# Priority 1 Design Document

This project develops a rudimentary robot simulator in which robot behavior is visualized within a graphics window. Besides, it could be a test environment for experimenting with autonomous robot control using sensor feedback. In this iteration, the robot simulator will move towards autonomous, intelligent robot behavior. The different stuffs between these two iterations are 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. However, if another autonomopus 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 orndinary Robot colliding with home base.

As far as the organization of this project, we can simply divide it to three partsL Robot Arena, Graphics Environment and GUI and User Input. Firstly I look at the Robot Arena. The Robot Arena is consisted with Robot, Player, SuperBot, obstacle, home base and charging station. 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 add three functions in Arena, which are "registerRobot(Robot *r), removeRobot(Robot *r) and notifyRobot()". And we need to add three more Accept function in robot.h, which are "void Accept(EventProximity * e);  void Accept(EventDistressCall * e); void Accept(EventTypeEmit * e);". Those function can update the observers.

Another change is the initialization way. 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.

Sensors:

Distress:

When Distress Sensor activated, those following entities would change like:

Robot: Unactivated Proximity Sensor and fixed heading angle of robot.

SuperBot: Unactivated Proximity Sensor and fixed heading angle of robot.

Player: Nothing.

Home Base: Nothing.

Recharge Station: Nothing.

Obstacles: Nothing.

Where is that information used to affect entity behavior?

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.

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

To sensed the distress signal, Distress Sensor need to know the radius and positions of sensed_entityand sensor_eneity. Because we need to determine whether the former one in the later one's range

Attributes: range_(double), sensed_pos (Position),

sensor_pos(Position), sensed_radius(double), isDistress(int)

Functions: double range() { return range_; } void range(double r) { range_ = r; }

int IsDistress() { return isDistress_; } void IsDistress(int d) { isDistress_ = d; }

```
Position SensedPos() { return sensed_pos; }

void SensedPos(Position p) { sensed_pos = p; }

Position SensorPos() { return sensor_pos; }

void SensorPos(Position p) { sensor_pos = p; }

 double SensedRad() { return sensed_radius; }

 void SensedRad(double rad) { sensed_radius = rad; }
```

Entity Type:

When Type Emit Sensor activated, those following entities would change like:

Robot: Return entity_type kRobot.

SuperBot: Return entity_type kSuperBot

Player: Return entity_type kPlayer

Home Base: Return entity_type kHomeBase

RechargeStation: Return entity_type kRechargeStation

Obstacle: Return entity_type kObstacle.

Where is that information used to affect entity behavior?

This sensor can store entity type then used this type in motion handler, to determine the motion of entity. The type of the entity emitting the signal can be sensed when it is within a defined range, but the direction of the signal cannot be determined. Sensor output is the enumerated entity type.

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

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

Attributes: range_(double), sensed_pos (Position),

sensor_pos(Position), sensed_radius(double), s_entity(enum entity_type)

Functions: double range() { return range_; } void range(double r) { range_ = r; }

enum entity_type get_entity() { return s_entity; }

void set_entity(enum entity_type t) { s_entity = t; }

Position SensedPos() { return sensed_pos; }

void SensedPos(Position p) { sensed_pos = p; }

Position SensorPos() { return sensor_pos; }

void SensorPos(Position p) { sensor_pos = p; }

double SensedRad() { return sensed_radius; }

void SensedRad(double rad) { sensed_radius = rad; }

Proximity:

When Proximity Sensor activated, those following entities would change like:

Robot: If there is a obstacle, then return the distance to the obstacle, if not, return -1.

SuperBot: Nothing.

Player: Nothing

Home Base: Nothing

RechargeStation: Nothing

Obstacle: Nothing

Where is that information used to affect entity behavior?

It used in the motion handler. When we recive a distance but not -1, robot need to turn an angle to avoid collision. 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.

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

To sensed the distance to the obstacle, Proximity Sensor has to use the field of vew and range to calculate whether or not the sensed entity is in the range of sensor.

Attributes: filed_of_view_(double), distance_(double), sensor_haeding_angle(double), sensed_pos (Position), sensor_pos(Position), sensed_radius(double)

Functions:

```
double filed_of_view () { return filed_of_view _; }
void filed_of_view (double fv) { filed_of_view _ = fv; }
double distance () { return distance_; }
void distance (double dist) { distance_ = dist; }
Position SensedPos() { return sensed_pos; }
void SensedPos(Position p) { sensed_pos = p; }
Position SensorPos() { return sensor_pos; }
void SensorPos(Position p) { sensor_pos = p; }
double SensedRad() { return sensed_radius; }
void SensedRad(double rad) { sensed_radius = rad; }
double SensedHd() { return sensed_heading_angle; }
void SensedHd(double hd) { sensed_heading_angle= hd; }
```

Touch

When Touch Sensor activated, those following entities would change like:

Robot: When robot touch obstacles or wall, then bouce away.

SuperBot: When SuperBot touch obstacles or wall, then bouce away.

Player: When Player touch obstacles or wall, then bouce away.

Home Base: Nothing

Recharge Station: Nothing.

Where is that information used to affect entity behavior?

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

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

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.

Attributes: point_of_contact_(Position), angle_of_contact_(double)

Functions:
```
double angle_of_contact () { return angle_of_contact_; }
void angle_of_contact(double a) { angle_of_contact_ = dist; }
Position point_of_contact() { return point_of_contact_; }
void point_of_contact(Position p) { point_of_contact_ = p; }
```

In the Event class, I create another three events named “event_proximity”, “event_distress_call” and “event_type_emit”. Those events provide information regarding the entities involved and the location of the event. Thus it contains the sensed position and sensor position. Besides, they need to contain the radius to calculate the distance to other entities. The events’pass the updated value to the sensor and sensor pass them to entity.

As far as motion_handler class, I add some functions which arguments are those sensors. Besides, I add the positions of sensed entity and sensor entity, radius of sensed entity and entity_type(enum) to the attributes. When sensor found the event happened and sensor's data was updated, then we can update the data in motion_handler. Then entities can use motion_handler to update themselves.

Additionally, the converting between Robot and SuperBot is a interesting point. I would like to use the stratage 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. Following this example, we can define a class named "BaseRobot". And it contains the constructor and some functions about Accept, getters and setters, TimestepUpdate, HeadingAngleInc/Dec, Reset and so on. Besides, the BaseRobot has to contain some attributes such as motion_handler, those all sensors, id and Player's battery. All of them can be defined as a pure virtual function. Then we create two class named Robot and SuperBot. They need to override all pure virtual functions, which need to show the different between Robot and SuperBot such as SuperBot do not have sensor_proximity and so on. After that, we can simply convert Robot and SuperBot. Each Robot who is hitted by the SuperBot would become a SuperBot. Those work need to be done in Arena.