Projet de programmation C++

Professeur superviseur : Patrick Amar

Par : Shutong ZHENG, Tianyi LI, Lanshi FU

29 décembre 2022

Table des matières

1 Introduction **3**
1.1 Description du projet 3
1.2 Cas d'usage 4

2 Détails d'implémentation **5**
2.1 Flux de programme 5
2.2 Structure des fichiers 5
2.3 Structure des classes 9
2.4 Méthodes et algorithmes 10
2.4.1 Environnement.h 10
2.4.2 Labyrinthe.h & Labyrinthe.cc 11
2.4.3 Gardien.h 13
2.4.4 Chasseur.h & Chasseur.cc 16
2.4.5 Mover.h 19
2.4.6 FireBall.h 20
2.4.7 Sound.h 21
2.5 Modèles de conception 21
2.6 Présentation de FLTK 23
2.7 Présentation d'OpenGL 24

3 Résumé du projet **24**

4 Bibliographie **25**

1 Introduction

1.1 Description du projet

Page Tâches du projet :https ://www.lri.fr/ pa/PROGCXX/laby-hante.html

Ceci est le rapport de projet de cours du Master QDCS, UE : Programmation Orientée Objet, Premier Semestre 2022-2023. Le professeur superviseur est M. Patrick Amar.

L'objectif du projet était de mettre en œuvre une interface 3D pour le jeu "Kill them all" sur la base du prototype fourni. Le jeu se concentre sur un labyrinthe dans lequel le joueur utilise un personnage de chasseur pour chercher un trésor. Le joueur attaque ou se déplace pour détruire ou éviter les gardes, et il y a des obstacles et d'autres mécanismes dans le labyrinthe. Pour gagner la partie, le joueur doit contrôler le chasseur et réussir à trouver le trésor. La partie est perdue si les HP du joueur sont $<= 0$.
Certains modèles de classe C++ et certaines ressources statiques de modèles 3D et de sons ont été fournis par le cadre de programmation, et l'API pour l'interface graphique est également fournie.

Les objectifs de formation 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.

1.2 Cas d'usage

Selon les exigences du projet, ce programme doit mettre en œuvre les fonctions suivantes

Chasseur :
Le joueur contrôle le chasseur en utilisant la souris et le clavier pour se déplacer dans le labyrinthe ou pour lancer une attaque. L'attaque du chasseur consiste à lancer des boules de feu qui explosent lorsqu'elles touchent des murs, des obstacles ou des gardiens. Le chasseur a des HP, qui commencent le jeu avec une bonne quantité de HP et qui diminuent lorsqu'il est attaqué par un défenseur et touché. Lorsque le chasseur ne subit pas d'attaque pendant un certain temps, ses HP sont automatiquement restaurés.

Les gardiens :
Les gardiens patrouillent dans le labyrinthe et lorsqu'ils repèrent un chasseur, le gardien l'attaque activement et se rapproche de lui. Le Guardian a deux états. Le premier est le mode patrouille, dans lequel les gardes se déplacent en ligne droite dans une direction aléatoire jusqu'à ce qu'ils rencontrent un obstacle, après quoi ils choisissent une nouvelle direction au hasard. L'autre état est le mode attaque, dans lequel lorsque les gardes "voient" un chasseur dans un certain rayon, ils passent en mode attaque et commencent à attaquer activement le chasseur, mais leurs attaques sont paramétrées pour être imprécises.

Labyrinthe :
Le constructeur de la classe Labyrinthe servira à initialiser un labyrinthe selon la représentation interne du programme de jeu à partir d'une représentation textuelle du labyrinthe (à l'aide des caractères + - |). Conceptuellement, le labyrinthe sera rendu en 3D. Le joueur se déplace sur une surface plane. Les murs du labyrinthe sont droits et horizontaux ou verticaux. Le labyrinthe est fermé et il y a trois endroits importants dans le labyrinthe :
1. l'endroit où le chasseur est né.
2. l'endroit où se trouve le trésor.
3. le portail.
Des obstacles seront placés dans le labyrinthe. Les murs du labyrinthe seront décorés de photos. Le programme sera capable de lire le plan du labyrinthe à partir d'un fichier texte et de générer le labyrinthe.

2 Détails d'implémentation

2.1 Flux de programme

Pour satisfaire le cas d'utilisation, la procédure suivra le flux suivant :

1. Liser un fichier qui décrit toutes les informations du labyrinthe.
2. Rendu des modèles graphiques, puis entrée dans la boucle principale du jeu.
3. Dans chaque boucle, les personnages vont agir en suivant les comportements que nous avons définis en code.
4. Dans chaque boucle, nous allons détecter la condition pour terminer le jeu, gagner ou perdre.

2.2 Structure des fichiers

Ce projet s'accorde sur le format des caractères utilisés pour représenter le labyrinthe dans le fichier txt :

"+" "-" "|" pour les murs
"C" pour chasseur
"G" pour chaque tuteur
"T" pour trésor
"X" pour chaque caisse
"a" "b" pour l'affiche

L'affiche sera identifiée par la lettre (minuscule) qui la représente.
Le format de fichier sonore utilisé pour ce projet est le .wav.
Les fichiers de modèles 3D utilisés dans ce projet sont aux formats .jpg et .md2.
Les fichiers de texture utilisés dans ce projet sont au format .jpg , .gif et .tga.

