<From value.h>

```c
#ifndef _VALUES_
#define _VALUES_

#define HEIGHT 100
#define WIDTH 100
#define INIT 0

#define VISION_DISTANCE 10
#define VISION_FIELD 60
#define ROTATION_SPEED 40
#define MEAN_PERIOD 2

#define NB_LEADERS 10
#define K_MEAN_ACCURACY 20
#define CROWD_DISTANCE 5

#endif
```

<From entity/guard.h>

```c
#ifndef _GUARD_
#define _GUARD_

#include "position.h"
#include "../scene/scene.h"

typedef struct{
    position there;
    int dx;
    int dy;
    int err;
    int angle;
    bool moving;
    bool rotating;
} goal; //Nécéssaire pour les mouvements simple

typedef struct behave_option_t behave_option;
typedef struct guard_t guard;
typedef struct guard_list_t guard_list;

struct behave_option_t {
    int* angle_list;
    position* position_list;
    int list_size;
    int current_task;
    void (*move_function)(guard_list*,scene_object*,int,int);
};//Permet la complexité de comportement

struct guard_t{
    position* pos;
    int angle;
    int behave;
    int rotation_speed;
    goal to_do;
    behave_option option;
};

struct guard_list_t {
    int nb;
    int color;
    int color2;
    guard** tab;
};

guard* new_guard(int,int,int,position*);
void delete_guard(guard*);
guard_list* new_guard_list(int,int,int,int,int);
void delete_guard_list(guard_list*);

void initializer(guard*);

guard* copy_guard(guard*,int,int);
guard_list* copy_guard_list(guard_list*,int,int);
void edit_behaves(guard_list*,int);

void behave(guard_list*,scene_object*,int);

void guard_rotate_by(guard*,int);
void guard_rotate_to(guard*,int);
void guard_move_to(guard*,position,scene_object*);
void guard_natural_move_to(guard*,position,scene_object*);

int* find_leaders(guard_list*);
void arange_gards(guard_list*);

void nothing(guard*);
void rotation(guard*);
void random_line(guard*);
void corridor(guard*);
void centred_square(guard*);
void mousaid(guard*);
void snake(guard*);

void speed_rotation_fonction(guard_list*,scene_object*,int,int);
void classic_move_fonction(guard_list*,scene_object*,int,int);
void random_move_fonction(guard_list*,scene_object*,int,int);
void mousaid_move_fonction(guard_list*,scene_object*,int,int);
void snake_move_fonction(guard_list*,scene_object*,int,int);

#endif
```

<From entity/player.h>

```c
#ifndef _PLAYER_
#define _PLAYER_

#include "position.h"
#include "../tools/stack.h"
#include "../tools/priority.h"
#include "../scene/scene.h"

typedef struct {
    position* pos;
} player;

player* new_player(int,int);
void delete_player(player*);

void move_left(position*,scene_object*);
void move_right(position*,scene_object*);
void move_up(position*,scene_object*);
void move_down(position*,scene_object*);

void naive_path(player*,scene_object**);
void A_path(player*,scene_object**,int,int,stack*,priority_list*);

void multicore_path(player*,scene_object**,int,int,stack*,priority_list*,int);
void path(player*,scene_object**,int,int,stack*,priority_list*);
int check_and_move(player*,scene_object*,scene_object*,int,int,
                   void (*f)(position*, scene_object*));

#endif
```

<From entity/position.h>

```c
#ifndef _POSITION_
#define _POSITION_

#include <stdbool.h>

typedef struct {
    int x;
    int y;
} position;

typedef struct {
    int size;
    position** tab;
} list_position;

position* new_position(int,int);
position* copy_position(position*);
void delete_position(position*);

list_position* new_list_position(int);
list_position* copy_list_position(list_position*);
```

```c
void delete_list_position(list_position*);

void moveto(position*,int,int);

bool equals(position*,position*);

float range(position*,position*);

typedef struct q_e queue_elt;
struct q_e {
    position pos;
    queue_elt* next;
    queue_elt* prev;
};

typedef struct {
    queue_elt* end;
    queue_elt* start;
} queue_position;

queue_elt* new_queue_elt(position);
void delete_queue_elt(queue_elt*);

queue_position* new_queue_position();
void delete_queue_position(queue_position*);

void enqueue(queue_position*,queue_elt*);
queue_elt* dequeue(queue_position*);

queue_position* copy_queue(queue_position*);

typedef struct t_n tree_node;
struct t_n {
    tree_node** nexts;
    int nb_nexts;
    int nb_deleted;
    tree_node* prev;
    int name;
    tree_node* start;
    position pos;
};

tree_node* new_tree_node(position,tree_node*,tree_node*);
void add_next_tree_node(tree_node*,tree_node*);
void delete_tree_node(tree_node*);
void delete_tree_node_rac(tree_node*);

tree_node* new_start_tree_node(tree_node*);

#endif
```

<From old_function/threads.h>

```c
#ifndef _THREADS_

#define _THREADS_

#include "stack.h"
#include "priority.h"
#include "../scene/scene.h"
#include "../entity/player.h"
#include <pthread.h>

typedef struct {
    scene_object** future;
    int time;
    int accuracy;
    position* end;
    stack* closedList;
    priority_list* openList;
    pthread_mutex_t* m_prio;
    pthread_mutex_t* m_stack;
} A_arg;

void* multicore_A_star(void*);

#endif
```

<From scene/scene.h>

```c
#ifndef _SCENE_
#define _SCENE_

typedef struct {
    int height;
    int width;
    int** grid;
    int init;
    int time;
} scene_object;

scene_object* new_scene(int,int,int);
void scene_init(scene_object*,int);
void delete_scene(scene_object*);

void wait(scene_object*);
scene_object* copy_scene(scene_object*);

#endif
```

<From scene/simulation.h>

```c
#ifndef _SIMULATION_

#define _SIMULATION_

#include <stdbool.h>
#include <stdio.h>
#include "scene.h"
#include "../entity/player.h"
#include "../entity/guard.h"

int simulate(scene_object*,guard_list*,player*,int,int,bool,FILE*);

#endif
```

<From tools/draw.h>

```c
#ifndef _DRAW_
#define _DRAW_

#include "priority.h"
#include "../entity/position.h"
#include "../scene/scene.h"
#include <stdlib.h>

void check_position(position*,scene_object*);
void draw_position(position*,scene_object*,int);
void draw_cross(position*,scene_object*,int);

list_position* select_circle(position*,int);
list_position* select_arc(position*,int,int,int);
void check_selection(list_position*,scene_object*);
list_position* delete_double(list_position*);

void _draw_line(position*,position*,scene_object*,int,int,bool);
void draw_line(position*,position*,scene_object*,int,int);
void draw_line_in_cross(position*,position*,scene_object*,int,int);

void draw_cone(position*,int,int,int,scene_object*,int,int);
//collorie plusieur fois la même case mais permet les obstacles
void
draw_cone_with_cross(position*,int,int,int,scene_object*,int,int);
void quick_draw_cone(position*,int,int,int,scene_object*,int,int);
void print_trajectory(scene_object*,priority_list*,int);

void clean_holl(scene_object*,int,int);

#endif
```

<From tools/graph.h>

```

#ifndef _GRAPH_
#define _GRAPH_

#include "../scene/scene.h"
#include "../entity/position.h"
#include "stack.h"
#include "priority.h"

list_position* neighbors2(scene_object*,position*,int,int);
list_position* neighbors(scene_object*,position*,int,int);
position*
A_star(scene_object**,int,int,position*,stack*,priority_list*);

#endif

--------------------------------------------------------------------------------


<From tools/out.h>


#ifndef _OUT_
#define _OUT_

#include <stdio.h>
#include "../scene/scene.h"

char* convert_to_string(int);
void display_grid(int**,int,int,FILE*);
void display_scene(scene_object*,FILE*);

#endif

--------------------------------------------------------------------------------


<From tools/priority.h>


#ifndef _PRIORITY_
#define _PRIORITY_

#include <stdbool.h>
#include "../entity/position.h"

typedef struct{
    int prio;
    int cout;
    position* node;
    tree_node* way;
}data;

typedef struct {
    int size;
    data** tas;
} priority_list;


data* new_data(int,position*,tree_node*,int);
void delete_data(data*);

priority_list* new_priority_list(int);
bool empty_priority_list(priority_list*);

void swap(int,int,priority_list*);
void percolate_up(int,priority_list*);
int choose_son(int,priority_list*);
void percolate_down(int,priority_list*);

void insert(data*,priority_list*);
data* remove_rac(priority_list*);

bool is_in_priority(priority_list*,position*,int);

void delete_priority_list(priority_list*);

void adjust_priority_list(priority_list*,position*,position);

#endif


--------------------------------------------------------------------------------


<From tools/stack.h>


#ifndef _stack_
#define _stack_

#include <stdbool.h>
#include "../entity/position.h"

typedef struct s_e stack_elt;
struct s_e{
    position* pos;
    int time;
    stack_elt* next;
};

typedef struct{
    stack_elt* top;
}stack;

stack_elt* new_stack_elt(position*,int);
void delete_stack_elt(stack_elt*);

stack* new_stack();
void delete_stack(stack*);

void enstack(stack_elt*,stack*);
stack_elt* destack(stack*);

bool empty_stack(stack*);
bool is_in_stack(position*,int,stack*);

void adjust_stack(stack*,int);

#endif


--------------------------------------------------------------------------------


<From main.c>


#include <stdio.h>
#include <stdlib.h>
#include <stdbool.h>
#include <time.h>
#include <string.h>
#include <pthread.h>
#include "values.h"
#include "entity/guard.h"
#include "entity/player.h"
#include "scene/scene.h"
#include "scene/simulation.h"

#define SOURCE_NAME_1 "./out/score/"
#define SOURCE_NAME_2 "./out/rotation/"
#define EXT ".txt"
#define MAX_BEHAVE 4

char* filled_zero(int k){
    char* res;
    if(k>=0){
        res = malloc((k+1)*sizeof(char));
        for(int i=0; i<k; i++){
            res[i] = '0';
        }
        res[k] = '\0';
    }
    else{
        res = malloc(sizeof(char));
        res[0] = '\0';
    }
    return res;
}

int nb_zeros(int k){
    if(k==0){
        return 1;
    }
    int cur = k;
    int res = 0;
    while(cur != 0){
        cur = cur/10;
        res++;
    }
    return res;
```

```c
}

char* new_name(int behave, int number, char* source_name){
    char name[20];
    char* postname = malloc(30*sizeof(char));
    char* zero1 = filled_zero(2-nb_zeros(behave));
    char* zero2 = filled_zero(3-nb_zeros(number));
    sprintf(name,"%s%d_%s%d",zero1,behave,zero2,number);
    sprintf(postname,"%s",source_name);
    char* filename = strcat(name,EXT);
    char* file = strcat(postname,filename);
    free(zero1);
    free(zero2);
    return file;
}

guard_list** copy_lab(guard_list** storage, int size, int behave, int r_speed){
    guard_list** lab = malloc(size*sizeof(guard_list*));
    for(int i=0; i<size; i++){
        lab[i] = copy_guard_list(storage[i],behave,r_speed);
    }
    return lab;
}

void delete_lab(guard_list** lab, int size){
    for(int i=0; i<size; i++){
        delete_guard_list(lab[i]);
    }
    free(lab);
}

typedef struct {
    int k;
    int number;
    bool behave;
    int behave_angle;
    int accuracy;
    guard_list** storage;
} k_sim_arg;

void* thread_simulation(void* parameter){
    k_sim_arg* args = parameter;
    guard_list** lab;
    char* file;
    if(args->behave){
        lab = copy_lab(args->storage,args->k,args->behave_angle,ROTATION_SPEED);
        file = new_name(args->behave_angle,args->number,SOURCE_NAME_1);
    }
    else{
        lab = copy_lab(args->storage,args->k,6,args->behave_angle);//Changer le comportement
        file = new_name(args->behave_angle,args->number,SOURCE_NAME_2);
    }
    FILE* score = fopen(file,"w");//pour conserver les précédents

    for(int i=0; i<args->k; i++){
        scene_object* scene = new_scene(WIDTH,HEIGHT,INIT);
        player* bot = new_player(WIDTH/2,0);
        if(args->behave_angle == 0){
            fprintf(score,"%d\n",simulate(scene,lab[i],bot,2,args->accuracy,false,NULL));
        }
        else{
            fprintf(score,"%d\n",simulate(scene,lab[i],bot,5,args->accuracy,false,NULL));
        }
        delete_scene(scene);
        delete_player(bot);
    }
    delete_lab(lab,args->k);
    fclose(score);
    free(file);
    pthread_exit(NULL);
}

void k_simulation(int k,int number,int behave_start,int behave_end, int accuracy){
    printf("number : %d\n",number);
    guard_list** storage = malloc(k*sizeof(guard_list*));
    for(int i=0; i<k; i++){
        storage[i] = new_guard_list(number,0,ROTATION_SPEED,1,4);
    }
    pthread_t* core = malloc((behave_end-behave_start+1)*sizeof(pthread_t));
    k_sim_arg* parametres = malloc((behave_end-behave_start+1)*sizeof(k_sim_arg));

    for(int behave=behave_start; behave<=behave_end; behave++){
        k_sim_arg parametre = {k,number,true,behave,accuracy,storage};
        parametres[behave-behave_start] = parametre;
        pthread_create(&core[behave-behave_start],NULL,thread_simulation,&parametres[behave-behave_start]);
    }
    for(int behave=behave_start; behave<=behave_end; behave++){
        pthread_join(core[behave-behave_start],NULL);
    }

    free(core);
    free(parametres);
    delete_lab(storage,k);
}

void simulate_from_to(int k, int n, int m, int pas, int range, int accuracy){
    for(int i=n; i<=m; i+= pas){
        k_simulation(k,i,5,6,accuracy); //Selectionner la Range
    }
}

void k_simulation_rotation(int k,int number,int angle_start,int angle_end, int pas, int accuracy){
    printf("number : %d\n",number);
    guard_list** storage = malloc(k*sizeof(guard_list*));
    for(int i=0; i<k; i++){
        storage[i] = new_guard_list(number,0,ROTATION_SPEED,1,4);
    }
    pthread_t* core = malloc((angle_end-angle_start+1)*sizeof(pthread_t));
    k_sim_arg* parametres = malloc((angle_end-angle_start+1)*sizeof(k_sim_arg));

    for(int r_s=angle_start; r_s<=angle_end; r_s+=pas){
        k_sim_arg parametre = {k,number,false,r_s,accuracy,storage};
        parametres[r_s-angle_start] = parametre;
        pthread_create(&core[r_s-angle_start],NULL,thread_simulation,&parametres[r_s-angle_start]);
