POLYTECH SORBONNE

SORBONNE UNIVERSITÉ

Rapport de projet C++ : Hazel

Par Mélissa LEBRETON et Younès CHEFOU

Sommaire

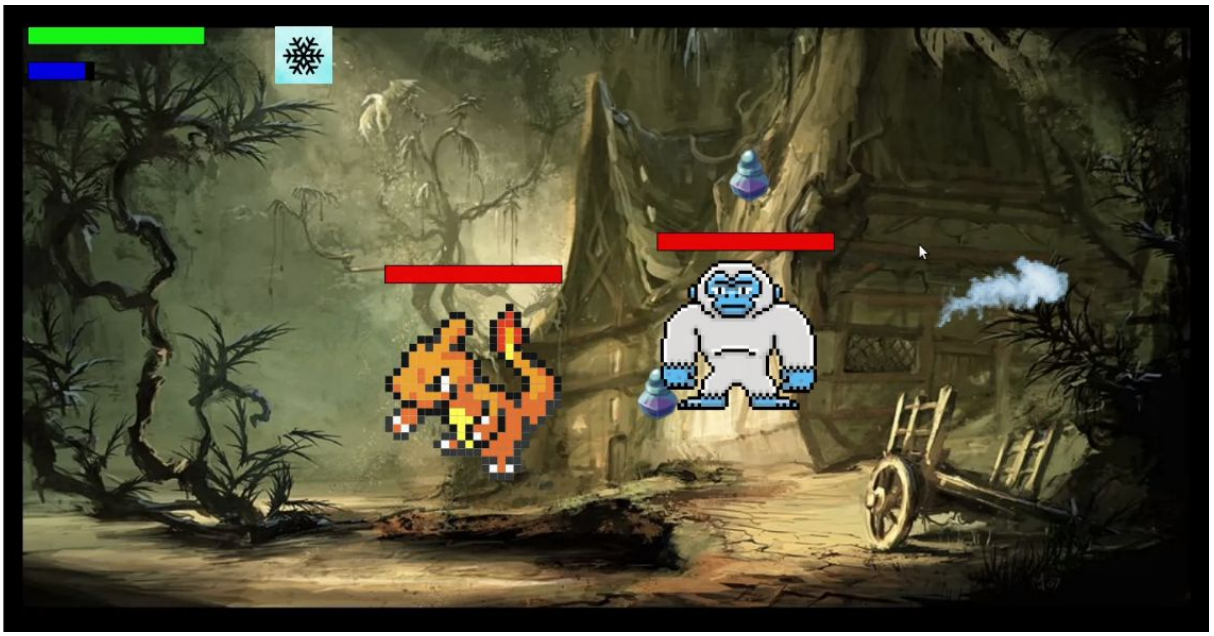
Introduction

Dans le cadre de l'UE de Programmation Objet en C++, il nous a été demandé de réaliser un projet sur le thème "Smoke". Etant tous les deux fans de jeux vidéo, nous avons décidé de développer un jeu 2D intitulé *Hazel.* Dans ce rapport, nous allons détailler le principe du jeu, décrire l'architecture du projet et comment les classes interagissent entre elles et comment installer le jeu sur une machine. Nous allons également des aspects du jeu dont nous sommes fiers et les perspectives d'amélioration que nous avons en tête pour le jeu.

Le principe du jeu

Hazel est un jeu 2D où le joueur incarne une sorcière pouvant se transformer en quatre types de fumées différentes correspondant chacune à un élément :

- fumée rouge pour FEU,
- fumée bleue pour GLACE,
- fumée violette pour POISON,
- fumée verte pour SOIN.



Capture d'écran du jeu où on peut voir la sorcière transformée en fumée bleue, un ennemi de type Feu, un de type Glace et deux objets pour recharger les points de vie