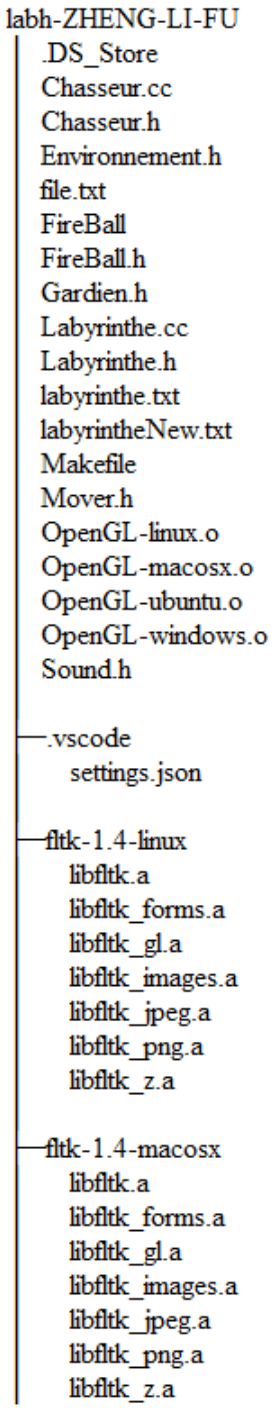

FIGURE 1 – File list 1

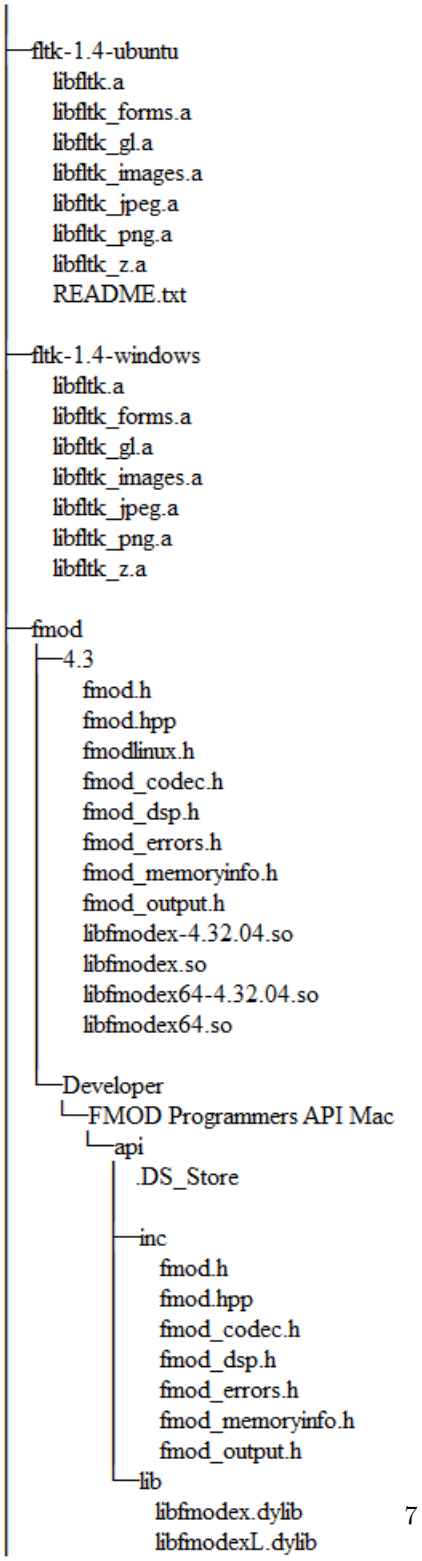

FIGURE 2 – File list 2

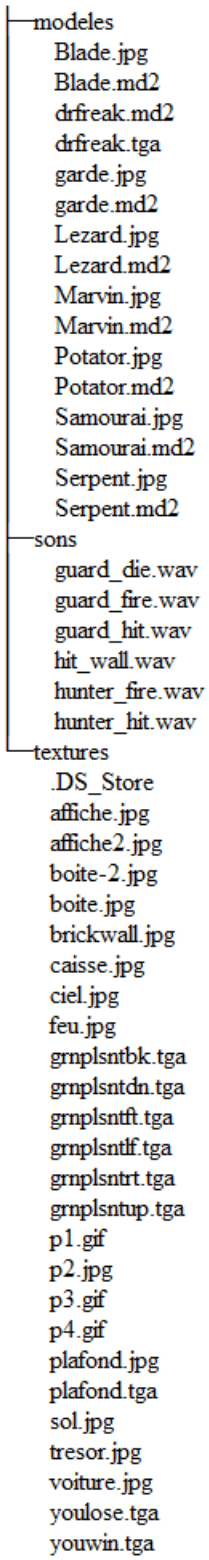

FIGURE 3 – File list 3

2.3 Structure des classes

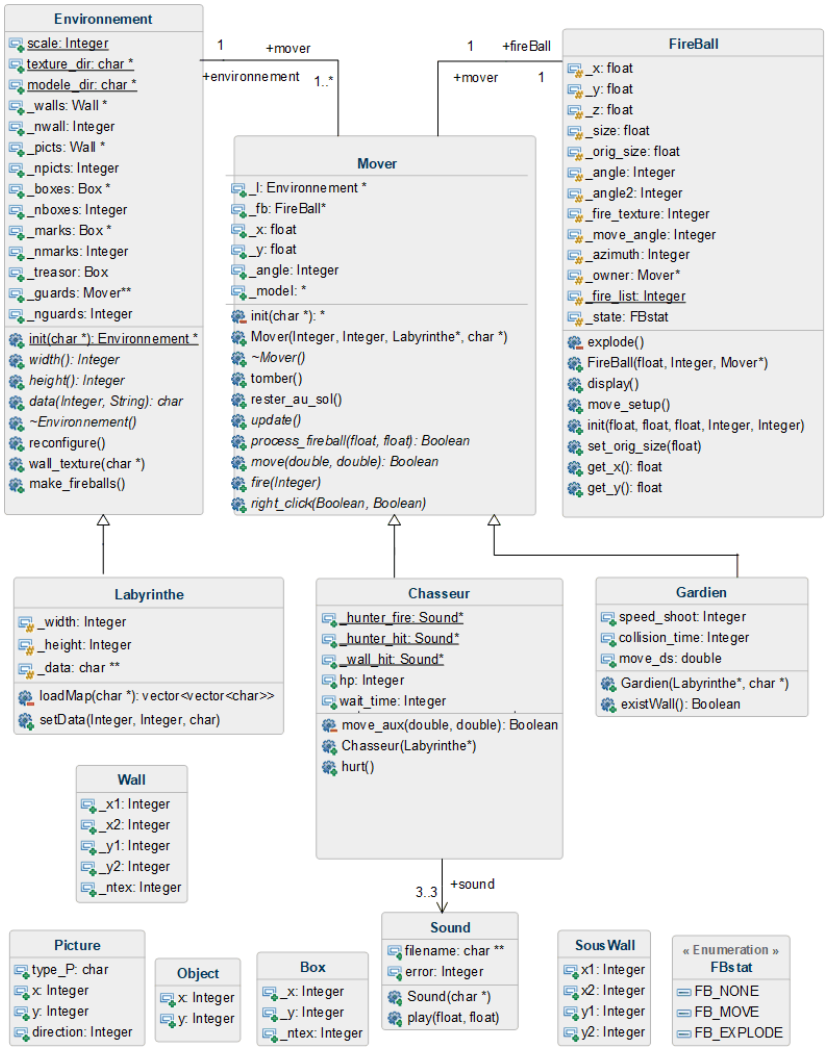

FIGURE 4 – UML

2.4 Méthodes et algorithmes

2.4.1 Environnement.h

La classe *Environnement* est la classe parente du *Labyrinthe*.

```
class Environnement {
public:
static const int scale;
static const char* texture_dir;
static const char* modele_dir;
static Environnement* init (char* filename);
//...
```

Environnement.h

scale est un facteur d'echelle, *texture_dir* est la rapertoire des textures. *modele_dir* est la rapertoire des personnages.

```
//...
Wall* _walls;   // les murs sous forme de tableau de segments.
int   _nwall;   // le nombre de segments.
Wall* _picts;   // les affiches.
int   _npicts;  // leur nombre.
Box*  _boxes;   // les caisses.
int   _nboxes;  // leur nombre.
Box*  _marks;   // les marques au sol.
int   _nmarks;  // leur nombre.
Box   _treasor; // le tresor.
Mover** _guards;  // les gardes (tableau de pointeurs de gardiens).
int   _nguards; // leur nombre.
//...
```

Environnement.h

Cette série de déclarations déclare les objets de divers éléments de jeu et leurs quantités.

```
//...
virtual int width () =0;// retourne la largeur du labyrinthe.
virtual int height () =0;// retourne la longueur du labyrinthe.
virtual char data (int i, int j) =0;// retourne le cas (i, j).
virtual ~Environnement () {}
