Path finding

20175104 오정석

S3.ai.position

먼저 맵 각 타일의 정보를 담는 position 클래스를 선언하고, position 클래스를 생성할 때 생성자 내에서 heuristic과 이를 이용한 f 값 계산을 하도록 구현했다. 또한 이후 이 position 클래스 리스트를 정렬할 때 필요한 compareTo 함수를 오버라이딩 했고, 정렬의 기준은 f 값이 되도록 설정했다.

```
ublic class position implements Comparable<position>{
    int node_id;
    public double x;
    public double y;
    public int g;
    public int h;
    public int f;
    position parent;

    public position(double x, double y, double goal_x, double goal_y, int g, position parent) {
        this.x = x;
        this.y = y;
        this.h = (int)Math.abs(x-goal_x) + (int)Math.abs(y-goal_y);
        this.g = g;
        this.f = g+h;
        this.parent = parent;
    }
```

```
@Override
public int compareTo(position P) {
    if(this.f > P.f) {
        return 1;
    }
    else {
        return -1;
    }
}
```

S3.ai.AStar

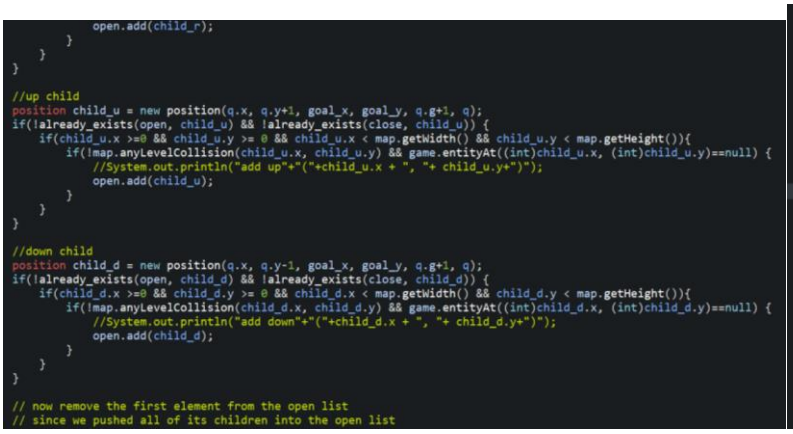
AStar 클래스 내부에서는 먼저 start와 goal의 x와 y 좌표를 저장할 double 변수와 open, close 리스트를 만들었다. 이 변수들은 AStar의 생성자에서 초기화 시켰다.

```
public class AStar {
    public S3 game;
    public S3Map map;
    S3PhysicalEntity entity;
    // variables to keep the start point and the goal
    double start_x, start_y, goal_x, goal_y;

    // lists to keep the tree of A*
    List<position> open;
    List<position> close;
```

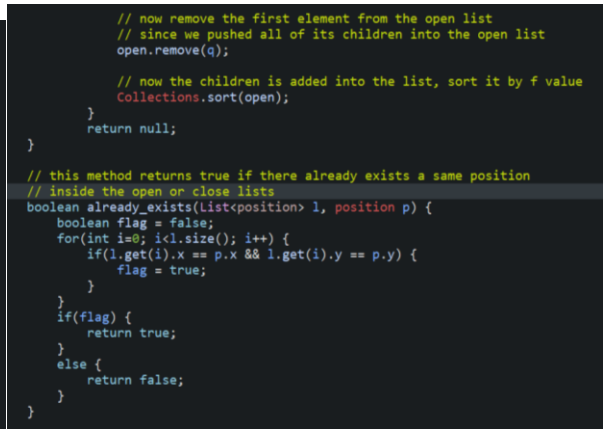
```
public AStar(double start_x, double start_y, double goal_x, double goal_y,
        S3PhysicalEntity i_entity, S3 the_game) {
        // initialize the variables and make the list
        this.game = the_game;
        this.entity = i_entity;
        this.start_x = start_x;
        this.start_y = start_y;
        this.goal_x = goal_x;
        this.goal_y = goal_y;
        this.map = the_game.getMap();
        open = new ArrayList<position>();
        close = new ArrayList<position>();
}
```