    }
    for(int r_s=angle_start; r_s<=angle_end; r_s+=pas){
        pthread_join(core[r_s-angle_start],NULL);
    }

    free(core);
    free(parametres);
    delete_lab(storage,k);
}

void find_angle_rotation(int angle_start, int angle_end, int angle_pas, int k,
                         int n, int m, int pas, int accuracy){
    for(int i=n; i<=m; i+=pas){
        k_simulation_rotation(k,i,angle_start,angle_end,angle_pas,accuracy);
    }
}

int main(int argc, char **argv){
    int seed = time(NULL);
    // seed = 1716554888;
    srand(seed);
    printf("Seed : %d\n",seed);

    if(argc < 6){
        printf("\nWARRING !!!\n \
        \nNeed 4 parametre (in this order) :\
        \n\t-k : the number of simulation per situation\
        \n\t-n and m : simule from n to m guards\
        \n\t-pas : it means what it means\
        \n\t-accuracy : accuracy of the player\n\n");
    }
    else{
        int k = atoi(argv[1]);
        int n = atoi(argv[2]);
        int m = atoi(argv[3]);
        int pas = atoi(argv[4]);
        int accuracy = atoi(argv[5]);
        simulate_from_to(k,n,m,pas,MAX_BEHAVE,accuracy);
        // find_angle_rotation(40,60,2,k,n,m,pas,accuracy);
    }

    return 0;
}

--------------------------------------------------------------------------------


<From entity/guard.c>


#include <stdio.h>
#include <stdlib.h>
#include <math.h>
```

```c
#include "player.h"
#include "guard.h"
#include "../tools/draw.h"
#include "../values.h"

guard* new_guard(int angle, int behave, int rotation_speed, position* pos){
    guard* res = calloc(1,sizeof(guard));
    res->angle = angle;
    res->behave = behave;
    res->rotation_speed = rotation_speed;
    res->pos = pos;
    res->to_do.angle=0;
    res->to_do.there = *pos;
    initializer(res);
    return res;
}

void delete_guard(guard* guard){
    delete_position(guard->pos);
    free(guard->option.position_list);
    free(guard->option.angle_list);
    free(guard);
}

guard_list* new_guard_list(int nb, int behave,int r_speed ,int color, int color2){
    guard_list* res = malloc(sizeof(guard_list));
    res->nb = nb;
    res->color = color;
    res->color2 = color2;
    res->tab = malloc(nb*sizeof(guard*));
    for(int i=0; i<nb; i++){
        position* pos;
        int x = rand()%100;
        int y = rand()%100;
        if(x<=WIDTH/2+VISION_DISTANCE && x>=WIDTH/2-VISION_DISTANCE
           && y<=VISION_DISTANCE)
            i--;
        else{

            pos = new_position(x,y);
            res->tab[i] = new_guard((360 + rand()%360)%360,behave,r_speed,pos);
        }
    }
    return res;
}

void delete_guard_list(guard_list* guard_list){
    for(int i=0; i<guard_list->nb; i++){
        delete_guard(guard_list->tab[i]);
    }
    free(guard_list->tab);
    free(guard_list);
}

guard* copy_guard(guard* guard, int behave, int r_speed){
    position* pos = copy_position(guard->pos);
    return new_guard(guard->angle,behave,r_speed,pos);
}

guard_list* copy_guard_list(guard_list* guards, int behave, int r_speed){
    guard_list* res = malloc(sizeof(guard_list));
    res->nb = guards->nb;
    res->color = guards->color;
    res->color2 = guards->color2;
    res->tab = malloc(res->nb*sizeof(guard*));
    for(int i=0; i<res->nb; i++){
        res->tab[i] = copy_guard(guards->tab[i],behave,r_speed);
    }
    return res;
}

void edit_behaves(guard_list* guards, int behave){
    for(int i=0; i<guards->nb; i++){
        guards->tab[i]->behave = behave;
        free(guards->tab[i]->option.angle_list);
        free(guards->tab[i]->option.position_list);
        initializer(guards->tab[i]);
    }
}

void behave(guard_list* guards, scene_object* scene, int time){
    scene_init(scene,scene->init);
    int* leader = find_leaders(guards);
    if(time == 0){
        arange_gards(guards);
    }
    for(int i=0;i<guards->nb;i++){
        guard* guard = guards->tab[i];
        guard->option.move_function(guards,scene,i,leader[i]);
        scene->time = time;
        draw_cone_with_cross(guard->pos,VISION_DISTANCE,
            guard->angle,VISION_FIELD,scene,guards->color,guards->color2);
        clean_holl(scene,guards->color,guards->color2);
    }
    free(leader);
}

void guard_rotate_by(guard* guard, int deg){
    goal* target = &guard->to_do;
    if(target->angle < guard->rotation_speed){
        target->angle = deg;
    }

    int signe = target->angle > 0 ? 1 : -1;
    if(target->angle*signe > guard->rotation_speed){
        target->angle += (-guard->rotation_speed)*signe;
        guard->angle += guard->rotation_speed*signe;
        target->rotating = true;
    }
    else{
        guard->angle += target->angle;
        target->angle = 0;
        target->rotating = false;
    }
    guard->angle = (360+guard->angle)%360;
}

void guard_rotate_to(guard* guard, int angle){
    int deg = abs(angle)-guard->angle;
    int signe = deg>0 ? 1 : -1;
    if(angle>=0){
        guard_rotate_by(guard,deg*signe>180 ? (deg*signe-360)*signe : deg);
    }
    else{
        guard_rotate_by(guard,deg*signe>180 ? deg : (deg*signe-360)*signe);
    }
}

void guard_move_to(guard* guard, position pos, scene_object* scene){
    goal* target = &guard->to_do;
        position* p1 = guard->pos;
        position* p2 = &target->there;
        int err2 = 2 * target->err;
        if (err2 + target->dy > 0) {
            target->err -= target->dy;
            p1->x += (p1->x - p2->x < 0) ? 1 : -1;
        }
        if (0 < target->dx - err2) {
            target->err += target->dx;
            p1->y += (p1->y - p2->y < 0) ? 1 : -1;
        }
        if(equals(p1,p2)){
            target->there = pos;
            check_position(&target->there,scene);
            target->dx = abs(p2->x - p1->x);
            target->dy = abs(p2->y - p1->y);
            target->err = target->dx - target->dy;
            target->moving = false;
        }
}

void guard_natural_move_to(guard* guard, position pos, scene_object* scene){
    goal* target = &guard->to_do;
    int x = pos.x-guard->pos->x;
    int y = pos.y-guard->pos->y;
    int angle = guard->angle;
    if(x!=0 || y!=0){
        angle = ((int)(-atan2(y,x)*(180.0)/(3.14)) + 360)%360;
    }
    if(!target->moving){
        guard_rotate_to(guard,angle);
    }
    if(!target->rotating || target->moving){
        target->moving = true;
        guard_move_to(guard,pos,scene);
    }
}

void _update_classe(guard_list* guards,int* res, int* classe, bool last){
    for(int i=0; i<guards->nb;i++){
        float min_range = range(guards->tab[i]->pos,guards->tab[res[0]]->pos);
        for(int j=1; j<NB_LEADERS;j++){
            float r = range(guards->tab[i]->pos,guards->tab[res[j]]->pos);
            if(r<min_range){
                min_range = r;
                classe[i] = (last?res[j]:j);
            }
```

```
        }
    }
}

void _closest_guards(guard_list* guards, int* res, position* tmp){
    for(int j=0; j<NB_LEADERS;j++){
        float min_range = range(guards->tab[0]->pos,&tmp[j]);
        for(int i=1; i<guards->nb; i++){
            float r = range(guards->tab[i]->pos,&tmp[j]);
            if(r<min_range){
                min_range = r;
                res[j] = i;
            }
        }
    }
}

void _update_res(guard_list* guards, int* res, int* classe){
    position* tmp = calloc(NB_LEADERS,sizeof(position));
    int* count = calloc(NB_LEADERS,sizeof(int));
    for(int i=0;i<guards->nb;i++){
        tmp[classe[i]].x += guards->tab[i]->pos->x;
        tmp[classe[i]].y += guards->tab[i]->pos->y;
        count[classe[i]] ++;
    }
    for(int i=0;i<NB_LEADERS;i++){
        int n = (count[i]>0?count[i]:1);
        tmp[i].x = tmp[i].x/n;
        tmp[i].y = tmp[i].y/n;
    }
    free(count);
    _closest_guards(guards,res,tmp);
    free(tmp);
}

int* find_leaders(guard_list* guards){
    int* res = malloc(NB_LEADERS*sizeof(int));
    int* classe = calloc(guards->nb,sizeof(int));
    for(int i=0;i<NB_LEADERS;i++){
        res[i] = i%guards->nb;
    }
    for(int i=0;i<K_MEAN_ACCURACY;i++){
        _update_classe(guards,res,classe,false);
        _update_res(guards,res,classe);
    }
    _update_classe(guards,res,classe,true);
    free(res);
    return classe;
}

void _guards_colone(guard_list* guards, int* res, position* tmp){
    for(int i=0; i<guards->nb; i++){
        float min_range = range(guards->tab[i]->pos,&tmp[0]);
        for(int j=1; j<guards->nb;j++){
            float r = range(guards->tab[i]->pos,&tmp[j]);
            if(r<min_range){
                min_range = r;
                res[i] = j;
            }
        }
    }
}
```

```

void _swap_guard(guard** tab, int i, int j){
    guard* tmp = tab[i];
    tab[i] = tab[j];
    tab[j] = tmp;
}

void arange_gards(guard_list* guards){
    int size = guards->nb;
    int* colone = calloc(size,sizeof(int));
    position* tmp = malloc(size*sizeof(position));
    bool* is_use = calloc(size,sizeof(bool));
    for(int i=0; i<size; i++){
        tmp[i].x = (int)(i*(double)WIDTH/size);
        tmp[i].y = HEIGHT/2;
    }
    _guards_colone(guards,colone,tmp);
    for(int i=0; i<size; i++){
        for(int j=0; j<size; j++){
            int k = (colone[i]+j)%size;
            if(!is_use[k]){
                is_use[k] = true;
                _swap_guard(guards->tab,i,k);
                break;
            }
        }
    }
    free(tmp);
    free(colone);
    free(is_use);
}

//____________________BEHAVES____________________

void initializer(guard* res){
    void (*init)(guard*);
    switch (res->behave){
            case 1:
                init = &rotation;
                break;
            case 2:
                init = &random_line;
                break;
            case 3:
                init = &corridor;
                break;
            case 4:
                init = &centred_square;
                break;
            case 5:
                init = &mousaid;
                break;
            case 6:
                init = &snake;
                break;
            default:
                init = &nothing;
                break;
        }
    init(res);
}
```

```
void nothing(guard* guard){
    guard->option.angle_list = calloc(1,sizeof(int));
    guard->option.position_list = malloc(sizeof(position));
    guard->option.position_list[0] = *guard->pos;
    guard->option.list_size = 1;
    guard->option.move_function = &classic_move_fonction;
}

void rotation(guard* guard){
    nothing(guard);
    guard->option.angle_list[0] = 360;
    guard->option.move_function = &speed_rotation_fonction;
}

void random_line(guard* guard){
    nothing(guard);
    guard->option.move_function = &random_move_fonction;
}

void corridor(guard* guard){
    guard->option.angle_list = calloc(2,sizeof(int));
    guard->option.position_list = malloc(2*sizeof(position));
    guard->option.position_list[0] = *guard->pos;
    guard->option.position_list[1] = *guard->pos;
    switch ((guard->angle-45)/90){
    case 0:
        guard->option.position_list[1].x += 20;
        break;
    case 1:
        guard->option.position_list[1].y -= 20;
        break;
    case 2:
        guard->option.position_list[1].x -= 20;
        break;
    default:
        guard->option.position_list[1].y += 20;
        break;
    }
    scene_object* scene = new_scene(WIDTH,HEIGHT,0);
    check_position(&guard->option.position_list[1],scene);
    delete_scene(scene);
    guard->option.list_size = 2;
    guard->option.move_function = &classic_move_fonction;
}

int _min_dist(guard* guard){
    int res = 0;
    position* list = guard->option.position_list;
    int size = guard->option.list_size;
    for(int i=1;i<size;i++){
        if(range(guard->pos,&list[i]) < range(guard->pos,&list[res])){
            res = i;
        }
    }
    if((guard->pos->x == list[res].x && guard->pos->x ==
list[(res+1)%size].x)
        || (guard->pos->y == list[res].y && guard->pos->y ==
list[(res+1)%size].y)){
        for(int i=0; i<size/2; i++){
            position tmp = list[i];
            list[i] = list[size-1-i];
            list[size-1-i] = tmp;
```

```
        }
        res = size-1-res;
    }
    return res;
}

void centred_square(guard* guard){
    int mid_x = WIDTH/2;
    int mid_y = HEIGHT/2;
    int r_x = abs(guard->pos->x - mid_x);
    int r_y = abs(guard->pos->y - mid_y);
    if(r_x<VISION_DISTANCE && r_y<VISION_DISTANCE){
        rotation(guard);
    }
    else{
        guard->option.list_size = 4;
        guard->option.angle_list = calloc(4,sizeof(int));
        guard->option.position_list = malloc(4*sizeof(position));
        int r = fmax(r_x,r_y);
        for(int i=0;i<4;i++){
            int s[2] = {1,-1};
            position p = {mid_x + s[i/2]*r, mid_y + s[(i+(i>1?1:0))
%2]*r};

            guard->option.position_list[i]=p;
        }
        guard->option.current_task = _min_dist(guard);
        guard->option.move_function = &classic_move_fonction;
    }
}

void mousaid(guard* guard){
    guard->option.angle_list = calloc(1,sizeof(int));
    guard->option.position_list = malloc(1*sizeof(position));
    guard->option.list_size = 1;
    guard->option.position_list[0] = *guard->pos;
    guard->option.current_task = 0;
    guard->option.move_function = &mousaid_move_fonction;
}

void snake(guard* guard){
    guard->option.angle_list = calloc(5,sizeof(int));
    guard->option.position_list = malloc(5*sizeof(position));
    guard->option.list_size = 5;
    for(int i=0; i<5; i++){
        guard->option.position_list[i] = *guard->pos;
    }
    guard->option.current_task = 0;
    guard->option.move_function = &snake_move_fonction;
}

void speed_rotation_fonction(guard_list* guards, scene_object* scene,
int i, int leader){
    guard* guard = guards->tab[i];
