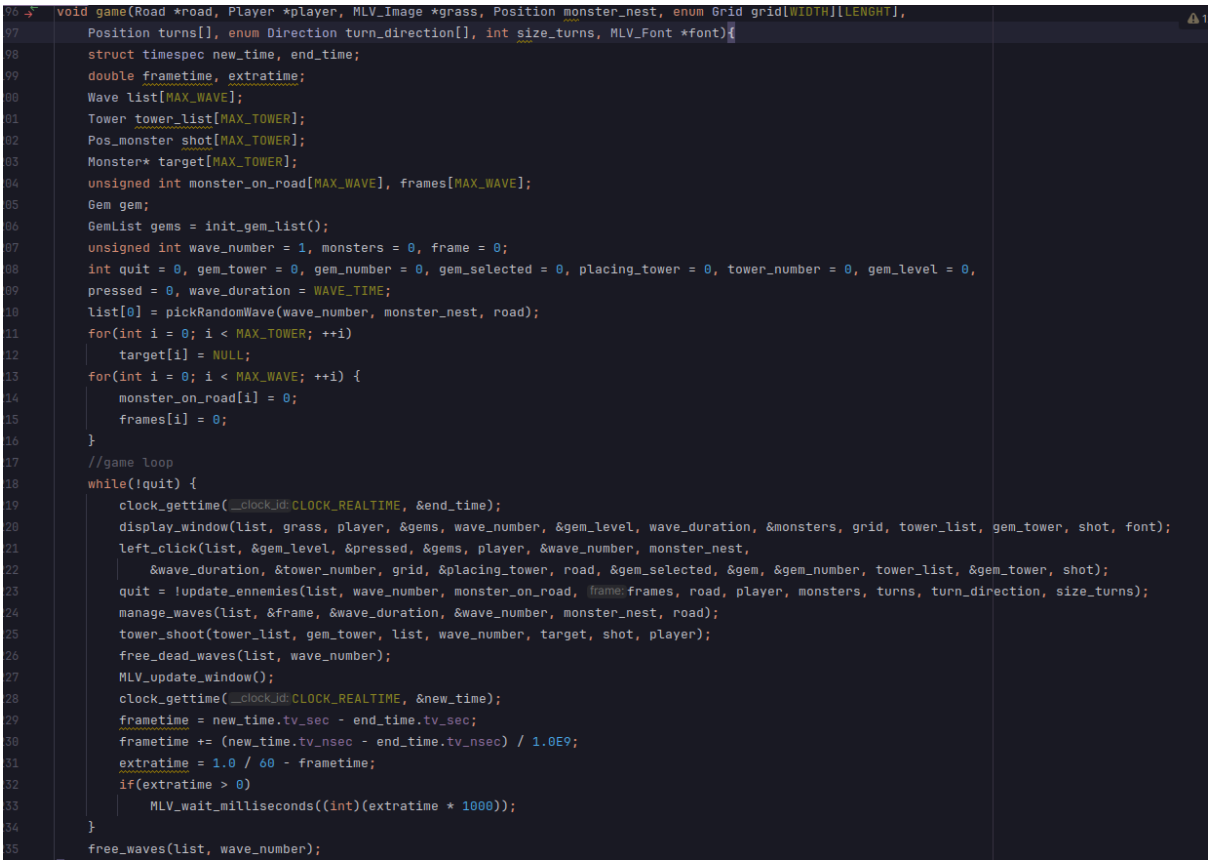
Projet Tower Defense

TOUATI Damien
GUNEY David

Voici la documentation technique de notre projet:

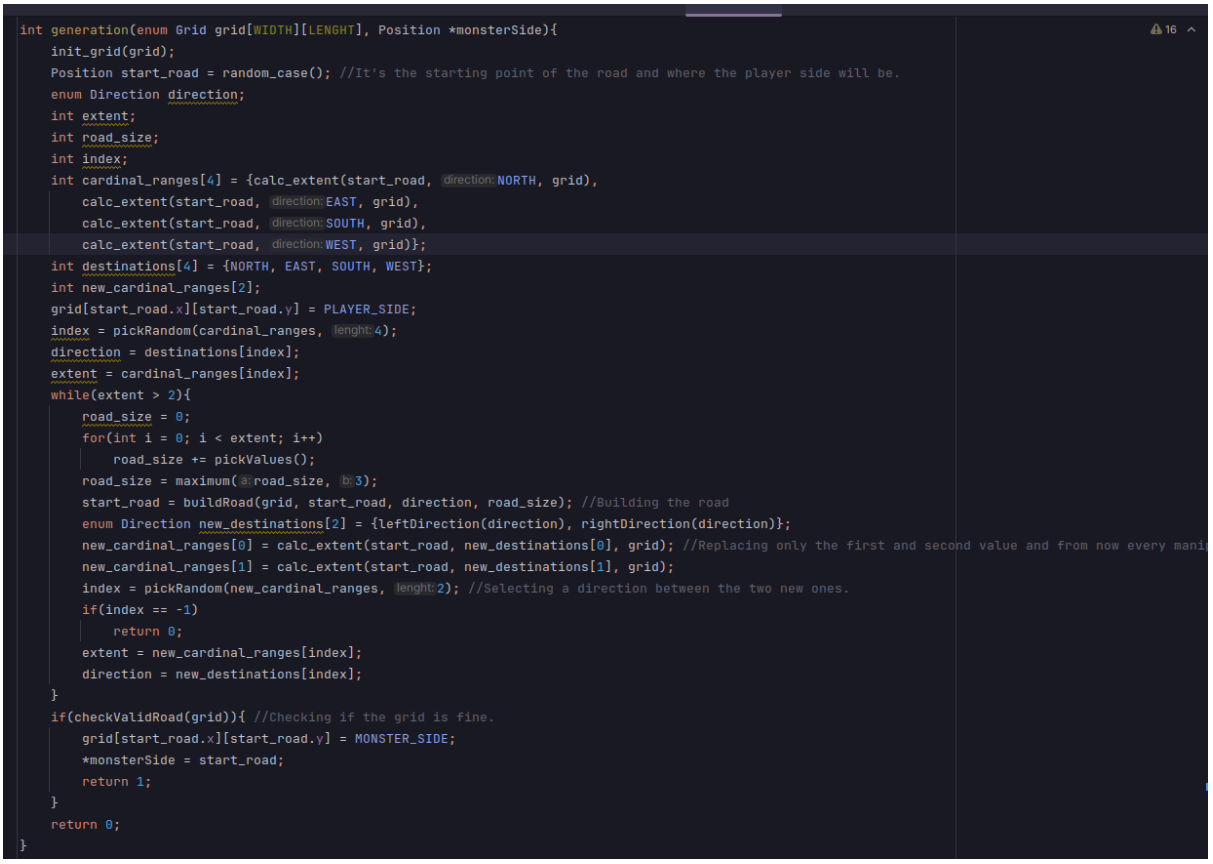
Cette fonction est la principale, elle permet le déroulement du jeu c'est celle qu'on appelle gameLoop.

On voit très clairement deux parties pour cette fonction, une avec toutes les variables déclarées qui permettent le bon fonctionnement du programme et le stockage des données et la boucle du jeu.
Dans cette deuxième partie le jeu se déroule avec 60 frames par seconde donc toutes les fonctions utilisées ci-dessous réalise une action frame par frame.



```
void game(Road *road, Player *player, MLV_Image *grass, Position monster_nest, enum Grid grid[WIDTH][LENGHT],
    Position turns[], enum Direction turn_direction[], int size_turns, MLV_Font *font){
    struct timespec new_time, end_time;
    double frametime, extratime;
    Wave list[MAX_WAVE];
    Tower tower_list[MAX_TOWER];
    Pos_monster shot[MAX_TOWER];
    Monster* target[MAX_TOWER];
    unsigned int monster_on_road[MAX_WAVE], frames[MAX_WAVE];
    Gem gem;
    GemList gems = init_gem_list();
    unsigned int wave_number = 1, monsters = 0, frame = 0;
    int quit = 0, gem_tower = 0, gem_number = 0, gem_selected = 0, placing_tower = 0, tower_number = 0, gem_level = 0,
    pressed = 0, wave_duration = WAVE_TIME;
    list[0] = pickRandomWave(wave_number, monster_nest, road);
    for(int i = 0; i < MAX_TOWER; ++i)
        target[i] = NULL;
    for(int i = 0; i < MAX_WAVE; ++i) {
        monster_on_road[i] = 0;
        frames[i] = 0;
    }
    //game loop
    while(!quit) {
        clock_gettime( clock_id: CLOCK_REALTIME, &end_time);
        display_window(list, grass, player, &gems, wave_number, &gem_level, wave_duration, &monsters, grid, tower_list, gem_tower, shot, font);
        left_click(list, &gem_level, &pressed, &gems, player, &wave_number, monster_nest,
            &wave_duration, &tower_number, grid, &placing_tower, road, &gem_selected, &gem, &gem_number, tower_list, &gem_tower, shot);
        quit = !update_ennemies(list, wave_number, monster_on_road, frames: frames, road, player, monsters, turns, turn_direction, size_turns);
        manage_waves(list, &frame, &wave_duration, &wave_number, monster_nest, road);
        tower_shoot(tower_list, gem_tower, list, wave_number, target, shot, player);
        free_dead_waves(list, wave_number);
        MLV_update_window();
        clock_gettime( clock_id: CLOCK_REALTIME, &new_time);
        frametime = new_time.tv_sec - end_time.tv_sec;
        frametime += (new_time.tv_nsec - end_time.tv_nsec) / 1.0E9;
        extratime = 1.0 / 60 - frametime;
        if(extratime > 0)
            MLV_wait_milliseconds((int)(extratime * 1000));
    }
    free_waves(list, wave_number);
```

