

Rapport

2019 - 2020

Groupe :

Shenjin LYU - Zixi CHEN

Enseignant :

Patrick Amar

Option :

Programmation objet avancée

Sommaire

Introduction	3
Présentation du projet	3
Spécifications	4
Caractéristiques de programme	5
Développement.....	6
Structure de programme	6
Hiérarchie des classes	7
Méthodes et algorithmes.....	9
Problèmes	17
Conclusion	18

Introduction

Présentation du projet

Nom de projet : Course au trésor dans un labyrinthe hanté

Date de rendu : 10 mai 2020

Les objectifs du projet :

- ✓ Apprendre à écrire un programme à partir de spécifications informelles.
- ✓ Apprendre à écrire lisiblement en les commentant judicieusement.
- ✓ Entrainer à encapsuler des données et des méthodes pour obtenir des programmes robustes.
- ✓ Comprendre l'intérêt de séparer la partie traitement d'un programme de sa partie interface utilisateur.

Le projet est fait en binôme. Les classes et les fonctions C++ pour afficher en 3D avec OpenGL du labyrinthe et des textures sont fournis.

Il faut qu'on réalise un jeu de type Kill'emAll en plus simple avec un affichage 3D. Le jeu doit pouvoir fonctionner en 3 modes :

- Autonome : le joueur joue seul contre la machine.
- Serveur : le jeu est en réseau (contre un seul client)
- Client : le jeu est en réseau (contre un seul serveur)

Spécifications

Ce qu'est sensé faire le programme

1. Labyrinthe

Le programme doit d'abord créer un labyrinthe. Il génère un labyrinthe basé sur une carte, y compris en identifiant et en créant des murs, des caisses, des affiches et un trésor. Il trouve l'emplacement de départ de chasseur et des gardiens et créez-les à cet emplacement.

2. Personnage

Lorsque le programme crée un personnage, il doit initialiser ses attributs, par exemple sa santé, sa précision de tir, sa vitesse de déplacement et son potentiel de protection... Ces attributs affecteront son mouvement, son comportement et sa capacité de combat.

3. Mouvement

Le programme doit vérifier si le mouvement d'un personnage ou d'une boule de feu est réalisable selon le labyrinthe. Pour un gardien, le programme lui donne deux types de mouvements, la patrouille en mode défense et la recherche de chasseurs en mode attaque.

4. Combat

Le programme doit calculer le champ de vision du gardien en mode attaque et laisser le gardien tirer sur le chasseur après qu'il a vu le chasseur. Le programme vérifie si la boule de feu frappe un personnage et met à jour sa valeur de santé, puis juge s'il est décédé en fonction de la valeur de santé.

Format des fichiers

Le format du labyrinthe est le fichier .txt.

Caractère	Objet
+ ; - ;	Mur
C	Chasseur
G	Gardien
x	Caisse
T	Trésor
a ; b	Affiche

Le format du son est le fichier .wav.

Le format du modèle est un fichier .jpg plus un fichier .md2.

Le format de la texture est soit le fichier .jpg, soit le fichier .tga.

Caractéristiques de programme

Le programme a réalisé le jeu en mode autonome.

Le joueur joue seul contre la machine. Le joueur peut attaquer les gardiens et les tuer. Il peut recouvrer son capital.

Les gardiens peuvent décider de chercher le joueur et lui attaquer ou de rester dans sa zone pour défendre le joueur. Ils peuvent aussi recouvrer leur capital.

Le mode serveur n'a pas été réalisé.