    guard->angle = (guard->angle + guard->rotation_speed)%360;
}

void classic_move_fonction(guard_list* guards, scene_object* scene,
int i, int leader){
    guard* guard = guards->tab[i];
    behave_option option = guard->option;
    int current = option.current_task;
    position next_pos = option.position_list[current];
    if(option.angle_list[current] != 0){
        guard_rotate_by(guard,option.angle_list[current]);
    }
    else{
        guard_natural_move_to(guard,next_pos,scene);
    }
    if(option.angle_list[current] == 0 && equals(guard->pos,&next_pos)){
        guard->option.current_task = (current+1)%option.list_size;
    }
}

void random_move_fonction(guard_list* guards, scene_object* scene, int i, int
leader){
    guard* guard = guards->tab[i];
    behave_option option = guard->option;
    position next_pos = option.position_list[0];
    guard_natural_move_to(guard,next_pos,scene);
    if(equals(guard->pos,&next_pos)){
        guard->option.position_list[0].x = rand()%scene->width;
        guard->option.position_list[0].y = rand()%scene->height;
    }
}

void _bounce_position(position* pos, scene_object* scene){
    int b1 = pos->x/scene->width > 0;
    int b2 = pos->y/scene->height > 0;
    if(pos->x < 0 || b1){
        pos->x = -pos->x + 2*b1*scene->width;
    }
    if(pos->y < 0 || b2){
        pos->y = -pos->y + 2*b2*scene->height;
    }
    check_position(pos,scene);
}

void _micro_decision(guard* leader,scene_object* scene, guard* guard){
    int D = 8;
    int x_min=guard->pos->x-D;
    int x_max=guard->pos->x+D;
    int y_min=guard->pos->y-D;
    int y_max=guard->pos->y+D;
    position* p1 = guard->pos;
    position* p = &guard->option.position_list[0];
    for(int x=x_min;x<=x_max ;x++){
        for(int y=y_min;y<=y_max ;y++){
            position q = {x,y};
            check_position(&q,scene);
            if(scene->grid[q.y][q.x] != scene->init){
                    p->y += (p1->y-q.y>0?1:-1);
                    p->x += (p1->x-q.x>0?1:-1);
            }
        }
    }
}

void mousaid_move_fonction(guard_list* guards, scene_object* scene, int i,
int j){
    guard* leader = guards->tab[j];
    guard* guard = guards->tab[i];
    behave_option option = guard->option;
    position next_pos = option.position_list[0];
    int D = 3;
    if(j == i){
        guard_natural_move_to(guard,next_pos,scene);
        if(equals(guard->pos,&next_pos)){
            position new_pos = *guard->pos;
            int angle = guard->angle*(3.14/180);
            new_pos.x += (int)(cos(angle)*D);
            new_pos.y -= (int)(sin(angle)*D);
            _bounce_position(&new_pos,scene);
            option.position_list[0] = new_pos;
        }
    }
    else{
        guard_natural_move_to(guard,next_pos,scene);
            position* p1 = guard->pos;
            position* p2 = leader->pos;
            position p = leader->option.position_list[0];
            if(range(p1,p2)<VISION_DISTANCE){
                if(((leader->angle-45)/90)%2){
                    p.y += (p1->y-p2->y>0?1:-1)*VISION_DISTANCE;
                }
                else{
                    p.x += (p1->x-p2->x>0?1:-1)*VISION_DISTANCE;
                }
            }
            guard->option.position_list[0]=p;
            _micro_decision(leader,scene,guard);
            _bounce_position(&option.position_list[0],scene);

    }
}

void snake_move_fonction(guard_list* guards, scene_object* scene, int
i, int leader){
    guard* guard = guards->tab[i];
    behave_option option = guard->option;
    int current = option.current_task;
    position next_pos = option.position_list[current];
    guard_natural_move_to(guard,next_pos,scene);
    if(option.angle_list[current] == 0 && equals(guard-
>pos,&next_pos)){
        guard->option.current_task = (current+1)%option.list_size;
        if(guard->option.current_task == 0){
            int x1 = (int)(i*(double)scene->width/guards->nb);
            int x2 = (int)((i+1)*(double)scene->width/guards->nb);
            if(x2>=scene->width){
                x2 = scene->width-1;
            }
            int dy = sin(VISION_FIELD*3.14/360)*VISION_DISTANCE;
            guard->option.current_task++;
            int x[2] = {x2,x1};
            int h = scene->height;
            for(int j=1;j<5;j++){
                guard->option.position_list[j].y = (guard->pos->y-
(j/2)*dy+h)%h;
                guard->option.position_list[j].x = x[(j-1)/2];
            }
        }
    }
}
```

```
}

<From entity/player.c>


#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include "player.h"
#include "../tools/graph.h"

player* new_player(int x, int y){
    player* bot = malloc(sizeof(player));
    position* pos = new_position(x, y);
    bot->pos = pos;
    return bot;
}

void delete_player(player* bot){
    delete_position(bot->pos);
    free(bot);
}

void move_left(position* pos,scene_object* scene){
    if(pos->x > 0){
        pos->x--;
    }
}

void move_right(position* pos,scene_object* scene){
    if(pos->x < scene->width-1){
        pos->x++;
    }
}

void move_down(position* pos,scene_object* scene){
    if(pos->y > 0){
        pos->y--;
    }
}

void move_up(position* pos,scene_object* scene){
    if(pos->y < scene->height-1){
        pos->y++;
    }
}

void naive_path(player* bot, scene_object** future) {
    scene_object* today = (future[0]->time > future[1]->time) ? future[1] : future[0];
    scene_object* tomorrow = (future[0]->time > future[1]->time) ? future[0] : future[1];

    if (bot->pos->x == 0) {
        if (bot->pos->y == 0) {
            if (check_and_move(bot, today, tomorrow, bot->pos->y + 1, bot->pos->x, move_up) ||
                check_and_move(bot, today, tomorrow, bot->pos->y, bot->pos->x + 1, move_right)) {
                return;
            }
        } else {
            if (check_and_move(bot, today, tomorrow, bot->pos->y + 1, bot->pos->x, move_up) ||
                check_and_move(bot, today, tomorrow, bot->pos->y, bot->pos->x + 1, move_right) ||
                check_and_move(bot, today, tomorrow, bot->pos->y - 1, bot->pos->x, move_down)) {
                return;
            }
        }
    } else if (bot->pos->x == today->width - 1) {
        if (bot->pos->y == 0) {
            if (check_and_move(bot, today, tomorrow, bot->pos->y + 1, bot->pos->x, move_up) ||
                check_and_move(bot, today, tomorrow, bot->pos->y, bot->pos->x - 1, move_left)) {
                return;
            }
        } else {
            if (check_and_move(bot, today, tomorrow, bot->pos->y + 1, bot->pos->x, move_up) ||
                check_and_move(bot, today, tomorrow, bot->pos->y, bot->pos->x - 1, move_left) ||
                check_and_move(bot, today, tomorrow, bot->pos->y - 1, bot->pos->x, move_down)) {
                return;
            }
        }
    } else {
        if (bot->pos->y == 0) {
            if (check_and_move(bot, today, tomorrow, bot->pos->y + 1, bot->pos->x, move_up) ||
                check_and_move(bot, today, tomorrow, bot->pos->y, bot->pos->x + 1, move_right) ||
                check_and_move(bot, today, tomorrow, bot->pos->y, bot->pos->x - 1, move_left)) {
                return;
            }
        } else {
            if (check_and_move(bot, today, tomorrow, bot->pos->y + 1, bot->pos->x, move_up) ||
                check_and_move(bot, today, tomorrow, bot->pos->y, bot->pos->x + 1, move_right) ||
                check_and_move(bot, today, tomorrow, bot->pos->y, bot->pos->x - 1, move_left) ||
                check_and_move(bot, today, tomorrow, bot->pos->y - 1, bot->pos->x, move_down)) {
                return;
            }
        }
    }
}

int check_and_move(player* bot, scene_object* today, scene_object* tomorrow,
                   int y, int x, void (*move_function)(position*, scene_object*)){
    if ((today->grid[y][x] == today->init) &&
        (tomorrow->grid[y][x] == tomorrow->init)) {
        move_function(bot->pos, today);
        return 1;
    }
    return 0;
}

void A_path(player* bot, scene_object** future, int time, int accuracy,stack* closedList,
            priority_list* openList){
    position* end = new_position(bot->pos->x,future[0]->height-1);
    position* next = A_star(future,time,accuracy,end,closedList,openList);
    if(next!=NULL){
        moveto(bot->pos,next->x,next->y);
    }
    delete_position(next);
    delete_position(end);
}

void path(player* bot, scene_object** future, int time,int accuracy,
          stack* closedList, priority_list* openList){
    if(accuracy == 2){
        naive_path(bot,future);
    }
    else{
        A_path(bot,future,time,accuracy,closedList,openList);
    }
}


<From entity/position.c>


#include <stdio.h>
#include <stdlib.h>
#include <math.h>
#include "position.h"
#include "../scene/scene.h"

position* new_position(int x, int y){
    position* pos = malloc(sizeof(position));
    pos->x = x;
    pos->y = y;
    return pos;
}

position* copy_position(position* pos){
    return new_position(pos->x, pos->y);
}

void delete_position(position* pos){
    free(pos);
}

list_position* new_list_position(int nb){
```

```
    list_position* list = malloc(sizeof(list_position));
    position** tab = calloc(nb, sizeof(position*));
    list->size = nb;
    list->tab = tab;
    return list;
}

void delete_list_position(list_position* list_pos){
    for(int i=0; i<list_pos->size; i++){
        delete_position(list_pos->tab[i]);
    }
    free(list_pos->tab);
    free(list_pos);
}

void moveto(position* pos, int x, int y){
    pos->x = x;
    pos->y = y;
}

bool equals(position* pos1,position* pos2){
    if(pos1->y == pos2->y && pos1->x == pos2->x){
        return true;
    }
    else{
        return false;
    }
}

float range(position* pos1,position* pos2){
    int dx = (pos1->x - pos2->x);
    int dy = (pos1->y - pos2->y);
    return sqrt(dx*dx+dy*dy);
}

queue_elt* new_queue_elt(position pos){
    queue_elt* res = malloc(sizeof(queue_elt));
    res->next = NULL;
    res->prev = NULL;
    res->pos = pos;
    return res;
}

void delete_queue_elt(queue_elt* elt){
    free(elt);
}

queue_position* new_queue_position(){
    queue_position* res = malloc(sizeof(queue_position));
    res->end = NULL;
    res->start = NULL;
    return res;
}

void delete_queue_position(queue_position* q){
    while (q->end != NULL){
        delete_queue_elt(dequeue(q));
    }
    free(q);
}

void enqueue(queue_position* q, queue_elt* elt){
    if(q->start != NULL){
        elt->next = q->start;
        q->start->prev = elt;
        q->start = elt;
    }
    else{
        q->end = elt;
        q->start = elt;
    }
}

queue_elt* dequeue(queue_position* q){
    queue_elt* res;
    if(q->end == q->start){
        res = q->start;
        q->start = NULL;
        q->end = NULL;
    }
    else{
        res = q->end;
        res->prev->next = NULL;
        q->end = res->prev;
    }
    res->prev = NULL;
    res->next = NULL;
    return res;
}

queue_position* copy_queue(queue_position* q){
    queue_position* res = new_queue_position();
    if(q != NULL && q->end != NULL){
        queue_elt* elt = q->end;
        while (elt != q->start){
            enqueue(res,new_queue_elt(elt->pos));
            elt = elt->prev;
        }
        enqueue(res,new_queue_elt(elt->pos));
    }
    return res;
}

tree_node* new_tree_node(position pos, tree_node* start, tree_node* prev){
    tree_node* res = malloc(sizeof(tree_node));
    res->nexts = calloc(5,sizeof(tree_node*));
    res->nb_nexts = 0;
    res->nb_deleted = 0;
    res->prev = prev;
    res->start = start;
    res->pos = pos;
    return res;
}

void add_next_tree_node(tree_node* node, tree_node* next){
    if(node->nb_nexts < 5){
        node->nexts[node->nb_nexts] = next;
        next->name = node->nb_nexts;
        node->nb_nexts ++;
    }
}

void swap_t_n(tree_node* node,int i, int j){
    node->nexts[i]->name = j;
    node->nexts[j]->name = i;
    tree_node* tmp = node->nexts[i];
    node->nexts[i] = node->nexts[j];
    node->nexts[j] = tmp;
}

void delete_tree_node(tree_node* node){
    for(int i=0; i<node->nb_nexts; i++){
        node->nexts[i]->prev = NULL;
        delete_tree_node_rac(node->nexts[i]);
    }
    if(node->prev != NULL){
        tree_node* prev = node->prev;
        swap_t_n(prev,node->name,prev->nb_nexts-1);
        prev->nb_deleted ++;
        prev->nb_nexts --;
        if(prev->nb_deleted >= 5){
            delete_tree_node(prev);
        }
        free(node->nexts);
        free(node);
    }
}

void delete_tree_node_rac(tree_node* rac){
    delete_tree_node(rac);
    free(rac->nexts);
    free(rac);
}

tree_node* new_start_tree_node(tree_node* node){
    tree_node* current = node;
    tree_node* prev = current->prev;
    if(prev == NULL || prev->prev == NULL){
        return NULL;
    }
    while(prev != NULL && prev->prev != NULL){
        current = prev;
        prev = current->prev;
    }
    return current;
}


--------------------------------------------------------------------------------
----


<From old_function/clean_holl.c>


#include "../entity/position.h"
#include "../scene/scene.h"

void clean_holl1(scene_object* scene,int color){
    for(int i=0;i<scene->height;i++){
        for(int j=0;j<scene->width;j++){
            if(scene->grid[i][j]!=color){
                if(i==0){
                    if(j==0){
```

```
                    if(scene->grid[i][j+1]==color && scene->grid[i+1][j]==color)
                        scene->grid[i][j] = color;
                }
                else if(j==scene->width-1){
                    if(scene->grid[i][j-1]==color && scene->grid[i+1][j]==color)
                        scene->grid[i][j] = color;
                }
                else{