//fonction a n'appeller QUE si les murs ou les caisses ou le tresor
    BOUGENT
void reconfigure ();
// retourne le numero de texture pour le fichier 1mage passe en
   argument.
int wall_texture (char*);
// recrae les boules de feu des movers, fonction a n'appeller QUE
   si des gardiens sont recruis.
void make_fireballs (void); };
```

Environnement.h

Une série de fonctions virtuelles sont définies pour permettre aux programmes d'appeler des fonctions de même nom de la classe *Labyrinthe* avec des pointeurs vers la classe *Environnement*. L'appel à une fonction virtuelle n'est pas

déterminé au moment de la compilation mais au moment de l'exécution, d'où l'utilisation de fonctions virtuelles pour obtenir l'effet du polymorphisme.

2.4.2 Labyrinthe.h & Labyrinthe.cc

```
1
class Labyrinthe : public Environnement {
3 private:
int    _width;
5 int    _height;
char**  _data;
7 vector<vector<char > > loadMap(char * filename);
public:
9 Labyrinthe (char*);
int width () { return _width;}
11 int height () { return _height;}
char data (int i, int j){return _data [i][j];}
13 };
```

Labyrinthe.h

La classe "Labyrinthe" hérite de la classe "Environnement". Sa fonction est de lire le fichier txt, d'obtenir le plan du labyrinthe selon le format. Ensuite, il génère des structures de tous les éléments en fonction des données lues et charge ensuite les ressources .
La fonction *loadMap* lit un fichier de carte à terminaison .txt sur le disque dur et renvoie un tableau vectoriel de deux bits de la classe char.