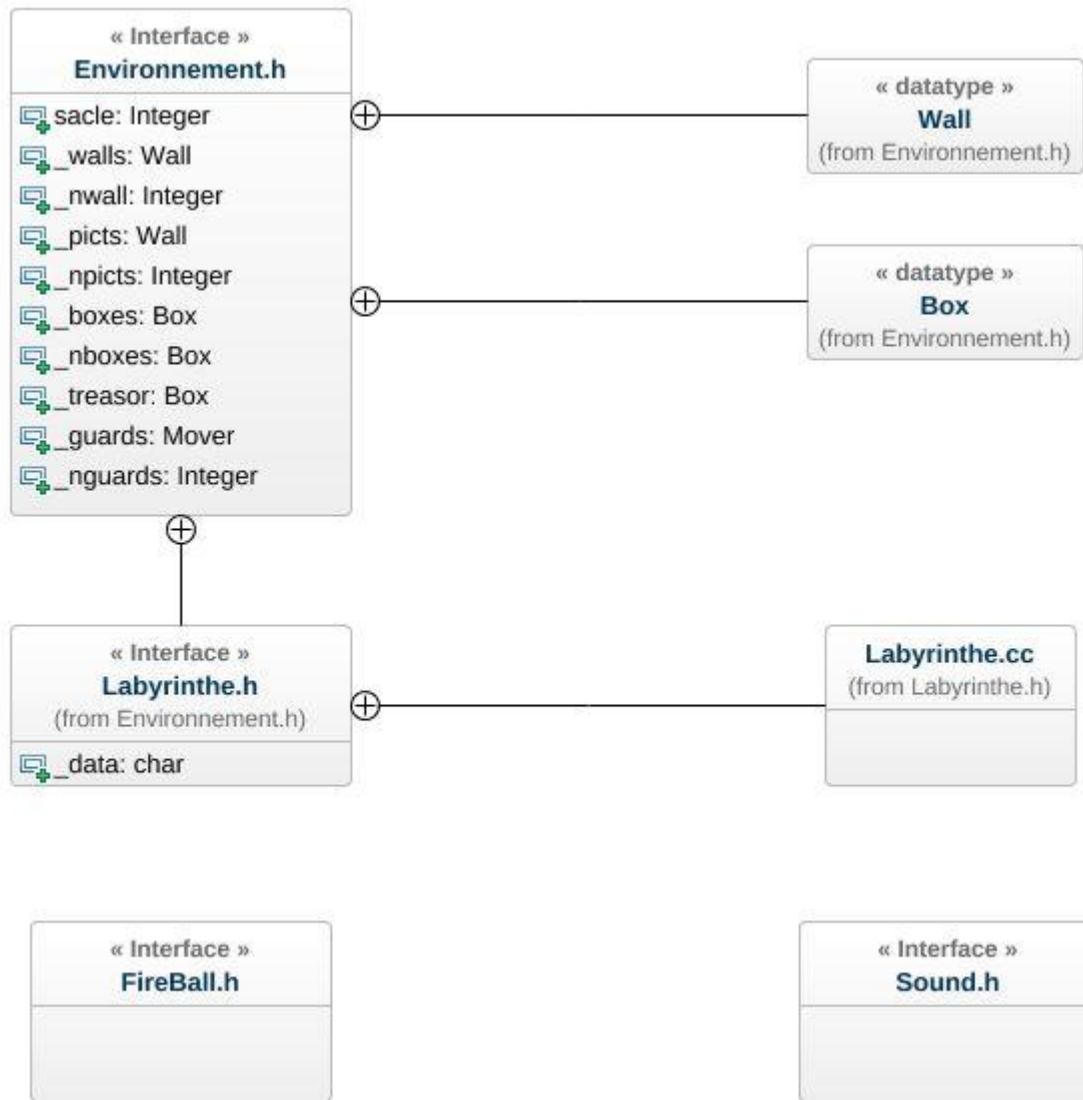
Le mode client n'a pas été réalisé.

Développement

Structure de programme

```
POA-projet
├── README.md
├── Sockets // définir le protocole et le thread
└── labh-proto-fltk // réaliser le jeu
    ├── Chasseur.cc // code source de chasseur
    ├── Chasseur.h // fichier d'entête de chasseur
    ├── Environnement.h // fichier d'entête de environnement
    ├── FireBall.h // fichier d'entête de boule de feu
    ├── Gardien.cc // code source de gardien
    ├── Gardien.h // fichier d'entête de gardien
    ├── Labyrinthe.cc // code source de labyrinthe
    ├── Labyrinthe.h // fichier d'entête de labyrinthe
    ├── Mover.h // fichier d'entête de personnage
    ├── Sound.h // fichier d'entête de son
    ├── labyrinthe.txt // carte de labyrinthe
    ├── modeles // répertoire des modèles
    ├── sons // répertoire des sons
    └── textures // répertoire des textures
```