                    if(scene->grid[i][j-1]==color && scene->grid[i][j+1]==color
                      && scene->grid[i+1][j]==color) scene->grid[i][j] = color;
                }
            }
            else if(i==scene->height-1){
                if(j==0){
                    if(scene->grid[i-1][j]==color && scene->grid[i][j+1]==color)
                        scene->grid[i][j] = color;
                }
                else if(j==scene->width-1){
                    if(scene->grid[i-1][j]==color && scene->grid[i][j-1]==color)
                        scene->grid[i][j] = color;
                }
                else{
                    if(scene->grid[i-1][j]==color && scene->grid[i][j-1]==color
                      && scene->grid[i][j+1]==color) scene->grid[i][j] = color;
                }
            }
            else{
                if(j==0){
                    if(scene->grid[i-1][j]==color && scene->grid[i][j+1]==color
                      && scene->grid[i+1][j]==color) scene->grid[i][j] = color;
                }
                else if(j==scene->width-1){
                    if(scene->grid[i-1][j]==color && scene->grid[i][j-1]==color
                      && scene->grid[i+1][j]==color) scene->grid[i][j] = color;
                }
                else{
                    if(scene->grid[i-1][j]==color && scene->grid[i][j-1]==color
                      && scene->grid[i][j+1]==color && scene->grid[i+1][j]==color)
                        scene->grid[i][j] = color;
                }
            }
        }
    }
}
```

```
----------------------------------------------------------------------------

<From old_function/draw_line.c>


#include "../entity/position.h"
#include "../scene/scene.h"
#include "../tools/draw.h"

void draw_line(position* p,position* p2,scene_object* scene,int color){

    position* p1 = copy_position(p);

    int dx = p2->x - p1->x;
    int dy = p2->y - p1->y;

    if (dx!=0){//Non vertical

        if(dx>0){//Partie droite

            if(dy!=0){//Non horizontal

                if(dy>0){//Partie haute, donc de 0 à pi/2

                    if(dx >= dy){//Donc 0 à pi/4, octant numéro 1

                        int e = dx;
                        dx = e*2;
                        dy = dy*2;

                        while(p1->x != p2->x){
                            draw_position(p1,scene, color);
                            e = e - dy;
                            if(e<0){
                                p1->y++;
                                e = e + dx;
                            }
                            p1->x++;
                        }

                    }
                    else {//donc de pi/4 à pi/2, octant numéro 2 (symétrique par rapport à x=y)

                        int e = dy;
                        dx = dx*2;
                        dy = e*2;

                        while(p1->y != p2->y + 1){
                            draw_position(p1,scene, color);
                            e = e - dx;
                            if(e<0){
                                p1->x++;
                                e = e + dy;
                            }
                            p1->y++;
                        }

                    }
                }
                else{//Donc Partie basse (toujours droite)

                    if(dx>=-dy){//Octant numéro 8

                        int e = dx;
                        dx = e*2;
                        dy = dy*2;

                        while(p1->x != p2->x + 1){
                            draw_position(p1,scene, color);
                            e = e + dy;
                            if(e<0){
                                p1->y--;
                                e = e + dx;
                            }
                            p1->x++;
                        }

                    }
                    else{//Octant numéro 7

                        int e = dy;
                        dx = dx*2;
                        dy = e*2;

                        while(p1->y != p2->y - 1){
                            draw_position(p1,scene, color);
                            e = e + dx;
                            if(e>0){
                                p1->x++;
                                e = e + dy;
                            }
                            p1->y--;
                        }

                    }
                }
            }
            else{//Horizontal à droite

                while(p1->x != p2->x + 1){
                    draw_position(p1,scene, color);
                    p1->x++;
                }

            }
        }
        else{//Partie Guauche

            if(dy!=0){//Non Horizontal

                if(dy>0){//Partie Haute-Guache

                    if(-dx >= dy){//Octant numéro 4

                        int e = dx;
                        dx = e*2;
```

```
                dy = dy*2;

                while(p1->x != p2->x - 1){
                    draw_position(p1,scene, color);
                    e = e + dy;
                    if(e>=0){
                        p1->y++;
                        e = e + dx;
                    }
                    p1->x--;
                }

            }
            else{//Octane numéro 3

                int e = dy;
                dx = dx*2;
                dy = e*2;

                while(p1->y != p2->y + 1){
                    draw_position(p1,scene, color);
                    e = e + dx;
                    if(e<=0){
                        p1->x--;
                        e = e + dy;
                    }
                    p1->y++;
                }

            }

        }
        else{

            if(dx<=dy){//Octant numéro 5

                int e = dx;
                dx = e*2;
                dy = dy*2;

                while(p1->x != p2->x - 1){
                    draw_position(p1,scene, color);
                    e = e - dy;
                    if(e>=0){
                        p1->y--;
                        e = e + dx;
                    }
                    p1->x--;
                }

            }
            else{//Octant numéro 6

                int e = dy;
                dx = dx*2;
                dy = e*2;

                while(p1->y != p2->y - 1){
                    draw_position(p1,scene, color);
                    e = e - dx;
                    if(e>=0){
                        p1->x--;
                        e = e + dy;
                    }
                    p1->y--;
                }

            }

        }
    }
    else{//Horizontale Gauche

        while (p1->x != p2->x - 1){
            draw_position(p1,scene,color);
            p1->x--;
        }

    }

}
else{//Vertical

    if(dy!=0){

        if(dy>0){

            while (p1->y != p2->y + 1){
                draw_position(p1,scene,color);
                p1->y++;
            }

        }
        else{

            while (p1->y != p2->y - 1){
                draw_position(p1,scene,color);
                p1->y--;
            }

        }
    }
    else{//p1 = p2

        draw_position(p1,scene,color);

    }

}

delete_position(p1);

}
```

<From old_function/k_simulation.c>

```


void k_simulation(int k,int number,int behave_start,int behave_end,
int accuracy){
    guard_list** storage = malloc(k*sizeof(guard_list*));
    for(int i=0; i<k; i++){
        storage[i] = new_guard_list(number,0,ROTATION_SPEED,1,4);
    }

    for(int behave=behave_start; behave<=behave_end; behave++){
        guard_list** lab = copy_lab(storage,k,behave);
        char* file = new_name(behave,number);
        FILE* score = fopen(file,"w");

        for(int i=0; i<k; i++){
            scene_object* scene = new_scene(WIDTH,HEIGHT,INIT);
            player* bot = new_player(WIDTH/2,0);
            if(behave == 0){
                fprintf(score,"%d\
n",simulate(scene,lab[i],bot,3,accuracy,false,NULL));
            }
            else{
                fprintf(score,"%d\
n",simulate(scene,lab[i],bot,10,accuracy,false,NULL));
            }
            //multithreads ici

            delete_scene(scene);
            delete_player(bot);
        }
        delete_lab(lab,k);
        fclose(score);
        free(file);
    }
    delete_lab(storage,k);
}
```

<From old_function/multicore_path.c>

```


void multicore_path(player* bot, scene_object** future, int time, int
accuracy,
                    stack* closedList, priority_list* openList, int
nb_threads){
    pthread_t* core = malloc(nb_threads*sizeof(pthread_t));
    pthread_mutex_t m_prio = PTHREAD_MUTEX_INITIALIZER;
    pthread_mutex_t m_stack = PTHREAD_MUTEX_INITIALIZER;

    position* end = new_position(bot->pos->x,future[0]->height-1);
    position* next = NULL;
    A_arg arg =
{future,time,accuracy,end,closedList,openList,&m_prio,&m_stack};

    for(int i = 0; i < nb_threads; i++){
```

```
        pthread_create(&core[i],NULL,multicore_A_star,&arg);
    }
    for(int i = 0; i < nb_threads; i++){
        if(next != NULL){
            free(next);
        }
        pthread_join(core[i],(void*)&next);
    }

    if(next!=NULL){
        moveto(bot->pos,next->x,next->y);
    }
    free(core);
    delete_position(next);
    delete_position(end);
}


--------------------------------------------------------------------------------


<From old_function/naive_path.c>


#include "../entity/player.h"
#include "../scene/scene.h"

void naive_path1(player* bot, scene_object** future){
    scene_object* today;
    scene_object* tomorrow;
    if(future[0]->time > future[1]->time){
        today = future[1];
        tomorrow = future[0];
    }
    else{
        today = future[0];
        tomorrow = future[1];
    }

    if(bot->pos->x == 0){//disjonction sur x
        if(bot->pos->y == 0){//disjonction sur y
            if((today->grid[bot->pos->y+1][bot->pos->x] == today->init) &&
                    (tomorrow->grid[bot->pos->y+1][bot->pos->x] == tomorrow->init)){
                move_up(bot->pos,today);
            }
            // else if((tomorrow->grid[bot->pos->y][bot->pos->x] == tomorrow->init)){
            //     moveto(bot->pos,bot->pos->x,bot->pos->y);//dont move
            // }
            else if((today->grid[bot->pos->y][bot->pos->x+1] == today->init) &&
                            (tomorrow->grid[bot->pos->y][bot->pos->x+1] == tomorrow->init)){
                move_right(bot->pos,today);
            }
            else{
                move_down(bot->pos,today);
            }
        }
        else if(bot->pos->y == today->width-1){
            if((today->grid[bot->pos->y+1][bot->pos->x] == today->init) &&
                    (tomorrow->grid[bot->pos->y+1][bot->pos->x] == tomorrow->init)){
                move_up(bot->pos,today);
            }
            // else if((tomorrow->grid[bot->pos->y][bot->pos->x] == tomorrow->init)){
            //     moveto(bot->pos,bot->pos->x,bot->pos->y);//dont move
            // }
            else if((today->grid[bot->pos->y][bot->pos->x-1] == today->init) &&
                    (tomorrow->grid[bot->pos->y][bot->pos->x-1] == tomorrow->init)){
                move_left(bot->pos,today);
            }
        }
        else{
            if((today->grid[bot->pos->y+1][bot->pos->x] == today->init) &&
                    (tomorrow->grid[bot->pos->y+1][bot->pos->x] == tomorrow->init)){
                move_up(bot->pos,today);
            }
            // else if((tomorrow->grid[bot->pos->y][bot->pos->x] == tomorrow->init)){
            //     moveto(bot->pos,bot->pos->x,bot->pos->y);//dont move
            // }
            else if((today->grid[bot->pos->y][bot->pos->x-1] == today->init) &&
                    (tomorrow->grid[bot->pos->y][bot->pos->x-1] == tomorrow->init)){
                move_left(bot->pos,today);
            }
            else{
                move_down(bot->pos,today);
            }
        }
    }
    else if(bot->pos->x == today->width-1){
        if(bot->pos->y == 0){
            if((today->grid[bot->pos->y+1][bot->pos->x] == today->init) &&
                    (tomorrow->grid[bot->pos->y+1][bot->pos->x] == tomorrow->init)){
                move_up(bot->pos,today);
            }
            // else if((tomorrow->grid[bot->pos->y][bot->pos->x] == tomorrow->init)){
            //     moveto(bot->pos,bot->pos->x,bot->pos->y);//dont move
            // }
            else if((today->grid[bot->pos->y][bot->pos->x-1] == today->init) &&
                    (tomorrow->grid[bot->pos->y][bot->pos->x-1] == tomorrow->init)){
                move_left(bot->pos,today);
            }
            else{
                move_down(bot->pos,today);
            }
        }
        else if(bot->pos->y == today->width-1){
            if((today->grid[bot->pos->y][bot->pos->x-1] == today->init) &&
                    (tomorrow->grid[bot->pos->y][bot->pos->x-1] == tomorrow->init)){
                move_left(bot->pos,today);
            }
            else{
                move_down(bot->pos,today);
            }
        }
        else{
            if((today->grid[bot->pos->y+1][bot->pos->x] == today->init) &&
                    (tomorrow->grid[bot->pos->y+1][bot->pos->x] == tomorrow->init)){
                move_up(bot->pos,today);
            }
            // else if((tomorrow->grid[bot->pos->y][bot->pos->x] == tomorrow->init)){
            //     moveto(bot->pos,bot->pos->x,bot->pos->y);//dont move
            // }
            else if((today->grid[bot->pos->y][bot->pos->x-1] == today->init) &&
                    (tomorrow->grid[bot->pos->y][bot->pos->x-1] == tomorrow->init)){
                move_left(bot->pos,today);
            }
            else{
                move_down(bot->pos,today);
            }