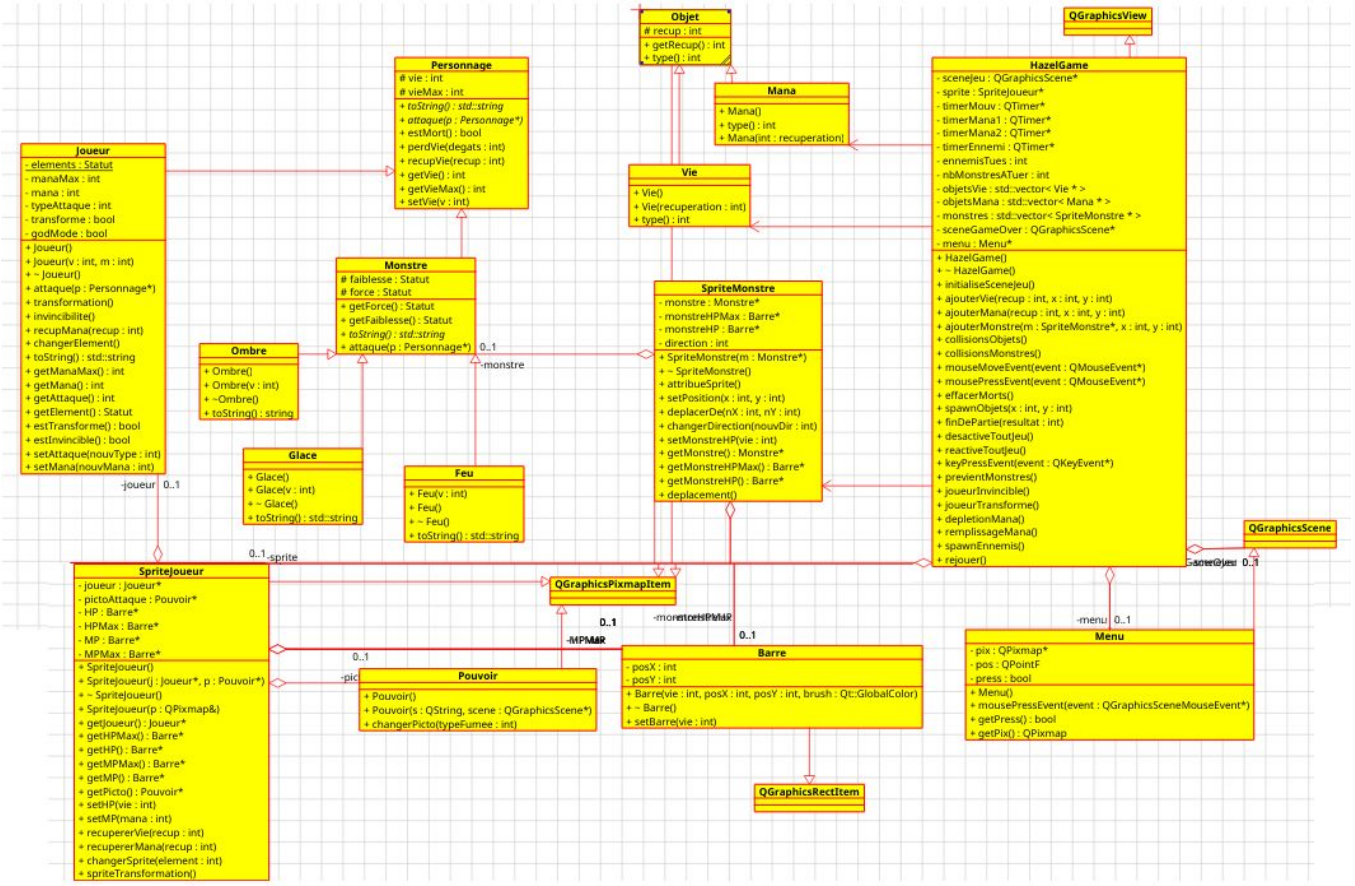
Le but du jeu est d'utiliser ces divers fumées à bon escient afin de venir à bout des monstres qui envahissent l'écran. Chaque monstre a une force et une faiblesse qui correspondent elles aussi à un élément. Pour vaincre un ennemi, le Joueur doit se transformer et utiliser la fumée qui a pour élément la faiblesse du Monstre.
Par exemple, pour tuer un Monstre Feu, le Joueur utilise la fumée bleue, s'il utilise la fumée rouge, le Monstre récupère de la vie.
Être transformé épuise la Mana (points de magie) et le Joueur se "détransforme" dès que celle-ci est entièrement vide.

Lorsque le joueur n'est pas transformé, il est vulnérable aux attaques des monstres et il y a aussi des règles s'appliquant par rapport aux éléments de chacun.
Par exemple, si le joueur a comme élément sélectionné Feu et qu'il est attaqué par un ennemi Glace alors il subira une plus grosse attaque que s'il avait sélectionné Glace.
Chaque attaque de Monstre fait baisser les points de vie du joueur, qu'il peut recharger avec les objets Vie apparaissant dès qu'un ennemi meurt. Si les points de vie du joueur arrivent à 0 alors c'est la fin de la partie.

Maintenant que nous connaissons les règles du jeu, étudions son architecture et les différentes classes permettant son bon fonctionnement.