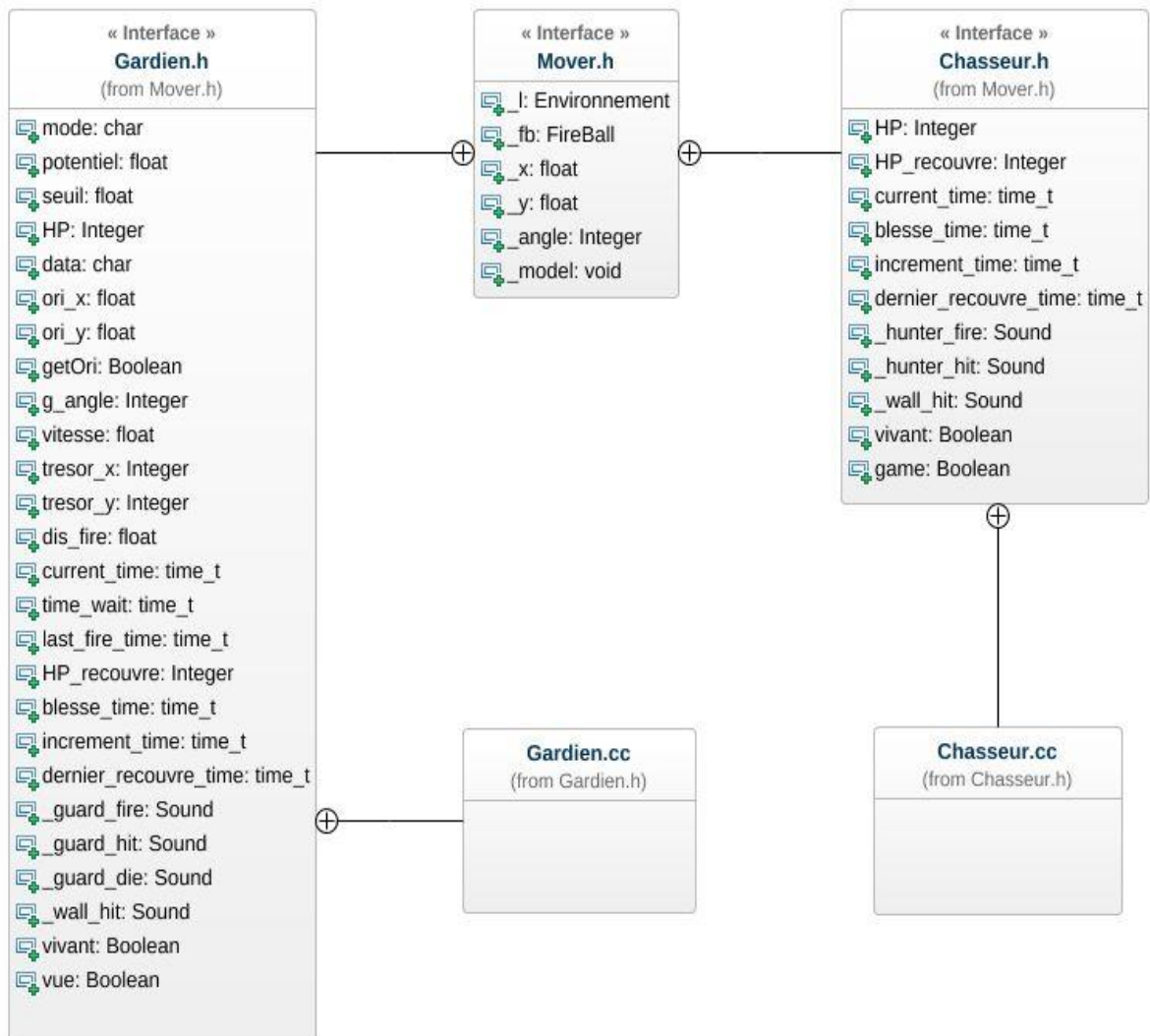
Hiérarchie des classes



Environnement.h définit deux types de donnée : Wall et Box.

Labyrinthe.h étend Environnement.h.

Labyrinthe.cc étend Labyrinthe.h.



Gardien.h étend Mover.h.

