile, so it only has radius, position, and color.

ArenaMobileEntity:

Is_mobile will return true, and since this is mobile, so it should have heading angle, speed to adjust its velocity, so we can access and modify them by getters and setters . Call get_collision_delta() to get Collision delta, which is used to compute the distance between two entity to indicate whether they have collision or not. And When collision happens, it will Accept(EventCollisiton) or Accept(EventRecharge) to change entities' state by handling different kinds of collisions. Set name is not needed.

RechargeStation:

Inherited from ArenaImMobileEntity. We can get its name by calling name() (just one rechargeStation)

Obstacles:

Inherited from ArenaImMobileEntity. We can get its name and id by calling name()

Homebase:

Inherited from ArenaMobileEntity.Accept(EventCollisions) when it has collisions with other entities.Inherited from ArenaMobileEntity.Homebase will randomly change direction and get updated its position and volecity by calling TimestepUpdated(dt)

Robot :

Inherited from ArenaMobileEntity. Robot accepts EventCollisions, EventTypeEmit, EventProximity, EventDistressCall by Accept functions.

Superbot:

Using Strategy Pattern. Superbot accepts EventCollisions, EventTypeEmit, EventProximity, EventDistressCall by Accept functions.

Player:

Inherited from ArenaMobileEntity.Accept(EventCollisions) when it has collisions with other entities.Only Accept(EventRecharge) when it has collisions with Rechargestation.When it collides with rechargestation, calling ResetBattery() to renew the battery it has. Change heading angle and speed when receiving EventCmd() by calling HeadingAngleInc(),HeadingAngleDec(),set_speed().

EventBaseClass:

When event occurs, print the related event content by calling EmitMessage() EventCollision.Call get_collided() to know if the collision happens, and when collistion happens calling set_collided(), set_point_of_contact(), set_angle_of_contact() to collect information which ca can be passed to Sensor touch. Then SensorTouch will get those informations from this event.

EventDistressCall:

Call get_distressed() to know if the collision happens, and when robot have collisions with player calling set_distressed() since robot need to be freezed.

EventKeyPress:

When user press the key, the get_key_cmd will be called, and then the get_key() will be called motionhandler to see which key is pressed and how to change the state of its velocity.

EventProximity:

Check the distance between two entities, using Set_position_sensor and set_position_sensed, set_heading_angle_of_sensor(), and set_radius_of_sensed(). Calling those functions to collect data, and transfer those data to Sensorproximity, so the sensor proximity will call those getters to get information.
EventRecharge:

When player has a collision with rechargestation. EmitMessage ”Battery Recharger”

EventTypeEmit:
Set_entity_type() to set what the type the current entity will collide or have collided. And SensorEntityType will get this type by calling get_entity_type()

Sensor:
Using integer (1, 0) represents whether this sensor is on or off by calling get_activated() And set_activated; Reset all sensors when game restart by calling reset Accept all kinds of events which are generated from Arena.

SensorDistress:
Calling Accept(EventDistress* e) to receive this event, and set_activated() of this sensor. The state of this sensor is based on this event. Then velocity will be decided by motionhandler via get_activated.

SensorEntityType:
Calling Accept(EventTypeEmit* e) to receive this event, then set_activated, set_entity_type of this sensor. The state of this sensor is based on this event. Then the velocity will be decided by motionhandler according to get_entity_type().

SensorProximity:
This sensor has detection range, field of view, distance from other entity. Access and modify those via getters and setters. And call the Accept(EventProximity) to get information from this event. To return the distance between two entities whose information is passed by this event by calling sensor_reading(). If sensed entity is out of range, sensor_rading() will return -1. Otherwise, return the computed distance. The motionhandler will change the entity's velocity based on this distance by calling this function.

SensorTouch:
Receive this event by calling Accept(EventCollision) , and in order to gain the information from the event by calling setters, then motionhandler will change the velocity by calling those getters.

RobotMotionHandler:
updateVolocity(SensorTouch, SensorEntityType, SensorProximity, SensorDistress) First check if the sensor touch is activated, it it is, check the the type of collided entity. If the type is player, then become freezed (set speed to zero), then activate the sensorDistress; If the type is Homebase, then become to superbot; if the type is robot, then reset the speed as original speed (in case this robot is freezed); Then do the normal velocity changes.If the sensor touch is not activated, then to check the sensorProximity. If the distance is not -1, then we check the type. If the type is player, set the heading angle in the opposite direction. If the type is robot, check if the speed of that robot is 0. If the speed is 0, then set the heading angle to unfreeze that robot.

SuperbotMotionHandler:
updateVolocity(SensorTouch, SensorEntityType, SensorProximity, SensorDistress)
First check if the sensor touch is activated, it it is, check the the type of collided entity. If the type is player, then set player become freezed for sever second(set speed to zero); if the type is robot, then reset the speed as original speed ; Then do the regular velocity changes.
If the sensor touch is not activated, then to check the sensorProximity. If the distance is not -1, then we check the type. If the type is player, set the heading angle to set direction (go to freeze player). If the type is robot, check if the speed of that robot is 0. If the speed is 0, then set the heading angle to that robot to unfreeze that robot.

PlayerMotionHandler:
updatevlociy(SensorTouch)When the sensor touch is activated, change the velocity in a reasonable way.Receiving the information from the keyboard by HeadingAngleInc, HeadingAngleDec, Set_speed. Then update its velocity by calling getters.

HomeBaseMotionHandler:
updatevlocity(SensorTouch)When the sensor touch is activated, change the velocity in a reasonable way.

6. HUMAN INTERFACE DESIGN

6.1 Overview of User Interface

GUI generated by Graphic viewer
Keypress by user

6.2 Screen Objects and Actions

When Restart button is pressed in the GUI, all entities will go to the beginning position with original state

When Pause button is pressed in the GUI, all entities will stop until press the same button again.

When left, right keys pressed, the player will change heading angle

When up, down keys pressed, the player will change the speed, but the speed cannot over the max or less than 0.