Constructeur

```
1 Labyrinthe::Labyrinthe(char *filename){
vector<vector<char> > map = this->loadMap(filename);
3 _width = map.size();
_height = getMapWidth(map);
5 vector<Picture> pictures = getPictures(map);
vector<Object> boxes = getObjects('x', map);
7 vector<Object> guards = getObjects('G', map);
vector<Object> treasures = getObjects('T', map);
9 vector<Object> player = getObjects('C',map);
vector<SousWall> walls =  getWalls(map);
11
_nwall = walls.size();
13  _walls = new Wall [_nwall];
_npicts = pictures.size();
15 _picts = new Wall [_npicts];
_nboxes = boxes.size();
17 _boxes = new Box [_nboxes];
//...etc...
19 }
```

Labyrinthe.cc

Le constructeur initialise une instance de la classe Labyrinthe en appelant la fonction *loadMap* avec un pointeur vers un tableau de chars en entrée, qui lit le fichier de mise en page txt au format convenu.
Cette instance va générer les différents structures du labyrinthe à l'aide de la série de méthodes get en fonction de la disposition du dessin dans le fichier txt. En raison de la taille limitée de la carte, nous chargerons les ressources de la carte en une seule fois. Si la carte est trop grande, ou si la charge de la cartographie et du chargement du modèle est élevée, nous pouvons également utiliser le matrix chunking pour charger la carte par sections.

```
//...
struct Picture{
char type_P;
int x ;
int y;
int direction; };

struct Object{
int x ;
int y;};

struct SousWall{
int x1;
int y1;
int x2;
int y2;};

int getMapWidth(vector<vector<char> > map);
vector<Picture>  getPictures(vector<vector<char> > map );
vector<Object> getObjects(char obj, vector<vector<char> > map);
vector<SousWall> getWalls(vector<vector<char> > map);
```

Labyrinthe.h

Le vecteur *Object* est le conteneur générique des structures du jeu dans la carte (obstacles, trésors, joueurs, gardesportails).La structure *Objet* est utilisée pour indiquer l'emplacement des Objects.
Le vecteur *SousWall* et le vecteur *Picture* sont les conteneurs des structures de la carte (murs, images).La structure *SousWall* est utilisée pour stocker les positions de départ et d'arrivée de la clôture. La structure *Image* pour sauver les peintures murales.
La function *getMapWidth* retourne la largeur.
La funtion *getPictures* retourne un vecteur de conteneurs Picture.
La funtion *getObjects* retourne un vecteur de conteneurs Object.
La function *getWalls* retourne un vecteur de conteneurs SousWall.

Le reste de Labyrinthe.cc est une implémentation concrète du chargement des ressources.Cela comprend huit monstres gardiens différents, deux obstacles différents, ainsi que des images et des marqueurs différents. Pour une explication de ce code, qui sera expliquée par des commentaires dans le code, veuillez lire le code.

2.4.3 Gardien.h

Voici le code de la classe "Gardien", qui hérite de la classe "Mover". Le constructeur est unique et requiert un pointeur sur char et un pointeur sur un objet Labyrinthe en entrée.

```
class Gardien : public Mover {
public:
int speed_shoot;
int collision_time ;
double move_ds ;
//...
```

Gardien.h

La variable *speed_shoot* est la vitesse de tir des gardes.
La variable *collision_time* est le temp de perforation.
La variable *move_ds* est la vitesse de déplacement des gardes.

```
//...
Gardien (Labyrinthe* l, const char* modele) : Mover (120, 80, l,
    modele){
speed_shoot = 0;
collision_time = 0;
move_ds = 0.1;}
//...
```

Constructeur

Le constructeur du Gardien initialise la cadence de tir du garde, le temps de charge, et la vitesse de déplacement du garde.

update() est la fonction membre la plus critique, qui garantit que l'état de la garde ainsi que l'état du joueur peuvent être mis à jour à temps au fur et à mesure de la progression du jeu. Le garde détecte les joueurs ainsi que les obstacles en temps réel et déclenche la mise à jour de l'état approprié : cela comprend le passage du garde entre les états d'attaque et de patrouille, le comportement d'attaque du garde, l'incapacité du garde à repérer les obstacles ou les joueurs hors de portée de détection. Et, les joueurs pressent pour récupérer des points de vie lorsqu'ils ne sont pas attaqués.

```
//...
if(collision_time == 3){
move_ds = 0 - move_ds;
collision_time = 0;}
if(!move(move_ds,move_ds))
{collision_time ++;}
//...
```

void update (void)

Si le garde heurte le mur 3 fois, le garde se déplace dans la direction opposée.

```
//...
int x_player = _l->_guards[0]->_x;
int y_player = _l->_guards[0]->_y;
int x_enemy = _x;
int y_enemy = _y;
float dis_y = (float)abs(y_player - y_enemy);
float dis_x = (float)abs(x_player - x_enemy);
float dis_total = dis_y*dis_y + dis_x*dis_x ;
//...
```

void update (void)

Calcul de la distance entre le joueur et les gardes.

```
//...
((Chasseur*)_l->_guards[0])->wait_time --;
if (((Chasseur*)_l->_guards[0])->wait_time < 0){
((Chasseur*)_l->_guards[0])->wait_time = 0;}

if(((Chasseur*)_l->_guards[0])->wait_time == 0 && ((Chasseur*)_l->
    _guards[0])->hp != 100){
((Chasseur*)_l->_guards[0])->hp ++;
message("Hunter HP : %d",((Chasseur*)_l->_guards[0])->hp);}
//...
```

void update (void)

Le temp d'attente du joueur−1, si le temps d'attente du joueur est négatif, on le laisse à 0.Si le joueur attend 0 et que la hp n'est pas égale à 100, la récupération du sang commence.

```
//...
if(dis_total <= 25000 && !existWall()){
speed_shoot ++;
int angle = (int)(atan(dis_y/dis_x)*180/PI) ;

//L'ennemi se retourne
if (y_enemy == y_player && x_player > x_enemy)
_angle = 270;
else if (y_enemy == y_player && x_player < x_enemy)
_angle = 90;
else if (x_player == x_enemy && y_player > y_enemy)
_angle = 0;
else if (x_player == x_enemy && y_player < y_enemy)
_angle = 180;
else if (x_player > x_enemy &&  y_player < y_enemy)
_angle = 270 -angle;
else if (x_player < x_enemy && y_player < y_enemy)
_angle = 90+angle;
else if (x_player < x_enemy && y_player > y_enemy)
_angle= 90-angle;
else if (x_player > x_enemy && y_player > y_enemy)
_angle= 270+angle;
if(speed_shoot == 100){fire(0);speed_shoot = 0;}}
```

void update (void)