Gardien.cc étend Gardien.h.

Chasseur.h étend Mover.h.

Chasseur.cc étend Chasseur.h.

Méthodes et algorithmes

Labyrinthe.h & Labyrinthe.cc

Le labyrinthe est défini par 200x200.

```
#define LAB_WIDTH  200
#define LAB_HEIGHT 200
```

getData() lit le fichier labyrinthe.txt et remplit la matrice `_data` en même temps. Si on rencontre une ligne vide ou une ligne contient la caractère '#', on passe dans la ligne suivante. Sinon, on met cette ligne dans une ligne de `_data`. Pour les cases non-utilisées de `_data`, on met `EMPTY`.

```
while(getline(ReadFile,tmp,'\n'))
    if (tmp==" " || tmp[0]=='#' || tmp.find('#')!=string::npos)
        continue; // ne pas voir cette ligne (vide ou commentaire)
    for (int j=0; j<LAB_HEIGHT; j++)
        if (j<tmp.size()) {
            _data[ligne][j] = tmp[j];
        } else {
            _data[ligne][j] = EMPTY;
        }
        ligne++;
    for (int i=ligne; i<LAB_WIDTH; i++)
        for (int j=0; j<LAB_HEIGHT; j++)
            _data[i][j] = EMPTY;
```

Une fois qu'on a rempli `_data`, on crée le labyrinthe selon `_data`.

createWallHorizontal() crée un mur horizontal.

createWallVertical() crée un mur vertical

createWallCroix() vérifie la case en droite de '+' et la case en bas de '+' pour décider de mettre un mur horizontal ou un mur vertical dans sa position.

```
if (x+1<LAB_WIDTH && (_data[x+1][y]=='+' || _data[x+1][y]=='|'))
    createWallVertical(x, y, walls, nwall);
if (y+1<LAB_HEIGHT && (_data[x][y+1]=='+' || _data[x][y+1]=='-'))
    createWallHorizontal(x, y, walls, nwall);
```

createChasseur() crée un chasseur et il doit toujours être le premier élément dans `_guards`.

createGuard() crée un gardien. Le modèle est choisi au hasard.

```
string modeles[8] = {"Blade", "drfreak", "garde", "Lezard",
                    "Marvin", "Potator", "Samourai", "Serpent"};
guards[*nguards] = new Gardien(1, modeles[rand()%8].data());
```

createBox() crée une caisse.

createAffiche() vérifie aussi la case de droite et la case en bas pour savoir que l'affiche doit mettre entre deux murs horizontal ou vertical.

Chasseur.h & Chasseur.cc

On définit le capital maximum d'un chasseur est 100.

```
#define HP_MAX 100
```

Chasseur::Chasseur() est la méthode principale de la classe. Le capital de chasseur est incrémenté quand il reste un certain temps sans subir de blessure. `HP_recouvre` définit le nombre de HP recouvré par seconde. `blesse_time`, `increment_time` et `dernier_recouvre_time` sont notre compte-temps. `dernier_recouvre_time` est toujours égal `blesse_time + increment_time`.

Chasseur::recouvre() est la fonction pour calculer le HP recouvré de chasseur. Comme on n'a pas de fonction `update()` pour chasseur. On utilise donc cette fonction dans `update()` de la classe Gardien.

```

current_time = time(0);
if (current_time - blesse_time >= increment_time)
    int HP_ajout = (current_time - dernier_recouvre_time)*HP_recouvre;
    if (HP + HP_ajout >= HP_MAX) {
        HP = HP_MAX;
    } else {
        HP += HP_ajout;
    }
    dernier_recouvre_time = current_time;

```

On récupère un temps du système, on compare si la durée entre ce temps actuel et le temps de dernier blessé est dépassé `increment_time`. Si oui, on ajoute un HP recouvré pour ce chasseur. Le HP ne peut pas dépasser au maximum.