        }
    }
    else{
        if(bot->pos->y == 0){
            if((today->grid[bot->pos->y+1][bot->pos->x] == today->init) &&
                    (tomorrow->grid[bot->pos->y+1][bot->pos->x] == tomorrow->init)){
                move_up(bot->pos,today);
            }
            // else if((tomorrow->grid[bot->pos->y][bot->pos->x] == tomorrow->init)){
            //     moveto(bot->pos,bot->pos->x,bot->pos->y);//dont move
            // }
            else if((today->grid[bot->pos->y][bot->pos->x+1] == today->init) &&
                            (tomorrow->grid[bot->pos->y][bot->pos->x+1] == tomorrow->init)){
                move_right(bot->pos,today);
            }
            else if((today->grid[bot->pos->y][bot->pos->x-1] == today->init) &&
                    (tomorrow->grid[bot->pos->y][bot->pos->x-1] == tomorrow->init)){
                move_left(bot->pos,today);
            }
            else{
                move_right(bot->pos,today);
            }
        }
        else if(bot->pos->y == today->width-1){
            if((today->grid[bot->pos->y+1][bot->pos->x] == today->init) &&
                    (tomorrow->grid[bot->pos->y+1][bot->pos->x] == tomorrow->init)){
                move_up(bot->pos,today);
            }
            // else if((tomorrow->grid[bot->pos->y][bot->pos->x] == tomorrow->init)){
            //     moveto(bot->pos,bot->pos->x,bot->pos->y);//dont move
            // }
            else if((today->grid[bot->pos->y][bot->pos->x+1] == today->init) &&
                            (tomorrow->grid[bot->pos->y][bot->pos->x+1] == tomorrow->init)){
                move_right(bot->pos,today);
            }
            else if((today->grid[bot->pos->y][bot->pos->x-1] == today->init) &&
                    (tomorrow->grid[bot->pos->y][bot->pos->x-1] == tomorrow->init)){
                move_left(bot->pos,today);
            }
            else{
                move_down(bot->pos,today);
            }
        }
        else{// Pas proche des bords
            if((today->grid[bot->pos->y+1][bot->pos->x] == today->init) &&
                    (tomorrow->grid[bot->pos->y+1][bot->pos->x] == tomorrow->init)){
                move_up(bot->pos,today);

            }
            // else if((tomorrow->grid[bot->pos->y][bot->pos->x] == tomorrow->init)){
            //     moveto(bot->pos,bot->pos->x,bot->pos->y);//dont move
            // }
            else if((today->grid[bot->pos->y][bot->pos->x+1] == today->init) &&
                            (tomorrow->grid[bot->pos->y][bot->pos->x+1] == tomorrow->init) &&
                            (today->grid[bot->pos->y][bot->pos->x-1] == today->init) &&
                            (tomorrow->grid[bot->pos->y][bot->pos->x-1] == tomorrow->init)){
                int left_range = abs(today->width/2 - bot->pos->x-1);
                int right_range = abs(today->width/2 - bot->pos->x+1);
                if(left_range < right_range){
                    move_left(bot->pos,today);
                }
                else{
                    move_right(bot->pos,today);
                }
            }
            else if((today->grid[bot->pos->y][bot->pos->x+1] == today->init) &&
                            (tomorrow->grid[bot->pos->y][bot->pos->x+1] == tomorrow->init)){
                move_right(bot->pos,today);
            }
            else if((today->grid[bot->pos->y][bot->pos->x-1] == today->init) &&
                            (tomorrow->grid[bot->pos->y][bot->pos->x-1] == tomorrow->init)){
                move_left(bot->pos,today);
            }
            else{
                if((today->grid[bot->pos->y+1][bot->pos->x] == today->init) &&
                        (tomorrow->grid[bot->pos->y+1][bot->pos->x] == tomorrow->init)){
                    move_up(bot->pos,today);
                }
                // else if((tomorrow->grid[bot->pos->y][bot->pos->x] == tomorrow->init)){
                //     moveto(bot->pos,bot->pos->x,bot->pos->y);//dont move
                // }
                else if((today->grid[bot->pos->y][bot->pos->x+1] == today->init) &&
                                (tomorrow->grid[bot->pos->y][bot->pos->x+1] == tomorrow->init) &&
                                (today->grid[bot->pos->y][bot->pos->x-1] == today->init) &&
                                (tomorrow->grid[bot->pos->y][bot->pos->x-1] == tomorrow->init)){
                    int left_range = abs(today->width/2 - bot->pos->x-1);
                    int right_range = abs(today->width/2 - bot->pos->x+1);
                    if(left_range < right_range){
                        move_left(bot->pos,today);
                    }
                    else{
                        move_right(bot->pos,today);
                    }
                }
                else if((today->grid[bot->pos->y][bot->pos->x+1] == today->init) &&
                                (tomorrow->grid[bot->pos->y][bot->pos->x+1] == tomorrow->init) &&
                                (today->grid[bot->pos->y][bot->pos->x-1] == today->init) &&
                                (tomorrow->grid[bot->pos->y][bot->pos->x-1] == tomorrow->init)){
                    move_right(bot->pos,today);
                }
                else if((today->grid[bot->pos->y][bot->pos->x-1] == today->init) &&
                                (tomorrow->grid[bot->pos->y][bot->pos->x-1] == tomorrow->init)){
                    move_left(bot->pos,today);
                }
                else{
                    // moveto(bot->pos,bot->pos->x,bot->pos->y);//dont move
                }
            }
        }
    }
}
```

```
                        (tomorrow->grid[bot->pos->y][bot->pos->x-
1] == tomorrow->init)){

                int left_range = abs(today->width/2 - bot->pos->x-1);
                int right_range = abs(today->width/2 - bot->pos-
>x+1);
                if(left_range < right_range){
                    move_left(bot->pos,today);
                }
                else{
                    move_right(bot->pos,today);
                }
            }
            else if((today->grid[bot->pos->y][bot->pos->x+1] ==
today->init) &&
                        (tomorrow->grid[bot->pos->y][bot->pos-
>x+1] == tomorrow->init)){
                move_right(bot->pos,today);
            }
            else if((today->grid[bot->pos->y][bot->pos->x-1] ==
today->init) &&
                        (tomorrow->grid[bot->pos->y][bot->pos->x-
1] == tomorrow->init)){
                move_left(bot->pos,today);
            }
            else{
                move_down(bot->pos,today);
            }
        }
    }
}

-------------------------------------------------------------------------
-----



<From old_function/neighbors.c>



#include "../entity/player.h"

list_position* neighbors(scene_object* scene, position* pos, int
height, int width){
    list_position* list = new_list_position(5);
    int cpt=0;

    if(pos->x == 0){
        if(pos->y == 0){
            if(scene->grid[pos->y][pos->x] == scene->init){
                list->tab[cpt] = new_position(pos->x,pos->y);
                cpt++;
            }
            if(scene->grid[pos->y+1][pos->x] == scene->init){
                list->tab[cpt] = new_position(pos->x,pos->y+1);
                cpt++;
            }
            if(scene->grid[pos->y][pos->x+1] == scene->init){
                list->tab[cpt] = new_position(pos->x+1,pos->y);
                cpt++;
            }
        }
        else if(pos->y == height-1){
            if(scene->grid[pos->y][pos->x] == scene->init){
                list->tab[cpt] = new_position(pos->x,pos->y);
                cpt++;
            }
            if(scene->grid[pos->y-1][pos->x] == scene->init){
                list->tab[cpt] = new_position(pos->x,pos->y-1);
                cpt++;
            }
            if(scene->grid[pos->y][pos->x+1] == scene->init){
                list->tab[cpt] = new_position(pos->x+1,pos->y);
                cpt++;
            }
        }
        else{
            if(scene->grid[pos->y][pos->x] == scene->init){
                list->tab[cpt] = new_position(pos->x,pos->y);
                cpt++;
            }
            if(scene->grid[pos->y+1][pos->x] == scene->init){
                list->tab[cpt] = new_position(pos->x,pos->y+1);
                cpt++;
            }
            if(scene->grid[pos->y-1][pos->x] == scene->init){
                list->tab[cpt] = new_position(pos->x,pos->y-1);
                cpt++;
            }
            if(scene->grid[pos->y][pos->x+1] == scene->init){
                list->tab[cpt] = new_position(pos->x+1,pos->y);
                cpt++;
            }
        }
    }
    else if(pos->x == width-1){
        if(pos->y == 0){
            if(scene->grid[pos->y][pos->x] == scene->init){
                list->tab[cpt] = new_position(pos->x,pos->y);
                cpt++;
            }
            if(scene->grid[pos->y+1][pos->x] == scene->init){
                list->tab[cpt] = new_position(pos->x,pos->y+1);
                cpt++;
            }
            if(scene->grid[pos->y][pos->x-1] == scene->init){
                list->tab[cpt] = new_position(pos->x-1,pos->y);
                cpt++;
            }
        }
        else if(pos->y == height-1){
            if(scene->grid[pos->y][pos->x] == scene->init){
                list->tab[cpt] = new_position(pos->x,pos->y);
                cpt++;
            }
            if(scene->grid[pos->y+1][pos->x] == scene->init){
                list->tab[cpt] = new_position(pos->x,pos->y+1);
                cpt++;
            }
            if(scene->grid[pos->y][pos->x-1] == scene->init){
                list->tab[cpt] = new_position(pos->x-1,pos->y);
                cpt++;
            }
        }
        else{
            if(scene->grid[pos->y][pos->x] == scene->init){
                list->tab[cpt] = new_position(pos->x,pos->y);
                cpt++;
            }
            if(scene->grid[pos->y+1][pos->x] == scene->init){
                list->tab[cpt] = new_position(pos->x,pos->y+1);
                cpt++;
            }
            if(scene->grid[pos->y-1][pos->x] == scene->init){
                list->tab[cpt] = new_position(pos->x,pos->y-1);
                cpt++;
            }
            if(scene->grid[pos->y][pos->x-1] == scene->init){
                list->tab[cpt] = new_position(pos->x-1,pos->y);
                cpt++;
            }
        }
    }
    else{
        if(pos->y == 0){

            if(scene->grid[pos->y][pos->x] == scene->init){
                list->tab[cpt] = new_position(pos->x,pos->y);
                cpt++;
            }
            if(scene->grid[pos->y+1][pos->x] == scene->init){
                list->tab[cpt] = new_position(pos->x,pos->y+1);
                cpt++;
            }
            if(scene->grid[pos->y][pos->x+1] == scene->init){
                list->tab[cpt] = new_position(pos->x+1,pos->y);
                cpt++;
            }
            if(scene->grid[pos->y][pos->x-1] == scene->init){
                list->tab[cpt] = new_position(pos->x-1,pos->y);
                cpt++;
            }
        }
        else if(pos->y == height-1){
            if(scene->grid[pos->y][pos->x] == scene->init){
                list->tab[cpt] = new_position(pos->x,pos->y);
                cpt++;
            }
            if(scene->grid[pos->y-1][pos->x] == scene->init){
                list->tab[cpt] = new_position(pos->x,pos->y-1);
                cpt++;
            }
            if(scene->grid[pos->y][pos->x+1] == scene->init){
                list->tab[cpt] = new_position(pos->x+1,pos->y);
                cpt++;
            }
            if(scene->grid[pos->y][pos->x-1] == scene->init){
                list->tab[cpt] = new_position(pos->x-1,pos->y);
                cpt++;
            }
        }
        else{
            if(scene->grid[pos->y][pos->x] == scene->init){
                list->tab[cpt] = new_position(pos->x,pos->y);
                cpt++;
            }
            if(scene->grid[pos->y-1][pos->x] == scene->init){
                list->tab[cpt] = new_position(pos->x,pos->y-1);
                cpt++;
            }
            if(scene->grid[pos->y+1][pos->x] == scene->init){
                list->tab[cpt] = new_position(pos->x,pos->y+1);
                cpt++;
            }
            if(scene->grid[pos->y][pos->x+1] == scene->init){
                list->tab[cpt] = new_position(pos->x+1,pos->y);
                cpt++;
            }
            if(scene->grid[pos->y][pos->x-1] == scene->init){
                list->tab[cpt] = new_position(pos->x-1,pos->y);
                cpt++;
            }
        }
    }
    else{
        if(pos->y == 0){

            if(scene->grid[pos->y][pos->x] == scene->init){
                list->tab[cpt] = new_position(pos->x,pos->y);
                cpt++;
            }
            if(scene->grid[pos->y+1][pos->x] == scene->init){
                list->tab[cpt] = new_position(pos->x,pos->y+1);
                cpt++;
            }
            if(scene->grid[pos->y][pos->x+1] == scene->init){
                list->tab[cpt] = new_position(pos->x+1,pos->y);
                cpt++;
            }
            if(scene->grid[pos->y][pos->x-1] == scene->init){
                list->tab[cpt] = new_position(pos->x-1,pos->y);
                cpt++;
            }
        }
        else if(pos->y == height-1){
            if(scene->grid[pos->y][pos->x] == scene->init){
                list->tab[cpt] = new_position(pos->x,pos->y);
                cpt++;
            }
            if(scene->grid[pos->y-1][pos->x] == scene->init){
                list->tab[cpt] = new_position(pos->x,pos->y-1);
                cpt++;
            }