Si la distance entre le joueur et le garde est <= 25000 et qu'il n'y a pas de mur entre les deux. le joueur sera repéré et attaqué par l'ennemi. Les gardes tourneront en fonction de la position du joueur par rapport à elle.

```
bool move (double dx, double dy) {
int old_x = _x;
int old_y = _y;

if (EMPTY == _l -> data ((int)((_x + dx) / Environnement::scale),
(int)((_y + dy) / Environnement::scale))||((int)old_x/Environnement
    ::scale ==(int)
((_x+dx) / Environnement::scale)  && (int)old_y/Environnement::
    scale ==(int)((_y+dy) / Environnement::scale) ) ){
if(_l->data((int)(old_x / Environnement::scale) , (int)(old_y /
    Environnement::scale) ) == 1){
((Labyrinthe*)_l)->setData((int)(old_x / Environnement::scale),(int
    )(old_y / Environnement::scale),EMPTY);}

_x += dx;
_y += dy;

((Labyrinthe*)_l)->setData((int)(_x / Environnement::scale),(int)(
    _y / Environnement::scale),1);
return true;}
return false;}
```

bool move (double dx double dy)

La fonction *move* permet aux gardes de patrouiller en zigzag. Les coordonnées planes des gardes sont entrées comme arguments de cette fonction. Une fois qu'ils ont fini de patrouiller en ligne droite, ils se retournent et patrouillent à nouveau en ligne droite, réalisant ainsi une patrouille en zigzag.

```
void fire (int angle_vertical) {
int angle_H = 360-_angle+random()%5+;
if (angle_H <0 ){angle_H = 0;}
_fb -> init ( _x, _y, 10.,angle_vertical, angle_H);}
```

void fire (int angle_vertical)

La fonction *fire* indique aux gardes de faire feu. Leurs tirs ne sont pas précis et produisent une déviation aléatoire de 5 degrés.

```
//...
if (EMPTY == _l -> data ((int)((_fb -> get_x () + dx) /
    Environnement::scale),
(int)((_fb -> get_y () + dy) / Environnement::scale))){
message ("Hunter HP : %d  Woooshh ..... %d\n", hp,(int) dist2);
return true;}
//...
if((int)((_fb -> get_x () + dx) / Environnement::scale)  == (int)(
    _l->_guards [i] -> _x / Environnement::scale)
&&(int)((_fb -> get_y () + dy) / Environnement::scale) ==(int)(_l->
    _guards [i] -> _y / Environnement::scale))
_l->_guards[i]->tomber();
//...
```

bool process_fireball (float dx float dy)

La fonction *process_fireball* est le mouvement de la boule de feu. Si la boule de feu tirée par le garde touche le joueur, la fonction *hurt* déduira les HP du joueur. Pour plus de détails, veuillez lire le code.

```
bool existWall(){
int x_player =  (int)(_l->_guards[0] ->_x/Environnement::scale);
int y_player = (int)(_l->_guards[0] -> _y/Environnement::scale);
int x_gardien = (int)(this->_x/Environnement::scale);
int y_gardien = (int)(this->_y/Environnement::scale);
for( int i = min(x_player,x_gardien); i<= max(x_player, x_gardien)
    ;i++ )
for (int j = min(y_player,y_gardien); j <= max(y_player,y_gardien);
     j++){
if((i== x_player && j == y_player) || (i==x_gardien && j ==
    y_gardien)){continue;}
if(_l->data( i, j) == 1){return true;}}
return false;}
```

bool existWall()

La fonction *existWall* détermine s'il existe un mur entre le joueur et le garde.

2.4.4 Chasseur.h & Chasseur.cc

```
class Chasseur : public Mover {
private:
bool move_aux (double dx, double dy);
public:
static Sound* _hunter_fire;// bruit de l'arme du chasseur.
static Sound* _hunter_hit;// cri du chasseur touche.
static Sound* _wall_hit;// on a tape un mur.
int hp;
int wait_time;
 //...etc...
```

Chasseur.h

La classe "Chasseur" hérite de la classe "Mover" et implémente divers événements pour le personnage du joueur : le joueur peut se déplacer et interagir

avec les éléments du jeu dans la carte, attaquer sur commande de la souris et émettre des sons lorsqu'il attaque, le joueur perd des HP lorsqu'il est attaqué et déclenche la fin du jeu lorsque HP=0.

```
Chasseur::Chasseur (Labyrinthe* l) : Mover (100, 80, l, 0){
hp=100;
_hunter_fire = new Sound ("sons/hunter_fire.wav");
_hunter_hit = new Sound ("sons/hunter_hit.wav");
if (_wall_hit == 0)
{_wall_hit = new Sound ("sons/hit_wall.wav");}
message("Hunter HP : %d",this->hp);
wait_time = 300;}
```