Chasseur::move_aux() juge si le chasseur peut faire un mouvement. Elle voit aussi si le chasseur arrive la position de trésor. Si oui, la partie est fini, le joueur est gagné et une fonction

Chasseur::game_over() change l'état des gardiens à mort. Tous les personnages dans le labyrinthe ne peuvent plus bouger sauf le chasseur.

```

for (int i=1; i<(_l->_nguards); i++)
    (((Gardien*)(_l -> _guards[i]))) -> vivant = false;

```

Chasseur::blesse() recalcule le HP de chasseur quand il est frappé par une boule de feu. Cette fonction est appelée dans la fonction

Gardien::process_fireball() et elle vérifie aussi si le HP de chasseur est inférieur ou égal 0. Si oui, une variable publique `vivant` est devient `false`. La fonction `update()` de gardien appelle **Chasseur::game_over()** mais avec la gaine de gardien. Les gardiens dansent et le chasseur ne peut plus bouger.

```

blesse_time = time(0);
dernier_recouvre_time = blesse_time + increment_time;
HP -= 20;
if (HP <= 0)
    vivant = false;

```

Chasseur::process_fireball() contrôle le mouvement de boule de feu. Avec une boucle, on obtient la position de tous les gardiens. Ensuite, on vérifie que si la boule va toucher un parmi eux. Si oui, elle rappelle la fonction **blesse()** de ce gardien. Et si le gardien est mort, on demande qu'il reste au sol.

```
for (int i=1; i<(_l->_nguards); i++)
    int gardien_x = (int)((_l -> _guards[i] -> _x) / Environnement::scale);
    int gardien_y = (int)((_l -> _guards[i] -> _y) / Environnement::scale);

    if (gardien_x == fb_x && gardien_y == fb_y)
        ((Gardien*)(_l -> _guards[i])) -> blesse();
        if (! (((Gardien*)(_l -> _guards[i])) -> vivant)) {
            ((Gardien*)(_l -> _guards[i])) -> rester_au_sol();
        }
    return false;
```

Gardien.h & Gardien.cc

Gardien::Gardien() est la méthode principale. Elle initialise des variables. La vitesse, la distance de tir et le seuil sont les variables aléatoires dans une plage. Le capital recouvré de gardien est 1 HP par seconde. Quand le gardien attaque le chasseur, il a un temps d'attente dans deux tirs successifs.

Gardien::blesse() est comme la fonction **blesse()** de chasseur. Elle est utilisée quand la boule touche le gardien.

Gardien::recouvre() est la fonction pour calculer le HP recouvré de gardien. Lorsque le gardien est vivant, on appelle la fonction dans **update()**.

Gardien::move() pareille que **move_aux()** de chasseur.

Gardien::potentiel_protection() calcule le potentiel de protection. On récupère d'abord la position de chasseur et de trésor. Ensuite, on calcule la distance entre gardien et chasseur, la distance entre gardien et trésor, la distance entre chasseur et trésor.

```
float dis_gc = distance(chasseur_x/scale, chasseur_y/scale, _x/scale, _y/scale);  
float dis_gt = distance(_x/scale, _y/scale, tresor_x, tresor_y);  
float dis_ct = distance(chasseur_x/scale, chasseur_y/scale, tresor_x, tresor_y);
```

Si la distance entre gardien et chasseur est inférieure que 30, on commence à incrémenter le potentiel, sinon le potentiel est remis en 0.

```
if (dis_gc<30) {  
    potentiel += (30-dis_gc);  
} else {  
    potentiel = 0;  
}
```

Si la distance entre gardien et trésor est supérieure que la distance entre chasseur et trésor et sa valeur est inférieure 1/2 la distance entre chasseur et trésor, le potentiel ajoute 50.

```
if (dis_gt>dis_ct && dis_gc<dis_ct)  
    potentiel += 50;
```

Enfin, si le gardien a vu le chasseur, le potentiel ajoute 50.

```
if (regarde(chasseur_x, chasseur_y, atan2(chasseur_y-_y, chasseur_x-_x)))  
    potentiel += 50;
```

Le potentiel est obtenu. Le seuil est un chiffre aléatoire entre 45 et 55. Si le potentiel est supérieur que le seuil, le gardien est en mode attaque, sinon il est en mode défense.