La fonction ci dessous est celle qui permet de générer notre chemin que les monstres vont suivres:



```c
int generation(enum Grid grid[WIDTH][LENGHT], Position *monsterSide){
    init_grid(grid);
    Position start_road = random_case(); //It's the starting point of the road and where the player side will be.
    enum Direction direction;
    int extent;
    int road_size;
    int index;
    int cardinal_ranges[4] = {calc_extent(start_road, direction: NORTH, grid),
        calc_extent(start_road, direction: EAST, grid),
        calc_extent(start_road, direction: SOUTH, grid),
        calc_extent(start_road, direction: WEST, grid)};
    int destinations[4] = {NORTH, EAST, SOUTH, WEST};
    int new_cardinal_ranges[2];
    grid[start_road.x][start_road.y] = PLAYER_SIDE;
    index = pickRandom(cardinal_ranges, lenght: 4);
    direction = destinations[index];
    extent = cardinal_ranges[index];
    while(extent > 2){
        road_size = 0;
        for(int i = 0; i < extent; i++)
            road_size += pickValues();
        road_size = maximum(a: road_size, b: 3);
        start_road = buildRoad(grid, start_road, direction, road_size); //Building the road
        enum Direction new_destinations[2] = {leftDirection(direction), rightDirection(direction)};
        new_cardinal_ranges[0] = calc_extent(start_road, new_destinations[0], grid); //Replacing only the first and second value and from now every mani
        new_cardinal_ranges[1] = calc_extent(start_road, new_destinations[1], grid);
        index = pickRandom(new_cardinal_ranges, lenght: 2); //Selecting a direction between the two new ones.
        if(index == -1)
            return 0;
        extent = new_cardinal_ranges[index];
        direction = new_destinations[index];
    }
    if(checkValidRoad(grid)){ //Checking if the grid is fine.
        grid[start_road.x][start_road.y] = MONSTER_SIDE;
        *monsterSide = start_road;
        return 1;
    }
    return 0;
}
```

La fonction suit cet algorithme à la lettre:

1. Initialiser la grille à vide.
2. Choisir une case aléatoire avec une distance au moins 3 aux bords comme le nid des monstres.
3. Calculer les étendus pour toute direction cardinale, puis choisir aléatoirement la direction initiale avec une probabilité proportionnelle à l'étendu de chacune.
4. Si l'étendu de la direction courante est plus petit ou égal à 2, alors aller à l'étape 7. Sinon, supposons que l'étendu est n. Tirer n valeurs aléatoires indépendamment, chacune vaut 1 avec probabilité 3/4, et 0 sinon. Soit s la somme de ces valeurs aléatoires. Le nombre de case à ajouter au chemin dans la direction courante sera max(s, 3).
5. Avancer jusqu'à la case au bout du nouveau segment de chemin, et calculer ses étendus dans les deux directions en tournant 90 degrés à gauche et à droite. La nouvelle direction courante sera choisie parmi les deux avec une probabilité proportionnelle à leurs étendus.
6. Revenir à l'étape 4.
7. Vérifier si le chemin obtenu a fait au moins 7 virages et est de longueur au moins 75. Si oui, l'algorithme termine avec succès. Sinon, revenir à l'étape 1.

Cette fonction-ci permet la gestion des monstres :

```c
void manage_monster(int i, double *speed, Wave *wave, int size_turns, Position turns[], enum Direction turn_direction[]) {
    *speed = (0.9 + ((double) rand() / RAND_MAX) * 0.2) * wave->monster_list[i].speed;
    if(wave->monster_list[i].hp > 0) {
        for(int j = 0; j < size_turns; ++j) {
            //making const the monster speed in the corner
            if(dist_monster(wave->monster_list[i].position, turns[j]) <= 1)
                *speed = wave->monster_list[i].speed;
            //if we are in the corner we fix the position to the corner center
            if(dist_monster(wave->monster_list[i].position, turns[j]) <= *speed / 2) {
                if(wave->monster_list[i].position.x != turns[j].x || wave->monster_list[i].position.y != turns[j].y) {
                    wave->monster_list[i].position.x = turns[j].x;
                    wave->monster_list[i].position.y = turns[j].y;
                }
                wave->monster_list[i].direction = turn_direction[j];
            }
        }
        move_monster(&(wave->monster_list[i]), *speed);
        //manage the debuff time
        if(wave->monster_list[i].debuff.time_left > 0)
            wave->monster_list[i].debuff.time_left--;
        if(wave->monster_list[i].debuff.parasite_damage > 0 && wave->monster_list[i].debuff.time_left % 30 == 0)
            wave->monster_list[i].hp -= wave->monster_list[i].debuff.parasite_damage;
    }
}
```

Si les monstres sont dans un tournant on leur donne une vitesse constante et on fait en sorte que quand ils sont supposés être dans le centre du coin on applique leur position auy centre du coin pour ne pas avoir d’erreur de placement puis on les déplace et on applique leur débuff.