computePath에서는 먼저 open 리스트에 시작점을 add한 상태로 while 반복문을 돌린다. 반복문 내부에서는 현재 position의 상하좌우 child를 보고, open 도는 close 리스트에 없고, 충돌하는 객체가 없으며, 맵 내부 범위에 해당 position의 좌표가 들어와있는 경우, 이를 open 리스트에 추가하도록 했다. 또한 이미 extension이 끝난 position은 close 리스트로 add 하도록 구현했다. 추가적으로, 이미 한 번 거쳤던 position인지를 알아내기 위해 already_exists 함수를 구현했다. 만약 이미 검사했던 position이라면, already_exists 함수는 true를 반환하고, 반대의 경우 false를 반환한다.

```
public List<Pair<Double, Double>> computePath() {

        // make a list to keep the final path
        ArrayList<Pair<Double, Double>> path = new ArrayList<Pair<Double, Double>>();

        // add the starting point to the open list
        open.add(new position(start_x, start_y, goal_x, goal_y, 0, null));

        // while there is an element inside the open list
        // find next position to move that leads to goal
        while(!open.isEmpty()) {
            // get the first element from the sorted open list
            position q = open.get(0);

            // the first element is now closed, so add it into the close list
            close.add(q);

            //System.out.println("("+q.x + ", "+ q.y+")");

            // if the current position is same as the goal position
            // end the while loop and return the path
            if(q.x==goal_x && q.y==goal_y) {
                //System.out.println("end");
                Pair<Double, Double> p = new Pair<Double, Double>(q.x, q.y);
                path.add(p);
                position parent = q.parent;
                while(parent!=null) {
                    path.add(new Pair<Double, Double>(parent.x, parent.y));
                    parent = parent.parent;
                }

                // reverse the order of path
                Collections.reverse(path);
                return path;
            }


            //left child
            // if there isn't any collision, the position is inside the map,
            // and the child doesn't exist inside open and close list
            // then push the child into the open list
            position child_l = new position(q.x-1, q.y, goal_x, goal_y, q.g+1, q);
            if(!already_exists(open, child_l) && !already_exists(close, child_l)) {
                if(child_l.x >=0 && child_l.y >= 0 && child_l.x < map.getWidth() && child_l.y < map.getHeight()){
                    if(!map.anyLevelCollision(child_l.x, child_l.y) && game.entityAt((int)child_l.x, (int)child_l.y)==null) {
                        //System.out.println("add left" + "("+child_l.x + ", "+ child_l.y+")");
                        open.add(child_l);
                    }
                }
            }

            //right child
            position child_r = new position(q.x+1, q.y, goal_x, goal_y, q.g+1, q);
            if(!already_exists(open, child_r) && !already_exists(close, child_r)) {
                if(child_r.x >=0 && child_r.y >= 0 && child_r.x < map.getWidth() && child_r.y < map.getHeight()){
                    if(!map.anyLevelCollision(child_r.x, child_r.y) && game.entityAt((int)child_r.x, (int)child_r.y)==null) {
                        //System.out.println("add right"+"("+child_r.x + ", "+ child_r.y+")");
```



```
            open.add(child_r);
        }
    }
}

//up child
position child_u = new position(q.x, q.y+1, goal_x, goal_y, q.g+1, q);
if(!already_exists(open, child_u) && !already_exists(close, child_u)) {
    if(child_u.x >=0 && child_u.y >= 0 && child_u.x < map.getWidth() && child_u.y < map.getHeight()){
        if(!map.anyLevelCollision(child_u.x, child_u.y) && game.entityAt((int)child_u.x, (int)child_u.y)==null) {
            //System.out.println("add up"+"("+child_u.x + ", "+ child_u.y+")");
            open.add(child_u);
        }
    }
}

//down child
position child_d = new position(q.x, q.y-1, goal_x, goal_y, q.g+1, q);
if(!already_exists(open, child_d) && !already_exists(close, child_d)) {
    if(child_d.x >=0 && child_d.y >= 0 && child_d.x < map.getWidth() && child_d.y < map.getHeight()){
        if(!map.anyLevelCollision(child_d.x, child_d.y) && game.entityAt((int)child_d.x, (int)child_d.y)==null) {
            //System.out.println("add down"+"("+child_d.x + ", "+ child_d.y+")");
            open.add(child_d);
        }
    }
}

// now remove the first element from the open list
// since we pushed all of its children into the open list
```



```
            // now remove the first element from the open list
            // since we pushed all of its children into the open list
            open.remove(q);

            // now the children is added into the list, sort it by f value
            Collections.sort(open);
        }
        return null;
    }

    // this method returns true if there already exists a same position
    // inside the open or close lists
    boolean already_exists(List<position> l, position p) {
        boolean flag = false;
        for(int i=0; i<l.size(); i++) {
            if(l.get(i).x == p.x && l.get(i).y == p.y) {
                flag = true;
            }
        }
        if(flag) {
            return true;
        }
        else {
            return false;
        }
    }
```

S3.ai.S3Map

AStar에서 다른 물체와 충돌하는지 여부를 판단하기 위해 anyLevelCollision 함수가 존재하나, 이를 직접적인 좌표를 받아 확인하는 함수가 필요해 오버라이딩했다.

```
// changed to get the position of specific point
public boolean anyLevelCollision(double x, double y) {
    return layers[1].collidesWith(x, y);
}
```

S3.ai.S3MapLayer

S3.ai.S3Map.anyLevelCollision에서 막힌 길인지 아닌지를 판단하기 위해 collidesWith 함수를 호출하는데, 이 함수 역시 직접적인 좌표를 통해 알 수 있도록 오버라이딩한 함수를 구현했다.

```
// changed to get the parameter of specific position
// if there is anything that collides with that position,
// collidesWidth will return true
public boolean collidesWith(double pos_x, double pos_y) {
    return !(map[(int)pos_x][(int)pos_y] instanceof WOGrass);
}
```