Gardien::fire(int angle_vertical) Le gardien a une probabilité de manquer sa cible. Cette probabilité dépend de l'état de santé du gardien : moins il a de points de vie, plus il tire mal. On définit une plage de -5 à 5 degrés. C'est-à-dire le gardien peut avoir une erreur de 5 degrés quand il a le HP maximum. Cette erreur augmente avec son HP diminue.

```
int angle_fire = 90-g_angle;  
srand(time(NULL));  
angle_fire += (int)(100/HP)*(rand()%10-5);
```

Gardien::process_fireball() La même chose que chasseur. Il vérifie si la boule touche le chasseur et si le chasseur est encore vivant.

```
if (chasseur_x == fb_x && chasseur_y == fb_y) {
    ((Chasseur*)(_l -> _guards[0])) -> blesse();
    if (! (((Chasseur*)(_l -> _guards[0])) -> vivant)) {
        message(" You are die. Good Game ");
        ((Chasseur*)(_l -> _guards[0])) -> game_over();
    }
    return false;
}
```

Gardien::regarde () Une fonction nous dit si le gardien a vu le chasseur. On a la position de chasseur et le radian de gardien face à chasseur. On fait une boucle pour voir dans tous les pas de gardien, s'il existe un mur. Il faut donc marcher tous les murs avec une boucle pour obtenir ses positions. Si on rencontre un mur, le gardien ne peut pas voir le chasseur et la fonction retourne faux, sinon le gardien voit le chasseur et la fonction retourne vrai.

```
float x,y;
x = _x; y = _y;
while (!(int(x/scale)==int(dx/scale) && int(y/scale)==int(dy/scale))) {
    for (int i=0; i<_l->_nwall; i++) {
        if ((_l->_walls[i]._x1 == int(x/scale) && _l->_walls[i]._y1 == int(y/scale)
) || (_l->_walls[i]._x2 == int(x/scale) && _l->_walls[i]._y2 == int(y/scale))) {
            return false;
        }
    }
    x += 0.25*vitesse*scale*cos(radian);
    y += 0.25*vitesse*scale*sin(radian);
}
return true;
```

Gardien::update()

Si le chasseur est vivant et la partie n'est pas terminée, on rappelle la fonction `recouvre()` de chasseur pour mise à jour le HP de chasseur.

```
if (((Chasseur*)(_l -> _guards[0])) -> vivant) {
    if (((Chasseur*)(_l -> _guards[0])) -> game) {
        ((Chasseur*)(_l -> _guards[0])) -> recouvre();
    }
} else {
    _angle += 90; // le gardien dance si chasseur est mort
}
```

On obtient la position initiale de gardien. Elle est utilisée dans le mode défense. Le gardien ne peut pas patrouiller dans tout le labyrinthe. On lui donne une zone de mouvement en fonction de cette position initiale. Il ne sort jamais cette zone.

```
if (!getOri)
    ori_x = _x; ori_y = _y;
getOri = true;
```

Si le gardien est vivant, on appelle `recouvre()` et `potentiel_protection()` pour recouvrer le capital de gardien et calculer son potentiel de protection. Si le potentiel est supérieur que le seuil, le gardien passe en mode attaque, sinon il est en mode défense. S'il change son mode de attaque à défense, on récupère sa position initiale de nouvelle fois.

```
recouvre();
potentiel = potentiel_protection();
if (potentiel < seuil) {
    if (mode == 'a') {
        ori_x = _x; ori_y = _y;
        mode = 'd';
    }
} else {
    mode = 'a';
}
```

En mode défense, le gardien patrouille dans une zone de 16x16. L'angle de mouvement est toujours égal l'angle de face dans le labyrinthe plus 90°. On calcule la position de prochain pas avec Chaque fois quand il rencontre un côté de zone ou un mur ou une caisse, il change sa direction. Son angle change aléatoirement dans une plage de 90° à 125°.

```
float dx,dy;
g_angle = _angle + 90;
dx = 0.25*vitesse*scale*cos(angle_to_radian(g_angle));
dy = 0.25*vitesse*scale*sin(angle_to_radian(g_angle));
if(move(dx,dy) && move(dx, 0.0) && move(0.0,dy) ){
    // move
} else {
    change_direction();
}
if (_x>ori_x+80 || _x<ori_x-80 || _y>ori_y+80 || _y<ori_y-80)
    change_direction();
```