Chasseur.cc

Le constructeur initialise les HP du personnage du joueur à 100 et le temps d'attente à 300.

```
bool Chasseur::move_aux (double dx, double dy){
if (EMPTY == _l -> data ((int)((_x + dx) / Environnement::scale),
(int)((_y + dy) / Environnement::scale))){
_x += dx;
_y += dy;
if((_l->_treasor._x == (int)_x/Environnement::scale
&& _l->_treasor._y == (int)_y/Environnement::scale+1)
||(_l->_treasor._x == (int)_x/Environnement::scale
&& _l->_treasor._y == (int)_y/Environnement::scale-1)){
partie_terminee(true);}

if((int) (_x/Environnement::scale) == _l->_marks[0]._x
&& (int) (_y/Environnement::scale) == _l->_marks[0]._y){
_x = _l->_marks[1]._x * Environnement::scale ;
_y = _l->_marks[1]._y * Environnement::scale ;
cout <<"here 1 "<<endl;}
else if ((int) (_x/Environnement::scale) == _l->_marks[1]._x
&& (int) (_y/Environnement::scale) == _l->_marks[1]._y){
_x = _l->_marks[0]._x * Environnement::scale ;
_y = _l->_marks[0]._y * Environnement::scale ;
cout<< "here 2 "<<endl;}
return true;}
return false;}
```

bool Chasseur : :move_aux (double dx double dy)

La fonction *move_aux* contrôle les événements de mouvement du chasseur. La position horizontale du joueur est transmise à cette fonction comme argument, et si le joueur atteint l'emplacement du trésor, il gagne et reçoit l'écran gagnant. Si le joueur se déplace vers un portail, il est transporté vers un autre portail.

```
bool Chasseur::process_fireball (float dx, float dy){
float x = (_x - _fb -> get_x ()) / Environnement::scale;
float y = (_y - _fb -> get_y ()) / Environnement::scale;
float dist2 = x*x + y*y;
if (EMPTY == _l -> data ((int)((_fb -> get_x () + dx) /
    Environnement::scale),
(int)((_fb -> get_y () + dy) / Environnement::scale))){
message ("Hunter HP : %d  Woooshh ..... %d\n", hp,(int) dist2);
return true;}
for(int i = 1 ; i <_l->_nguards; i++){if(
(int)((_fb -> get_x () + dx) / Environnement::scale)
== (int)(_l->_guards [i] -> _x / Environnement::scale)
&&(int)((_fb -> get_y () + dy) / Environnement::scale)
==(int)(_l->_guards [i] -> _y / Environnement::scale)){_l->_guards[
    i]->tomber();}}
float dmax2 = (_l -> width ())*(_l -> width ()) + (_l -> height ())
    *(_l -> height ());
_wall_hit -> play (1. - dist2/dmax2);
message ("Hunter HP : %d  Booom...\n",this->hp);
return false;
}
```

bool Chasseur : :process_fireball (float dx

La fonction *process_fireball* contrôle l'état de la boule de feu et la position de la boule de feu est passée comme argument à cette fonction.Il calculer la distance entre le chasseur et le lieu de l'explosion. Si la boule de feu touche le garde, ce dernier tombera.
Il calculer la distance maximum en ligne droite.faire exploser la boule de feu avec un bruit fonction de la distance.

```
void Chasseur::fire (int angle_vertical){
message ("Hunter HP : %d  Woooshh...",this->hp);
_hunter_fire -> play ();
_fb -> init ( _x, _y, 10., angle_vertical, _angle);}
```

void Chasseur : :fire (int angle$_v ertical$)

La fonction $fire$ met en œuvre le comportement de tir du joueur en définissant la position 3D initiale de la boule de feu et en utilisant les angles horizontal et vertical du tir comme paramètres.

```
void Chasseur::hurt(){
this->hp  = this->hp -5;
message("Hunter HP : %d",this->hp);
if(this->hp == 0)
partie_terminee(false);
wait_time = 300*(_l->_nguards-1);}
```

void Chasseur : :hurt()

La fonction *hurt* implémente la réduction des HP lorsque le joueur est touché par l'attaque d'un garde. Le joueur subit des dégâts de boule de feu, puis hp-5 Si hp=0, le jeu est terminé.

2.4.5 Mover.h

```
void Chasseur::hurt(){
class Mover {
private:
  static void*  init (const char*);
public:
  Environnement*  _l;
  FireBall* _fb;
  float    _x, _y;
  int      _angle;
  void*    _model;    }
```

Mover.h

La classe *Mover* est la classe de base pour *Chasseur* et *Gardien*. La fonction init est utilisée pour initialiser le modèle 3D au format md2. La variable $\_l$ représente la carte du labyrinthe, la variable $\_fb$ représente la boule de feu, $\_x$ et $\_y$ représentent la position plane de l'objet, et $\_angle$ représente l'angle auquel l'objet se déplace ou tire.

```
Mover (int x, int y, Labyrinthe* l, const char* modele) :
_l ((Environnement*)l), _fb (0), _x ((float)x)
, _y((float)y),_angle (0)
{ _model = init (modele); }
virtual ~Mover () {}
```

Constructeur

Le constructeur initialise ici la position de maintenance de l'objet mover et charge les ressources du modèle.

```
//...etc...
void tomber ();
void rester_au_sol ();
//Permet    personnage (Guardien) de mettre    jour le statut et d'
    ex cuter les    vnements    en temps voulu.
virtual void update (void) =0;

// fait bouger la boule de feu du personnage.
virtual bool process_fireball (float dx, float dy) =0;

// tente de deplacer le personnage de <dx,dy>.
virtual bool move (double dx, double dy) =0;

 // fait tirer le personnage sur un ennemi (vous pouvez ignorer l'
     angle vertical).
virtual void fire (int angle_vertical) =0;

// appelee pour le gardien 0 (chasseur) quand l'utilisateur fait un
     clic droit;
// shift (control) est vrai si la touche shift (control) est
    appuyee.
virtual void right_click (bool shift, bool control) {}
```

Mover.h

La function *tomber* fait tomber un gardien et se relever.
La function *rester_au_sol* fait tomber un gardien et le laisse au sol.
L'importance du polymorphisme dans le développement orienté objet est à nouveau démontrée par une série de déclarations de fonctions virtuelles.