            if(scene->grid[pos->y][pos->x+1] == scene->init){
                list->tab[cpt] = new_position(pos->x+1,pos->y);
                cpt++;
            }
            if(scene->grid[pos->y][pos->x-1] == scene->init){
                list->tab[cpt] = new_position(pos->x-1,pos->y);
                cpt++;
            }
        }
        else{
            if(scene->grid[pos->y][pos->x] == scene->init){
                list->tab[cpt] = new_position(pos->x,pos->y);
```

```
                cpt++;
            }
            if(scene->grid[pos->y+1][pos->x] == scene->init){
                list->tab[cpt] = new_position(pos->x,pos->y+1);
                cpt++;
            }
            if(scene->grid[pos->y-1][pos->x] == scene->init){
                list->tab[cpt] = new_position(pos->x,pos->y-1);
                cpt++;
            }
            if(scene->grid[pos->y][pos->x+1] == scene->init){
                list->tab[cpt] = new_position(pos->x+1,pos->y);
                cpt++;
            }
            if(scene->grid[pos->y][pos->x-1] == scene->init){
                list->tab[cpt] = new_position(pos->x-1,pos->y);
                cpt++;
            }
        }
    }
    list->size = cpt;
    return list;
}

-------------------------------------------------------------------------
-----


 <From old_function/threads.c>


#include <stdio.h>
#include "threads.h"
#include "graph.h"

void* multicore_A_star(void* args){
    scene_object** scene_list = ((A_arg*)args)->future;
    int time = ((A_arg*)args)->time;
    int accuracy = ((A_arg*)args)->accuracy;
    position* default_end = ((A_arg*)args)->end;
    stack* closedList = ((A_arg*)args)->closedList;
    priority_list* openList = ((A_arg*)args)->openList;
    pthread_mutex_t* m_prio = ((A_arg*)args)->m_prio;
    pthread_mutex_t* m_stack = ((A_arg*)args)->m_stack;

    position* res = NULL;

    int height = scene_list[0]->height;
    int width = scene_list[0]->width;
    position end = *default_end;

    while (true){
        pthread_mutex_lock(m_prio);
        data* u;
        bool is_empty = empty_priority_list(openList);
        if(!is_empty){
            u = openList->tas[1];
            if(u->way != NULL && u->way->start != NULL){
                end.x = u->way->start->pos.x;
            }
            else{
                end.x = default_end->x;
            }

            if((u->node->x == end.x && u->node->y == end.y) || u->cout >=
accuracy-1){
                position* res = copy_position(&(u->way->end->pos));
                pthread_mutex_unlock(m_prio);

                pthread_exit((void*)res);
        }

        u = remove_rac(openList);
        }
        pthread_mutex_unlock(m_prio);

        if(!is_empty){
            list_position* voisins
                  = neighbors(scene_list[(time+u->cout+1)%accuracy],u-
>node,height,width);
            for(int i = 0; i<voisins->size;i++){
                position* elt = voisins->tab[i];

                pthread_mutex_lock(m_prio);
                bool condition = !(is_in_stack(elt,u->cout+1,closedList) ||
                    is_in_priority(openList,elt,u->cout+1));
                pthread_mutex_unlock(m_prio);

                if(condition){
                    elt = copy_position(elt);
                    int d = range(&end,elt);
                    queue_position* new_way = copy_queue(u->way);
                    enqueue(new_way,new_queue_elt(*elt));
                    data* v = new_data(u->cout+1+d,elt,new_way,u->cout+1);

                    pthread_mutex_lock(m_prio);
                    insert(v,openList);
                    pthread_mutex_unlock(m_prio);
                }
            }

            delete_list_position(voisins);

            pthread_mutex_lock(m_stack);
            enstack(new_stack_elt(copy_position(u->node),u-
>cout),closedList);
            pthread_mutex_unlock(m_stack);

            delete_data(u);
        }
        else{
            break;
        }
    }

    pthread_exit((void*)res);
}

-------------------------------------------------------------------------
-----


 <From scene/scene.c>


#include <stdio.h>
#include <stdlib.h>
#include "scene.h"
#include "../tools/out.h"

scene_object* new_scene(int width, int height, int init){
    scene_object* scene = malloc(sizeof(scene_object));
    int** grid = malloc(height*sizeof(int*));
    for(int i=0;i<height;i++){
        int* row = calloc(width,sizeof(int));
        grid[i] = row;
    }
    scene->height = height;
    scene->width = width;
    scene->grid = grid;
    scene->time = 0;
    scene->init = init;
    return scene;
}

void scene_init(scene_object* scene,int value){
    for(int i=0;i<scene->height;i++){
        for(int j=0;j<scene->width;j++){
            scene->grid[i][j] = value;
        }
    }
}

void delete_scene(scene_object* scene){
    for(int i=0; i< scene->height;i++){
        free(scene->grid[i]);
    }
    free(scene->grid);
    free(scene);
}

void wait(scene_object* scene){
    scene->time++;
}

scene_object* copy_scene(scene_object* scene){
    scene_object* res = new_scene(scene->width,scene->height,scene-
>init);
    for(int i=0; i<res->height;i++){
        for(int j=0; j<res->width;j++){
            res->grid[i][j] = scene->grid[i][j];
        }
    }
    res->init=scene->init;
    return res;
}

-------------------------------------------------------------------------
----
```

```

<From scene/simulation.c>



#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include "../tools/draw.h"
#include "../tools/out.h"
#include "simulation.h"
#include "../tools/graph.h"

int simulate(scene_object* scene, guard_list* guard_list, player*
bot, int speed,
             int accuracy, bool export, FILE* output){

    int limit_time = scene->height*speed;
    int time = 0;

    scene_object** future = calloc(accuracy, sizeof(scene_object*));
    for(time=0; time<accuracy; time++){
        future[time] = copy_scene(scene);
        future[time]->time = time;
        behave(guard_list,future[time],time);
    }

    stack* closedList = new_stack();
    priority_list* openList = new_priority_list(1);
    position* start = copy_position(bot->pos);
    tree_node* rac = new_tree_node(*start,NULL,NULL);
    data* data_start = new_data(scene->height,start,rac,0);
    insert(data_start,openList);

    for(time = 0;time<limit_time;time++){
        draw_position(bot->pos,scene,2);
        // draw_position(guard_list->tab[0]->pos,scene,2);
        // printf("Avancement : %d %c ; time = %d/%d\n"
        //         ,100*bot->pos->y/scene-
>height,'%',time,limit_time);

        int curent_value = future[time%accuracy]->grid[bot->pos->y]
[bot->pos->x];
        if(curent_value == guard_list->color
            || curent_value == guard_list->color2){
            if(export){
                draw_position(bot->pos,future[time%accuracy],2);
                display_scene(future[time%accuracy],output);
            }
            time = -time;
            break;
        }
        if(bot->pos->y >= scene->height-1){
            break;
        }

        if(export){
            draw_position(bot->pos,future[time%accuracy],2);
            display_scene(future[time%accuracy],output);
        }

        behave(guard_list,future[(time+accuracy-1)%accuracy],time+accuracy-
1);

        path(bot,future,time,accuracy,closedList,openList);
        if(export){
            print_trajectory(future[(time+1)%accuracy],openList,5);
        }
        position end = {bot->pos->x,scene->height-1};
        adjust_priority_list(openList,bot->pos,end);
        adjust_stack(closedList,time);
    }

    for(int i=0; i<accuracy; i++){
        delete_scene(future[i]);
    }

    free(future);
    if(export){
        display_scene(scene,output);
    }
    delete_tree_node_rac(rac);
    delete_priority_list(openList);
    delete_stack(closedList);
    if(time == limit_time){
        time = -time;
    }
    return time;
}


--------------------------------------------------------------------------------



<From tests/tests.c>



#include <stdio.h>
#include <stdlib.h>
#include <math.h>
#include <time.h>
#include "../values.h"
#include "../tools/draw.h"
#include "../scene/scene.h"
#include "../entity/player.h"
#include "../entity/guard.h"
#include "../scene/simulation.h"
#include "../tools/out.h"
#include "../entity/position.h"

void test_scene(){
    scene_object* scene = new_scene(10,5,0);
    scene_init(scene,scene->init);
    display_scene(scene,stdout);
    delete_scene(scene);
}

void test_player(){
    player* bot = new_player(0,0);
    delete_player(bot);
}


void test_guard(){
    guard_list* guards = new_guard_list(10,0,ROTATION_SPEED,1,4);
    delete_guard_list(guards);
}

void test_postion(){
    scene_object* scene = new_scene(10,5,0);
    position* pos = new_position(0,0);
    scene_init(scene,scene->init);

    draw_position(pos,scene,1);
    move_right(pos,scene);
    printf("position : (%d,%d)\n",pos->x,pos->y);
    draw_position(pos,scene,2);
    move_up(pos,scene);
    printf("position : (%d,%d)\n",pos->x,pos->y);
    move_down(pos,scene);
    printf("position : (%d,%d)\n",pos->x,pos->y);
    move_left(pos,scene);
    printf("position : (%d,%d)\n",pos->x,pos->y);
    draw_position(pos,scene,3);
    move_down(pos,scene);
    printf("position : (%d,%d)\n",pos->x,pos->y);
    move_left(pos,scene);
    printf("position : (%d,%d)\n",pos->x,pos->y);
    draw_position(pos,scene,4);
    display_scene(scene,stdout);

    moveto(pos,9,4);
    printf("position : (%d,%d)\n",pos->x,pos->y);
    draw_position(pos,scene,5);
    move_right(pos,scene);
    printf("position : (%d,%d)\n",pos->x,pos->y);
    draw_position(pos,scene,6);
    display_scene(scene,stdout);

    move_down(pos,scene);
    printf("position : (%d,%d)\n",pos->x,pos->y);
    draw_position(pos,scene,5);
    display_scene(scene,stdout);

    delete_position(pos);
    delete_scene(scene);
}

void test_select_circle(){//pas bon à partir de la
    scene_object* scene = new_scene(101,101,0);
    scene_init(scene,scene->init);
    position* pos = new_position(50,50);
    list_position* circle = select_circle(pos,45);

    for(int i = 0; i < circle->size;i++){
        draw_position(circle->tab[i],scene,1);
        wait(scene);
        display_scene(scene,stdout);
    }

    printf("Il y a %d points !!\n\n",circle->size - 1);

    delete_position(pos);
    delete_list_position(circle);
    delete_scene(scene);
```

```
}

void test_select_arc(){
    scene_object* scene = new_scene(101,101,0);
    scene_init(scene,scene->init);
    position* pos = new_position(20,20);

    for(int j=0;j < 37;j++){
        list_position* arc = select_arc(pos,45,10*j,60);
        //modifier 360 pour voir la différence suivant les angels
        check_selection(arc,scene);
        arc = delete_double(arc);
        draw_position(pos,scene,0);
        for(int i=0; i<arc->size;i++){
            draw_position(arc->tab[i],scene,1);
        }
        draw_line(pos,arc->tab[arc->size/2],scene,1,4);
        wait(scene);
        display_scene(scene,stdout);
        scene_init(scene,scene->init);
        printf("Il y a %d points !!\n\n",arc->size - 1);
        delete_list_position(arc);
    }


    delete_position(pos);
    delete_scene(scene);
}

void test_draw_line(){
    scene_object* scene = new_scene(101,101,0);
    scene_init(scene,scene->init);
    position* pos = new_position(50,50);
    list_position* circle = delete_double(select_circle(pos,45));

    for(int i = 0; i< circle->size;i++){
        draw_line(pos,circle->tab[i],scene,1,4);
        display_scene(scene,stdout);
        scene_init(scene,scene->init);
        wait(scene);
    }

    delete_position(pos);
    delete_list_position(circle);
    delete_scene(scene);
}

void test_draw_cone(){
    scene_object* scene = new_scene(201,201,0);
    scene_init(scene,scene->init);
    position* pos = new_position(99,50);

    for(int i = 0; i< 37;i++){
        /*quick_*/draw_cone_with_cross(pos,100,10*i,60,scene,1,4);
        // clean_holl(scene,1,1);
        display_scene(scene,stdout);
        scene_init(scene,scene->init);
        wait(scene);
    }

    delete_position(pos);
    delete_scene(scene);
}

void test_copy_scene(){
    scene_object* scene1 = new_scene(10,10,0);
    scene_object* scene2 = copy_scene(scene1);

    scene_init(scene2, scene2->init);
    display_scene(scene1,stdout);
    display_scene(scene2,stdout);

    delete_scene(scene1);
    delete_scene(scene2);
}

void test_simulate_path(int n){
    int h = HEIGHT, w = WIDTH;
    scene_object* scene = new_scene(w,h,0);
    scene->init=0;
    scene_init(scene,scene->init);
    int number = 50;
    guard_list* guards = new_guard_list(number,6,ROTATION_SPEED,1,4);
    player* bot = new_player(w/2,0);

    // printf("init angle : %d\n",guards->tab[0]->angle);
    // position pos = {50,50};
    // moveto(guards->tab[0]->pos,50,50);

    FILE* f = fopen("./out/test.txt","w");
    fprintf(f,"%d\n%d\n",scene->width,scene->height);
    printf("Score : %d\n",simulate(scene,guards,bot,3,n,true,f));
    fclose(f);

    delete_guard_list(guards);
    delete_player(bot);
    delete_scene(scene);
}

int main(void){
    int seed = time(NULL);
    // seed = 1716130203;
    srand(seed);
    printf("seed : %d\n",seed);
    // test_scene();
    // test_player();
    // test_guard();
    // test_postion();
    // test_select_circle();
    // test_select_arc();
    //test_draw_line();
    // test_draw_cone();
    // test_copy_scene();
    test_simulate_path(1);

    return 0;
}


-------------------------------------------------------------------------------



<From tools/draw.c>
```

```


#include <stdio.h>
#include <math.h>
#include "draw.h"