En mode attaque, le gardien cherche le chasseur. Si la distance entre le chasseur et le gardien est inférieur que la distance de tir et le gardien a vu le chasseur, on appelle la fonction fire() pour tirer la boule. Après un tir, on enregistre le temps de système dans la variable last_fire_time pour avoir un temps d'attente devant le prochain tir. Sinon, le gardien bouge vers le chasseur.

```
if (dis_cg <= dis_fire)
    current_time = time(0);
if (current_time - last_fire_time >= time_wait && regarde(chasseur_x,
    chasseur_y,radian)) {
    fire(0);
    last_fire_time = current_time;
}
```

Si le gardien est mort, il reste au sol.

```
if (vivant) {
    .....
} else {
    rester_au_sol();
}
```


Problèmes

Récupération de labyrinthe

Pour chaque mouvement de gardien, on veut libérer l'ancienne position de gardien dans la matrice de labyrinthe et mettre à jour la case de nouvelle position. La méthode `_l -> data()` nous permet d'obtenir une case mais il faut qu'on peut aussi faire des modifications sur `_data[][]`. On importe donc la classe labyrinthe dans gardien.

```
#include "Labyrinthe.h"
```

Ensuite, on déclare un pointeur data.

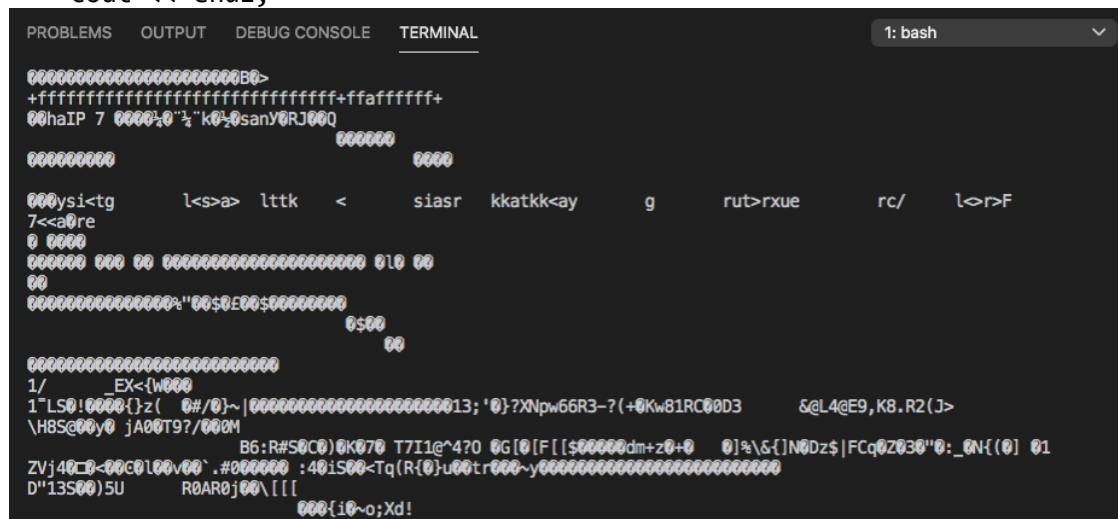
```
char (*data)[200][200];
```

On initialise data dans la méthode principale de gardien.

```
data = &(l -> _data);
```

Cependant, quand on essaie d'afficher data. Elle nous donne n'importe quoi. Par conséquent, notre chasseur peut marcher sur les gardiens.

```
for (int i=0;i<200;i++)
    for (int j=0;j<200;j++)
        cout << *data[i][j];
    cout << endl;
```



Conclusion

Ce projet améliore notre compréhension sur l'énoncé français. Nous lisons et discutons la fonction du jeu et le but du jeu.

Sur les codes, Comme c'est un travail en binôme, nous devons écrire clairement des commentaires sur les codes que nous avons ajouté pour faciliter la compréhension d'une autre personne.

En programmation, nous apprenons le langage C ++ et la programmation orientée objet. On voit comment concevoir et hériter des classes, comment déclarer des variables et des fonctions et comment encapsuler des données et des méthodes.

Enfin, ce projet nous permet de voir comment réaliser un jeu simple en 3D à l'aide de OpenGL.