2.4.6 FireBall.h

La FireBall, un élément clé du programme, possède de nombreuses propriétés uniques qui font qu'il est inapproprié d'hériter du Mover.

```
class FireBall {
private:
float _x, _y, _z;
float _size;
float _orig_size;
int    _angle;
int    _angle2;
unsigned int   _fire_texture;
int    _move_angle;
int    _azimuth;
Mover*  _owner;

static unsigned int _fire_list;
FBstat  _state;
void explode ();
//...
```

FireBall.h

Les variables et fonctions membres privées de la classe Fireball comprennent la position 3D de la boule, la taille de la boule, la taille initiale de la boule, la texture de la boule et l'angle de mouvement de la boule. Sont également inclus les objets dont l'instance de la boule de feu dépend, l'état de la boule de feu et une fonction d'explosion.

```
//...
public:
  FireBall (float size, unsigned int tex, Mover*);
  void display ();
  void move_step ();
  void init (float x, float y, float z, int angle_vertical, int
      angle_horizontal);
  void set_orig_size (float size) { _orig_size = size; }
  float get_x () { return _x; }
  float get_y () { return _y; }
};
```

FireBall.h

Le constructeur de la boule de feu utilise la taille de la boule de feu, un message de la console, et un pointeur vers un objet Mover comme paramètres entrants. En outre, la classe boule de feu déclare un certain nombre de fonctions importantes : initialisation d'un tir, définition de la taille de la boule de feu et lecture de la position actuelle de la boule de feu.

2.4.7 Sound.h

La classe *Sound* fournit une ressource sonore que l'application doit charger. Comme cela nécessite des appels à l'API audio, qui dépend du système d'exploitation, il est nécessaire de vérifier la version actuelle du système.

```
//....Vrifier le type du systme.....
class Sound {
private:
#ifdef _WIN32
char* _sound;
#else
static FMOD::System*  _system;
static FMOD::Channel* _channel;
FMOD::Sound*       _sound;
#endif
static int  _nsounds;
void init (void);
public:
Sound (const char*);
~Sound ();
void play (float volume =1., float pan =0.);
};
```

Sound.h

Le constructeur de la classe Sound charge le fichier audio en utilisant un pointeur de classe char.
Les variables membres de la classe Sound comprennent une variable de comptage, ainsi que des variables spécifiques aux différentes API du système.
Les fonctions membres de la classe Sound comprennent une méthode d'initialisation et une méthode de lecture.

2.5 Modèles de conception

En 1994, Erich Gamma, Richard Helm, Ralph Johnson et John Vlissides ont coécrit un livre intitulé *Design Patterns − Elements of Reusable Object − Oriented Software*[1], qui faisait pour la première fois référence au concept de design patterns dans le développement de logiciels.

L'intérêt d'utiliser des patrons de conception est que, pour que le développement de logiciels soit maintenable et évolutif, il est essentiel de réutiliser autant de code que possible et de réduire le couplage du code. Le modèle de conception est principalement affiné sur la base de la programmation OOP et repose sur les principes suivant :

1.Open Closed Principle : Le principe d'ouverture et de fermeture a été proposé par Bertrand Meyer et signifie que les logiciels doivent être ouverts aux extensions et fermés aux modifications. L'idée ici est que, lors de l'ajout de nouvelles fonctionnalités, il est préférable de laisser le code inchangé si possible, et il est préférable que les nouvelles fonctionnalités soient réalisées en ajoutant

simplement du code.

2.Le principe de substitution de Barbara Liskov : Si nous pouvons appeler une méthode d'une classe parente avec succès, alors la remplacer par un appel d'une sous-classe devrait également fonctionner parfaitement bien.

3.Single Responsibility Principle : Il ne devrait pas y avoir plus d'une raison pour qu'une classe change, c'est-à-dire que chaque classe devrait avoir une responsabilité unique, sinon la classe devrait être divisée.

4.Dependence Inversion Principle : Les modules de niveau supérieur ne doivent pas dépendre des modules sous-jacents ; ils doivent tous dépendre de l'abstraction. Les abstractions ne doivent pas dépendre des détails, les détails doivent dépendre des abstractions.

5.Interface Segregation Principle : Aucune dépendance à l'égard d'interfaces indésirables. Les dépendances entre les classes doivent être basées sur des interfaces minimales. Il ne doit pas y avoir de méthodes dans chaque interface qui ne sont pas utilisées par les sous-classes mais qui doivent être implémentées ; sinon, l'interface doit être divisée. Il est préférable d'utiliser plusieurs interfaces isolées que d'utiliser une seule interface (une collection de méthodes de plusieurs interfaces en une seule).

6.Law of Demeter : Moins une classe en sait sur les classes dont elle dépend, mieux c'est. Quelle que soit la complexité de la classe dont on dépend, la logique doit être encapsulée dans les méthodes et rendue accessible à l'extérieur par des méthodes publiques. Ainsi, lorsque la classe dépendante change, cela n'affecte que très peu cette classe.

Les modèles de conception distillent certaines idées de conception communes en une série de modèles, puis donnent un nom à chaque modèle afin de faciliter la communication lors de son utilisation. Les 23 modèles communs sont donc répartis en trois grandes catégories : Creational Pattern, Structural Pattern et Behavioral Pattern.