L’architecture du projet

UML/Classes



Classe Personnage

La classe Personnage est la première classe que nous avons codé mais elle a continué à évoluer au fil du projet. Cette classe abstraite est la base pour le Joueur et ses ennemis car elle introduit les attributs vie et vieMax mais aussi quelques méthodes essentielles pour les classes Joueur et Monstre :

```
bool estMort(){return vie <= 0 ? true : false;}
//Retire des pts de vie au personnage
//degats : pts de vie perdus
void perdVie(int degats){=

//Rajoute de la vie au personnage
void recupVie(int recup)
{=

//Accesseurs
int getVie(){return vie;}
int getVieMax() {return vieMax;}

//Mutateurs
void setVie(int v){vie = v;}
```

Méthodes définies de la classe Personnage

Dans le fichier Personnage.hh, juste avant la définition de la classe Personnage, se trouve la définition du type Statut, (FEU, GLACE, etc.). Cela permet donc aux classes héritant de Personnage ou ayant “Personnage.hh” dans les fichiers inclus de profiter de ces éléments.

```
//Nouveau type Statut qui contient les différents pouvoirs, forces et faiblesses
typedef enum {PHYSIQUE, FEU, POISON, SOIN, GLACE} Statut;
```

Type Statut

En C++, une classe est considérée abstraite lorsqu’elle possède au moins une méthode virtuelle. Ces méthodes doivent être définies dans les classes fille pour que celles-ci ne soient pas également abstraites. Dans la classe Personnage, les méthodes abstraites sont :

```
std::string virtual toString() = 0;
void virtual attaque(Personnage* p) = 0;
```

Méthodes abstraites de Personnage

Classe Joueur

La classe Joueur hérite de la classe Personnage et contient toutes les données concernant le joueur : les points de vie, les points de Mana, et si la sorcière est transformé ou pas. La classe Joueur a également un tableau statique elements[]avec les différents éléments utilisables et un attribut, qui est un entier, typeAttaque. Cet attribut est l'indice de l'élément dans le tableau que le joueur a selectionné. Pour récupérer l'élément, il existe une méthode getElement() qui renvoie elements[typeAttaque].

Comme dit précédemment, pour que la classe Joueur ne soit pas une classe abstraite, elle doit définir les méthodes void attaque(Personnage* p) et std::string toString().

```
void Joueur::attaque(Personnage *p){
    Monstre* M = (Monstre*) p;
        if(getElement() == M->getForce()){
            std::cout << "L'attaque n'est pas très efficace... " << getElement() << std::endl;
            p->recupVie(5);
        }
        else if(getElement() == M->getFaiblesse()){
            std::cout << "L'attaque est très efficace !" << getElement() << std::endl;
            p->perdVie(60);
        }
        else{
            std::cout << "L'attaque est normale." << getElement() << std::endl;
            p->perdVie(20);
        }

}
```

Méthode attaque(Personnage p)*

Cette méthode est au coeur du jeu car c'est celle-ci qui va permettre à notre Joueur d'attaquer un Monstre. On commence tout d'abord par caster p en Monstre* et avec la méthode getElement() qui renvoie l'Element actuellement sélectionné, on teste si l'attaque est forte, normale ou si elle donne des points de vie au Monstre.

On introduit également dans la classe Joueur, le concept d'invincibilité, représenté par le booléen *godMode.* Quand l'attribut *godMode* est true, le Monstre ne peut pas attaquer le joueur.

Classe Monstre

La classe Monstre dérive de la classe Personnage, on introduit de nouveaux attributs pour caractériser les monstres (force, faiblesse, couleur). Il s'agit d'une classe abstraite car nous voulons avoir des monstres avec des forces et des faiblesses qui varient selon le type d'élément auxquels ils appartiennent. Les classes Feu, Ombre, Glace héritent de la classe Monstre. Chacune de ces classes ont des forces et des faiblesses spécifiques.

Maintenant que nous avons parlé des classes de type *Jeu,* intéressons-nous aux classes de type *Graphique*.

Toutes ces classes dérivent de classes de l'API Qt qui est la bibliothèque graphique que nous avons utilisé pour ce projet. Sa flexibilité et le nombre de classes proposées étaient parfaitement adaptés à ce que nous souhaitions réalisés.

Classe QGraphicsPixmapItem

Comme de nombreuses classes dérivent de celle-ci, parlons de QGraphicsPixmapItem. C'est une classe de Qt permettant d'avoir un objet graphique dont l'apparence est basé sur une image chargé grâce à la méthode setPixmap(cheminImage). Les classes SpriteJoueur, SpriteMonstre, Pouvoir, Objet, Vie et Mana sont donc des QGraphicsPixmapItem.

Classe SpriteJoueur

Cette classe est donc la représentation graphique de la classe Joueur. Elle contient donc le Joueur et aussi les éléments graphiques permettant de visualiser le type d'attaque représenté par la classe Pouvoir, les barres de vie et de mana représentées par des instances de la classe Barre.
Afin de faciliter certains traitements, la plupart des méthodes de SpriteJoueur sont des appels de fonctions de Joueur. Il y a par exemple, recupererVie(int recup) permettant d'appeler recupVie(int recup) de Joueur et aussi de mettre à jour la barre de vie.