void check_position(position* pos, scene_object* scene){
    int b1 = pos->x/scene->width > 0;
    int b2 = pos->y/scene->height > 0;
    if(pos->x < 0 || b1){
        pos->x = b1*(scene->width-1);
    }
    if(pos->y < 0 || b2){
        pos->y = b2*(scene->height-1);
    }
}

void draw_position(position* pos, scene_object* scene, int color){
    check_position(pos,scene);
    scene->grid[pos->y][pos->x] = color;
}

void draw_cross(position* pos, scene_object* scene, int color){
    int x = pos->x;
    int y = pos->y;
    int all_x[5] = {x, x+1, x-1, x, x};
    int all_y[5] = {y, y, y, y+1, y-1};
    for(int i=0; i<5; i++){
        position* new_pos = new_position(all_x[i], all_y[i]);
        draw_position(new_pos,scene,color);
        delete_position(new_pos);
    }
}

list_position* select_circle(position* pos,int radius){
    int x = 0,y = radius, m = 5 - 4*radius;
    int cpt = 0;
    while(x<=y){
        cpt++;
        if(m>0){
            y = y - 1;
            m = m - 8*y;
        }
        x = x + 1;
        m = m + 8*x + 4;
    }//count

    int size = cpt*8;
    list_position* res = new_list_position(size);
    x = 0,y = radius, m = 5 - 4*radius;

    for(int i = 0;i<cpt;i++){
          // res->tab[i] = new_position(y+pos->x, x+pos->y);// 0 to pi/4

        // res->tab[2*cpt - i - 1] = new_position(x+pos->x, y+pos->y);// pi/2 to pi/4

          // res->tab[i + 2*cpt] = new_position(-x+pos->x, y+pos->y);// pi/2 to 3pi/4
          // res->tab[4*cpt - i - 1] = new_position(-y+pos->x, x+pos->y);// pi to 3pi/4
```

```
2794
2795            // res->tab[i + 4*cpt] = new_position(-y+pos->x, -x+pos-
>y);// pi to -3pi/4
2796            // res->tab[6*cpt - i - 1] = new_position(-x+pos->x, -
y+pos->y);// -pi/2 to -3pi/4
2797
2798            // res->tab[i + 6*cpt] = new_position(x+pos->x, -y+pos-
>y);// -pi/2 to -pi/4
2799            // res->tab[8*cpt - i - 1] = new_position(y+pos->x, -x+pos-
>y);// 0 to -pi/4
2800
2801        for(int j =0; j<8;j++){
2802            int var1 = (j&1) ? (j+1)*cpt - i - 1 : j*cpt + i ;
2803            int var2_ = (!((j>>1)&1)^(j&1))*y + (((j>>1)&1)^(j&1))*x;
2804            int var2 = (j>1 && j<6) ? -var2_ : var2_ ;
2805            int var3_ = (!((j>>1)&1)^(j&1))*x + (((j>>1)&1)^(j&1))*y;
2806            int var3 = (j<4) ? var3_ : -var3_ ;
2807            res->tab[var1] = new_position(var2 + pos->x,var3 + pos-
>y);
2808        }
2809
2810        if(m>0){
2811            y = y - 1;
2812            m = m - 8*y;
2813        }
2814        x = x + 1;
2815        m = m + 8*x + 4;
2816    }
2817
2818    res->size = size;
2819
2820    return res;
2821 }
2822
2823 list_position* select_arc(position* pos, int radius, int direction,
int angle){
2824    list_position* circle = select_circle(pos, radius);
2825    direction = (360+direction%360)%360;
2826    int octant = circle->size/8;
2827
2828    int start=0;
2829    float pourcentage = angle/720.0;
2830    int nb_points = circle->size  * pourcentage;
2831
2832    if(direction !=0){
2833        int dire_45 = (direction-1)/45;
2834        start = (7 - dire_45)*octant;
2835        int condition1 = 45<direction && dire_45 == 1;
2836        int condition2 = 225<direction && dire_45 == 5;
2837        if(condition1 || condition2){
2838            start++;
2839        }
2840    }
2841
2842    // if(325 < direction) start = 0; //switch
2843    // else if(270 < direction) start = octant;
2844    // else if(direction == 270) start = 2*octant;
2845    // else if(225 < direction) start = 2*octant+1;
2846    // else if(180 < direction) start = 3*octant;
2847    // else if(135 < direction) start = 4*octant;
2848    // else if(90 < direction) start = 5*octant;
2849    // else if(direction == 90) start = 6*octant;
2850    // else if(45 < direction) start = 6*octant+1;
2851    // else if(0 < direction) start = 7*octant;
2852
2853    position* mid = circle->tab[start];
2854    int num = start;
2855
2856    if(direction != 90 && direction != 270){
2857        double tan_direction = tan(direction*3.14/180);
2858        double tan_mid = (double)(pos->y - mid->y)/(pos->x - mid->x);
2859        for(int i = 0; i < octant;i++){
2860            double tan_point = (double)(pos->y - circle->tab[start+i]->y)/
2861                               (circle->tab[start+i]->x - pos->x);
2862            if(fabs(tan_mid - tan_direction) > fabs(tan_point -
tan_direction)){
2863                tan_mid = tan_point;
2864                num = start + i;
2865                mid = circle->tab[num];
2866            }
2867        }
2868    }
2869
2870    list_position* res = new_list_position(2*nb_points+1);
2871    res->tab[nb_points] = copy_position(mid);
2872
2873    for(int i = 0; i<nb_points;i++){
2874        position* pos1 = circle->tab[(circle->size + num - i)%circle->size];
2875        position* pos2 = circle->tab[(i+num)%circle->size];
2876        res->tab[nb_points - i - 1] = copy_position(pos1);
2877        res->tab[1 + i + nb_points] = copy_position(pos2);
2878    }
2879
2880    delete_list_position(circle);
2881
2882    return res;
2883
2884 }
2885
2886 void check_selection(list_position* arc, scene_object* scene) {
2887    for(int i = 0;i<arc->size;i++) {
2888        check_position(arc->tab[i],scene);
2889    }
2890 }
2891
2892 list_position* delete_double(list_position* arc){
2893    int nb_doubles = 0;
2894    for(int i = 1; i <arc->size;i++){
2895        if (equals(arc->tab[i-1],arc->tab[i])){
2896            nb_doubles++;
2897        }
2898    }
2899
2900    list_position* res = new_list_position(arc->size - nb_doubles);
2901    res->tab[0] = copy_position(arc->tab[0]);
2902    int nb_deleted = 0;
2903
2904    for(int i = 1; i <arc->size;i++){
2905        if (!equals(arc->tab[i-1],arc->tab[i])){
2906            res->tab[i-nb_deleted] = copy_position(arc->tab[i]);
2907        }
2908        else{
2909            nb_deleted++;
2910        }
2911    }
2912
2913    delete_list_position(arc);
2914    return res;
2915 }
2916
2917 void _draw_line(position* p1, position* p2, scene_object* scene,
2918                 int color,int color2, bool cross){
2919
2920    int dx = abs(p2->x - p1->x);
2921    int dy = abs(p2->y - p1->y);
2922    int incX = (p1->x < p2->x) ? 1 : -1;
2923    int incY = (p1->y < p2->y) ? 1 : -1;
2924    int err = dx - dy;
2925    position* p = new_position(p1->x,p1->y);
2926    draw_position(p, scene, color);
2927    do{
2928        if(!cross){
2929            draw_position(p, scene, color);
2930        }
2931        int err2 = 2 * err;
2932        if (err2 > -dy) {
2933            err -= dy;
2934            if(cross && err2 < dx){
2935                p->y += incY;
2936                draw_position(p, scene, color);
2937                p->y -= incY;
2938            }
2939            p->x += incX;
2940            if(cross){
2941                draw_position(p, scene, color);
2942            }
2943        }
2944        if (err2 < dx) {
2945            err += dx;
2946            p->y += incY;
2947            if(cross){
2948                draw_position(p, scene, color);
2949            }
2950        }
2951    } while (!equals(p,p2));
2952    draw_position(p, scene, color2);
2953    delete_position(p);
2954 }
2955
2956 void draw_line(position* p1, position* p2, scene_object* scene,
2957                int color,int color2){
2958    _draw_line(p1,p2,scene,color,color2,false);
2959 }
2960
2961 void draw_line_in_cross(position* p1, position* p2, scene_object*
scene,
2962                int color,int color2){
2963    _draw_line(p1,p2,scene,color,color2,true);
2964 }
2965
2966 void draw_cone(position* pos, int radius, int direction, int angle,
2967                scene_object* scene, int color, int color2){
2968
2969    list_position* arc = select_arc(pos,radius,direction,angle);
2970    check_selection(arc,scene);
2971    arc = delete_double(arc);
```

```
    draw_position(pos,scene,color);

    for(int i=0;i<arc->size;i++){
        draw_line(pos,arc->tab[i],scene,color,color2);
    }

    delete_list_position(arc);
}

void draw_cone_with_cross(position* pos, int radius, int direction,
int angle,
               scene_object* scene, int color, int color2){
    list_position* arc = select_arc(pos,radius,direction,angle);
    check_selection(arc,scene);
    arc = delete_double(arc);
    draw_position(pos,scene,color);

    draw_line(pos,arc->tab[0],scene,color,color2);
    draw_line(pos,arc->tab[arc->size-1],scene,color,color2);
    for(int i=1;i<arc->size-1;i++){
        draw_line_in_cross(pos,arc->tab[i],scene,color,color2);
    }

    delete_list_position(arc);
}

void quick_draw_cone(position* pos,int radius,int direction,int
angle,
                     scene_object* scene,int color,int color2){
    list_position* arc = select_arc(pos,radius,direction,angle);
    check_selection(arc,scene);
    arc = delete_double(arc);

    draw_line(pos,arc->tab[0],scene,color,color2);
    draw_line(pos,arc->tab[arc->size-1],scene,color,color2);

    for(int i=0;i<arc->size;i++){
        draw_position(arc->tab[i],scene,color);
    }

    delete_list_position(arc);
}

void print_trajectory(scene_object* scene, priority_list* list, int
color){
    if(!empty_priority_list(list)){
        tree_node* elt = list->tas[1]->way;
        tree_node* prev = elt->prev;
        if(prev != NULL){
            while (prev->prev != NULL){
                position pos = {elt->pos.x,elt->pos.y};
                draw_position(&pos,scene,color);
                elt = prev;
                prev = elt->prev;
            }
        }
    }
}

void clean_holl(scene_object* scene, int color, int color2){
    for (int i = 0; i < scene->height; i++) {
        for (int j = 0; j < scene->width; j++) {
            if (scene->grid[i][j] == scene->init) {
                int neighbors = 0;
                if (i > 0 &&
                   (scene->grid[i - 1][j] == color
                   || scene->grid[i - 1][j] == color2)) neighbors++;
                if (i < scene->height - 1 &&
                   (scene->grid[i + 1][j] == color
                   || scene->grid[i + 1][j] == color2)) neighbors++;
                if (j > 0 &&
                   (scene->grid[i][j - 1] == color
                   || scene->grid[i][j-1] == color2)) neighbors++;
                if (j < scene->width - 1 &&
                   (scene->grid[i][j + 1] == color
                   || scene->grid[i][j + 1] == color2)) neighbors++;

                if (neighbors >= 3) scene->grid[i][j] = color;
            }
        }
    }
}
```

```
<From tools/graph.c>


#include <stdio.h>
#include <string.h>
#include <stdlib.h>
#include "graph.h"
#include "out.h"

list_position* neighbors2(scene_object* scene, position* pos, int height, int
width) {
    list_position* list = new_list_position(5);
    int cpt = 0;

    for (int dx = -1; dx <= 1; dx++) {
        for (int dy = -1; dy <= 1; dy++) {
            if (abs(dx * dy) != 1) {
                int new_x = pos->x + dx;
                int new_y = pos->y + dy;

                if (new_x >= 0 && new_x < width && new_y >= 0
                    && new_y < height && scene->grid[new_y][new_x] == scene-
>init) {
                    list->tab[cpt++] = new_position(new_x, new_y);
                }
            }
        }
    }

    list->size = cpt;
    return list;
}

list_position* neighbors(scene_object* scene, position* pos, int height, int
width){
    list_position* list = new_list_position(5);
    int cpt=0;

    int positionsx[5] = {pos->x,pos->x,pos->x,pos->x+1,pos->x-1};
    int positionsy[5] = {pos->y-1,pos->y,pos->y+1,pos->y,pos->y};

    for(int i = 0;i<5;i++){
        int new_x = positionsx[i];
        int new_y = positionsy[i];
        int condition = new_x >= 0 && new_x < width && new_y >= 0 &&
new_y < height;
        if (condition && scene->grid[new_y][new_x] == scene->init) {
            list->tab[cpt] =
new_position(positionsx[i],positionsy[i]);
            cpt++;
        }
    }

    list->size = cpt;
    return list;
}

position* A_star(scene_object** scene_list, int time,int accuracy,
position* default_end,
                 stack* closedList, priority_list* openList){
    int height = scene_list[0]->height;
    int width = scene_list[0]->width;
    position end = *default_end;

    while (!empty_priority_list(openList)){
        data* u = openList->tas[1];
        end.x = u->way->pos.x;

        if((u->node->x == end.x && u->node->y == end.y) || u->cout >=
accuracy-1){
            position* res = copy_position(u->node);
            if(u->way->start != NULL){
                delete_position(res);
                res = copy_position(&(u->way->start->pos));
            }
            return res;
        }

        u = remove_rac(openList);
        list_position* voisins = neighbors(scene_list[(time+u-
>cout+1)%accuracy],u->node,height,width);
        bool have_neighbors = false;

        for(int i = 0; i<voisins->size;i++){
            position* elt = voisins->tab[i];
            if(!(is_in_stack(elt,u->cout+1,closedList) ||
                 is_in_priority(openList,elt,u->cout+1))){
                have_neighbors = true;
                elt = copy_position(elt);
                int d = range(&end,elt);
                tree_node* start = u->way->start;
                if(start == NULL && u->way->prev != NULL){
                    start = u->way;
                }
                tree_node* new_way = new_tree_node(*elt,start,u->way);
                add_next_tree_node(u->way,new_way);
                data* v = new_data(u->cout+1+d,elt,new_way,u->cout+1);
```

```
                insert(v,openList);
            }
        }
        if(!have_neighbors){
            delete_tree_node(u->way);//soulage la mémoire
        }