Creational Pattern abstrait le processus d'instanciation des classes, permettant de séparer la création d'objets dans les modules logiciels de l'utilisation des objets, plutôt que de les instancier directement à l'aide de l'opérateur new. Cela donne au programme plus de flexibilité pour déterminer quels objets doivent être créés pour une instance donnée.

Structural Pattern se concentre sur la composition des objets et les dépendances entre eux, décrivant comment les classes ou les objets peuvent être combinés pour former une structure plus large. Le concept d'héritage est utilisé pour combiner les interfaces et pour définir les moyens de combiner les objets afin d'obtenir de nouvelles fonctionnalités.

Behavioral Pattern s'intéresse au comportement des objets et constitue une abstraction de la répartition des responsabilités et des algorithmes entre différents objets ; il se concentre non seulement sur la structure des classes et des objets, mais aussi sur leurs interactions.

Template Pattern est un Behavioral Pattern dans lequel une classe abstraite définit ouvertement la manière/le modèle d'exécution de ses méthodes. Template Pattern définit le squelette d'un algorithme en fonctionnement, tout en reportant certaines étapes aux sous-classes. Les méthodes modèles permettent aux sous-classes de redéfinir des étapes particulières d'un algorithme sans modifier la structure de cet algorithme.

Template Pattern est souvent utilisé dans le développement de jeux car de nombreux éléments d'un jeu sont souvent réutilisés pour certaines structures qui doivent être réécrites en détail. Ce modèle de conception est également utilisé dans ce projet.

La mise en œuvre de ce modèle de conception nécessite une analyse de l'algorithme cible pour déterminer s'il peut être décomposé en plusieurs étapes. Nous devons considérer les étapes qui sont génériques et celles qui sont différentes les unes des autres du point de vue de toutes les sous-classes. Bien qu'il soit possible d'avoir toutes les étapes en tant que types abstraits, l'implémentation par défaut peut être bénéfique pour certaines des étapes car les sous-classes n'ont pas besoin d'implémenter ces méthodes. Bien entendu, le modèle de modèle présente un inconvénient : chaque fois que l'algorithme cible change, le développeur peut être amené à modifier toutes les classes.

En fait, le Factory Pattern du Creational Pattern est une forme spéciale du Template Pattern. En même temps, Factory Pattern peut être utilisé comme une étape dans un Template Pattern plus large.

2.6 Présentation de FLTK

FLTK (Fast Light Tool Kit) est un kit d'outils GUI développé en C++ pour les systèmes d'exploitation Unix, Linux, MS-Windows95/98/NT/2000 et MacOS. Il est petit, rapide et a une meilleure portabilité que beaucoup d'autres kits de développement GUI (par exemple MFC, GTK, QT, etc.).

FLTK fournit des artefacts d'interface graphique multiplateforme avec des menus, des boutons, des fenêtres, etc. La gestion des événements FLTK est mise en œuvre à l'aide de fonctions de rappel et la gestion des messages FLTK est mise en œuvre à l'aide de la fonction virtuelle $Fl\_Widegt :: handle()$. FLTK supporte OpenGL et les opérations liées à OpenGL, les développeurs doivent juste utiliser l'artefact Fl_Gl_Window, redéfinir une classe dérivée de Fl_Gl_Window et ensuite surcharger draw() et handle().

2.7 Présentation d'OpenGL

OpenGL est une API multi-langage et multi-plateforme pour le rendu de graphiques vectoriels 2D et 3D, une interface composée de près de 350 appels de fonction différents pour dessiner tout, des simples bits graphiques aux scènes 3D complexes. Il s'agit d'un langage indépendant du système d'exploitation et de programmation croisée, et les applications développées à partir de ce langage peuvent être portées facilement et commodément sur diverses plates-formes. openGL n'est pas un logiciel libre, et ses droits d'auteur et sa marque (le nom OpenGL) sont la propriété de SGI. Cependant, il existe un remplacement pour OpenGL sous Linux : Mesa. Mesa fournit une interface qui est presque identique à OpenGL.

OpenGL comporte sept fonctions principales : la construction de modèles 3D, les transformations graphiques, les modes de couleur, les paramètres d'éclairage et de matériaux, le mappage de textures, les fonctions d'amélioration de l'image et les extensions pour les affichages bitmap, ainsi que les fonctions de mise en cache double.

OpenGL abstrait les ressources de l'ordinateur en objets OpenGL, et les opérations sur ces ressources en instructions OpenGL. Avant qu'une application puisse appeler une instruction OpenGL, elle doit d'abord s'arranger pour créer un contexte OpenGL. Ce contexte est une très grande machine d'état qui contient les différents états d'OpenGL, qui est la base de l'exécution des instructions OpenGL. Les fonctions OpenGL sont orientées vers les procédures, opérant essentiellement sur un état ou un objet dans la grande machine d'état qui est le contexte OpenGL, sous l'objet courant.

De nombreuses bibliothèques de développement et d'interface utilisateur offrent la possibilité de créer automatiquement des contextes OpenGL, FLTK étant l'une d'entre elles.

3 Résumé du projet

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 orien-

tée objet. On voit comment concevoir et hériter des classes, comment déclarer des variables et des fonctions et comment encapsuler des données et des méthodes.

Ce projet nous permet de voir comment réaliser un jeu simple en 3D à l'aide de OpenGL

Enfin, ce projet nous a montré comment séparer la partie traitement du programme de sa partie interface utilisateur.

4 Bibliographie

Références

[1] Erich Gamma, Richard Helm, Ralph Johnson, John Vlissides. "Design Patterns", 1994.