        delete_list_position(voisins);
        enstack(new_stack_elt(copy_position(u->node),u->cout),closedList);
        delete_data(u);
    }
    return NULL;
}


--------------------------------------------------------------------------------


<From tools/out.c>


#include "out.h"
#include <stdio.h>
#include <stdlib.h>

char* convert_to_string(int value){
    switch (value)
    {
    case 0 :
        return " ";
        break;
    case 1 :
        return "1";
        break;
    case 2 :
        return "x";
        break;
    case 3 :
        return "3";
        break;
    case 4 :
        return "4";
        break;
    case 5 :
        return "5";
        break;
    case 6 :
        return "6";
        break;
    default:
        return "";
        break;
    }
}

void display_grid(int** grid, int height, int width, FILE* file){
    for(int i = 0;i<height;i++){
        fprintf(file,"| ");
        for(int j=0;j<width;j++){
            fprintf(file,"%s ",convert_to_string(grid[i][j]));
        }
        fprintf(file,"|\n");
    }
    fprintf(file,"\n");
}

void display_data(int** grid, int height, int width,FILE* file){
    fprintf(file,"O\n");
    for(int i=0;i<height;i++){
        for(int j=0;j<width;j++){
            if(grid[i][j]!=0){
                fprintf(file,"%d;%d;%d\n",i,j,grid[i][j]);
            }
        }
    }
    fprintf(file,"C\n");
}

void display_scene(scene_object* scene,FILE* file){
    if(file == stdout){
        fprintf(file,"Scène en : %d par %d ; numéro de tour : %d\n\n",
            scene->width,scene->height,scene->time);
        display_grid(scene->grid,scene->height,scene->width,file);
    }
    else{
        display_data(scene->grid,scene->height,scene->width,file);
    }

}


--------------------------------------------------------------------------------


<From tools/priority.c>


#include <stdio.h>
#include <stdlib.h>
#include "priority.h"

data* new_data(int prio, position* pos, tree_node* old, int cout){
    data* res = malloc(sizeof(data));
    res->prio = prio;
    res->node = pos;
    res->way = old;
    res->cout = cout;
    return res;
}

void delete_data(data* e){
    delete_position(e->node);
    free(e);
}

priority_list* new_priority_list(int n){
    priority_list* list = malloc(sizeof(priority_list));
    data** tab = calloc(n+1, sizeof(data*));
    list->size = n;
    list->tas = tab;
    list->tas[0] = new_data(0,0,0,0);
    return list;
}

bool empty_priority_list(priority_list* list){
    return list->tas[0]->cout == 0;
}

void delete_priority_list(priority_list* list){
    while (!empty_priority_list(list)){
        delete_data(remove_rac(list));
    }
    free(list->tas[0]);
    free(list->tas);
    free(list);
}

void swap(int i,int j,priority_list* list){
    data* tmp = list->tas[i];
    list->tas[i] = list->tas[j];
    list->tas[j] = tmp;
}

void percolate_up(int k,priority_list* list){
    if(k>1){
        int m = k/2;
        if(list->tas[m]->prio > list->tas[k]->prio){
            swap(m,k,list);
            percolate_up(m,list);
        }
    }
}

int choose_son(int k, priority_list* list){
    if(k*2<list->tas[0]->cout){
        if(list->tas[2*k]->prio < list->tas[2*k+1]->prio){
            return 2*k;
        }
        else{
            return 2*k+1;
        }
    }
    else{
        if(2*k == list->tas[0]->cout){
            return 2*k;
        }
        else{
            return -1;
        }
    }
}

void percolate_down(int k, priority_list* list){
    int m = choose_son(k,list);
    if(m>-1 && list->tas[m]->prio < list->tas[k]->prio){
        swap(m,k,list);
        percolate_down(m,list);
    }
}

void insert(data* e,priority_list* list){
```

```
    if(list->tas[0]->cout < list->size){
        list->tas[0]->cout++;
        list->tas[list->tas[0]->cout] = e;
        percolate_up(list->tas[0]->cout,list);
    }
    else{
        list->tas = realloc(list->tas, (2*list-
>size+1)*sizeof(data*));
        list->size = list->size*2;
        insert(e,list);
    }
}

data* remove_rac(priority_list* list){
    data* res = NULL;
    if(list->tas[0]->cout > 0){
        res = list->tas[1];
        list->tas[1] = list->tas[list->tas[0]->cout];
        list->tas[0]->cout--;
        percolate_down(1,list);
    }
    return res;
}

bool is_in_priority(priority_list* list, position* pos, int cout) {
    for(int i = 1; i <= list->tas[0]->cout; i++){
        if(equals(list->tas[i]->node,pos) && list->tas[i]->cout <=
cout) return true;
    }
    return false;
}

void adjust_priority_list(priority_list* list, position* pos,
position end){
    int n = list->tas[0]->cout;
    priority_list* tmp = new_priority_list(list->size);

    for(int i = 1; i<=n; i++){
        if(list->tas[i]->way->start != NULL
            && equals(&(list->tas[i]->way->start->pos),pos)){
            data* e = list->tas[i];
            e->cout--;
            e->prio = e->cout+range(e->node,&end);

            e->way->start->prev = NULL;
            e->way->start = new_start_tree_node(e->way);

            insert(e,tmp);
        }
        else{
            delete_tree_node(list->tas[i]->way);
            delete_data(list->tas[i]);
        }
    }

    free(list->tas[0]);
    free(list->tas);
    list->tas = tmp->tas;
    free(tmp);
}

--------------------------------------------------------------------------------



<From tools/stack.c>



#include <stdio.h>
#include <stdlib.h>
#include "stack.h"

stack_elt* new_stack_elt(position* pos,int time){
    stack_elt* elt = malloc(sizeof(stack_elt));
    elt->pos = pos;
    elt->time = time;
    elt->next = NULL;
    return elt;
}

void delete_stack_elt(stack_elt* elt){
    delete_position(elt->pos);
    free(elt);
}

stack* new_stack(){
    stack* s = malloc(sizeof(stack));
    s->top = NULL;
    return s;
}

void delete_stack(stack* s){
    while(!empty_stack(s)){
        delete_stack_elt(destack(s));
    }
    free(s);
}

void enstack(stack_elt* elt,stack* s){
    elt->next = s->top;
    s->top = elt;
}

stack_elt* destack(stack* s){
    if(!empty_stack(s)){
        stack_elt* elt = s->top;
        s->top = elt->next;
        return elt;
    }
    return NULL;
}

bool empty_stack(stack* s){
    return s->top == NULL;
}

bool is_in_stack(position* pos, int time, stack* s){
    stack_elt* elt = s->top;
    while (elt != NULL){
        if(equals(pos,elt->pos) && time == elt->time){
            return true;
        }
        elt = elt->next;
    }
    return false;
}

void adjust_stack(stack* s, int time){
    stack_elt* elt = s->top;
    while(elt != NULL){
        stack_elt* next = elt->next;
        if(next != NULL && next->time < time){
            elt->next = next->next;
            delete_stack_elt(next);
        }
        else{
            elt = next;
        }
    }
    if(s->top != NULL && s->top->time < time){
        stack_elt* elt = s->top;
        s->top = s->top->next;
        delete_stack_elt(elt);
    }
}


--------------------------------------------------------------------------------
----



<From out/sublim.py>



from PIL import Image
import numpy as np

from const import *

def color(n):
    match n:
        case 1:
            return [0,0,255]
        case 2:
            return [255,0,0]
        case 3:
            return [0,255,0]
        case 4:
            return [0,0,100]
        case 5:
            return [255,0,255]
        case _:
            return [255,255,255]

def minmax(t,cpt):
    min,max = cpt,0
    for l in t:
        for v in l:
            if v[2]<min:
                min = v[2]
```

```
            elif v[2]>max:
                max = v[2]
    return min,max

def grids(filename,k):
    file = open(rac+filename,"r")
    width = int(file.readline())
    height = int (file.readline())
    cpt = 0
    img = None
    array = None
    line = file.readline()
    while line:
        if line[0] == "O":
            array = np.zeros([width*k,height*k,3],dtype=np.uint8)
            array[:,:] = [255,255,255]
        elif line[0] == "C":
            img = Image.fromarray(array)
            img.save(rac+"jpg/"+str(cpt)+".bmp","bmp")
            cpt += 1
        else:
            values = line.split(";")
            v1,v2,v3=int(values[0])*k,int(values[1])*k,int(values[2])
            array[v1:v1+k,v2:v2+k] = color(v3)
        line = file.readline()
    file.close()
    return cpt

def mean_grid(filename,cpt,k):
    file = open(rac+filename,"r")
    width = int(file.readline())
    height = int (file.readline())
    mid = np.zeros([width,height,3],dtype=np.uint64)
    b = np.zeros([width,height])
    line = file.readline()
    while line:
        if line[0] == "O":
            b[:,:] = True
        elif line[0] != "C":
            values = line.split(";")
            v1,v2=int(values[0]),int(values[1])
            if b[v1,v2]:
                mid[v1,v2,2] += 1
                b[v1,v2] = False
        line = file.readline()
    min,max = minmax(mid,cpt)
    new_mid = np.zeros([width*k,height*k,3],dtype=np.uint8)
    for i in range(width):
        for j in range(height):
            new_mid[i*k:(i+1)*k,j*k:(j+1)*k] = [255,255,255]
            new_mid[i*k:(i+1)*k,j*k:(j+1)*k,1:3] = 255 -
int((mid[i,j,2]-min)*(255/(max-min+1)))
    img = Image.fromarray(new_mid)
    img.save(rac+"mid.bmp")
    file.close()


--------------------------------------------------------------------------
-----


<From out/analyses.py>



from matplotlib import pyplot as plt
from os import listdir

from const import *

# def color(k):
#     match k:
#         case 0:
#             return "r"
#         case 1:
#             return "g"
#         case 2:
#             return "y"
#         case 3:
#             return "m"
#         case 4:
#             return "b"
#         case 5:
#             return "c"
#         case _:
#             return "k"

def color(k):
    match k:
        case 1:
            return "g"
        case 5:
            return "r"
        case _:
            return "b"

def noms(k):
    match k:
        case 0:
            return "immobile"
        case 1:
            return "rotation"
        case 2:
            return "aléatoir"
        case 3:
            return "aller-retour"
        case 4:
            return "patrouille carré"
        case 5:
            return "en essaim"
        case 6:
            return "en S"

def fill_dico(dico):
    files = sorted(listdir(rac+folder))
    for f in files:
        data = f.split("_")
        behave = int(data[0])
        nb = int(data[1].split(".")[0])
        file = open(rac+folder+f,"r")
        line = file.readline()
        if not (behave in dico):
            dico[behave] = {}
        dico[behave][nb] = []
        while line:
            try:
                l = int(line)
                if l == -200 and behave == 0:
                    dico[behave][nb].append(-500)
                else:
                    dico[behave][nb].append(l)
            except:
                pass
            line = file.readline()


def sum(l):
    sum = 0
    for e in l:
        sum += e
    return sum

def mean(l):
    res = sum(l)
    if res == 0:
        return 0
    else:
        return res/len(l)

def smooth(x,y) :
    k = 5
    new_x = []
    new_y = []
    for xi in x :
        new_x.append(xi)
    for i in range(len(y)):
        new_y.append(mean(y[max(i-k,0):i+k+1]))
    return new_x,new_y

def plot_mean_dico(dico,legend):
    plt.figure("Mean")
    for num_fil,d in dico.items():
        x = []
        y = []
        for nb,l in d.items():
            x.append(nb)
            y.append(mean(l))
        x,y = smooth(x,y)
        plt.plot(x,y,color=color(num_fil),label=legend+noms(num_fil))
    plt.legend()
    plt.xlabel("nombre de gardes")
    plt.ylabel("temps moyen mis par le joueur")
    plt.show()

def scatter_all_dico(dico,legend):
    for num_fil,d in dico.items():
        cpt = 0
        plt.figure(legend+" : "+str(num_fil))
        for nb,l in d.items():
            cpt += len(l)
            y = [e for e in l if e<301 and e>-301]
            x = [nb for i in range(len(y))]
            plt.scatter(x,y,color=color(num_fil))
        plt.xlabel("nombre de gardes")
```

```
        plt.ylabel("temps mis par le joueur")
        plt.show()
        print(cpt)

def plot_achive_rate(dico,legend):
    plt.figure("Rate")
    for num_fil,d in dico.items():
        x = []
        y = []
        for nb,l in d.items():
            x.append(nb)
            y.append(sum([1/len(l) for e in l if e > 0]))
        x,y = smooth(x,y)
        plt.plot(x,y,color=color(num_fil),label=legend+noms(num_fil))
    plt.legend()
    plt.xlabel("nombre de gardes")
    plt.ylabel("taux de réussite du joueur")
    plt.show()


--------------------------------------------------------------------------
-----


<From out/const.py>



rac = "out/"
folder = "view/"

--------------------------------------------------------------------------
-----


<From out/main.py>



import glob
import shutil
import os

from sublim import *
from analyses import *

if os.path.exists(rac+"jpg"):
    shutil.rmtree(rac+"jpg")
os.mkdir(rac+"jpg")

if os.path.exists(rac+"test.txt"):
    cpt = grids("test.txt",3)
    mean_grid("test.txt",cpt,3)

    frames = [Image.open(image) for image in
sorted(glob.glob(rac+"jpg/*.bmp"), key=lambda x:
int(os.path.basename(os.path.splitext(x)[0])))]
    frame_one = frames[0]
    frame_one.save(rac+"test.gif", format="GIF",
append_images=frames,save_all=True, duration=80, loop=1)


if os.path.exists(rac+folder) and listdir(rac+folder) != []:
    dico = {}
    fill_dico(dico)
    legend = "comportement : "
    plot_achive_rate(dico,legend)
    plot_mean_dico(dico,legend)
    # scatter_all_dico(dico,legend)

# shutil.rmtree(rac+"jpg")
```