Classe SpriteMonstre

La classe SpriteMonstre est similaire à la classe SpriteJoueur puisqu'elle est la représentation graphique de la classe Monstre. Elle contient tout ce qui concerne les barres de vie. Contrairement à la classe SpriteJoueur, cette classe gère automatiquement le déplacement du sprite sur la carte. Ce déplacement se fait en fonction du sprite du joueur afin que le monstre puisse se rapprocher et attaquer le joueur.

Classe Barre

Cette classe dérive de la classe QGraphicsRectItem. Elle permet de visualiser sur l'écran de jeu, les barres de vie et mana d'un personnage. Selon le type de la barre (Vie, Mana) et le personnage, la barre n'aura pas la même couleur. On a également ajouté des attributs posX et posY à cette classe afin de pouvoir choisir l'emplacement de la barre sur l'écran.

Classe Pouvoir

La classe Pouvoir permet d'afficher l'icône de l'attaque actuellement choisie par le joueur. Lors du démarrage du jeu, l'attaque de base du joueur est "Physique". Donc l'icône affichée correspond à cette attaque. Lorsque le joueur change d'attaque, la fonction changerPicto charge l'icône de la nouvelle attaque.

Classes Objet, Mana, Vie

La classe Objet est une classe abstraite qui est la classe mère des classes Vie et Mana. La classe Vie permet au joueur de récupérer de manière aléatoire de la vie en récupérant un sprite de coeur. De façon similaire, la classe Mana permet au joueur de récupérer du mana de manière aléatoire en récupérant un sprite d'une potion.

Classe HazelGame

La classe HazelGame est notre classe principale, celle qui gère entièrement notre jeu dont elle reprend le nom. Cette classe hérite d'une classe Qt appelé *QGraphicsView* nous permettant ainsi d'avoir un affichage dans lequel nous faisons apparaître différentes *QGraphicsScene* correspondant aux différents états de jeu.
Les deux premières scènes que nous affichons sont l'Intro et le Menu où nous expliquons à l'utilisateur les commandes. La troisième scène est le jeu en lui-même, que vous pouvez voir sur la première photo de ce rapport.
HazelGame prend comme attributs le SpriteJoueur, la scene du jeu, 4 *QTimer* que nous allons détailler juste après, le nombre d'ennemis tués par le joueur et le nombre d'ennemis à tuer pour gagner la partie, et trois vecteurs contenant chacun les objets Vie, les objets Mana et les ennemis présents à l'écran.
La méthode la plus longue ici est donc l'initialisation de tous ces éléments : initialisationSceneJeu() :

- On initialise d'abord la scene du Jeu comme une QGraphicsScene de 1350x700 et on ajoute un fond grâce à une image de notre dossier *Ressources*,
- On crée ensuite le SpriteJoueur à partir d'une instance de Joueur et de Pouvoir, cela crée aussi les barres de vie et de mana et on les ajoute à la scène crée juste avant avec la méthode addItem,
- On définit le nombre d'ennemis tués à 0 et le nombre d'ennemis à tuer à 25,
- On place l'instance de Pouvoir à côté des barres de vie et de mana,
- On active le MouseTracking pour utiliser la position de la souris dans les evenements.

La dernière partie de la méthode est l'initialisation des quatre QTimer :

- timer de déplacement des monstres,
- timers de depletion de Mana quand le joueur est transformé et remplissage de Mana quand il ne l'est pas,
- timer faisant apparaître les ennemis de façon périodique.

Un QTimer marche de la manière suivante : on crée le timer, puis on utilise la fonction connect pour le lier à QGraphicsView, une fonction SLOT et un SIGNAL. Le SIGNAL est une sorte d'interruption indiquant au timer d'appeler la fonction. Pour tous nos timers, nous avons utiliser SIGNAL(timeout()) qui signifie que la fonction sera appelée dès que le timer atteint 0.

```
timerMouv = new QTimer();
//Chaque fois que le timer arrive à zero, on appelle deplacement
connect(timerMouv, SIGNAL(timeout()), this, SLOT(previentMonstres()));
timerMouv->start(5); //Toutes les 5 ms
```

Timer de déplacement des monstres

Par exemple, dans l'image ci-dessus, on appelle la fonction previentMonstres() toutes les 5 millisecondes. Cette méthode compare la position des Monstres et celle du SpriteJoueur pour leur indiquer quelle direction prendre.

Fonctions commandes

Le jeu se joue à l'aide de la souris : le personnage est contrôlé en faisant bouger le curseur, on change d'élément avec un clic droit et on attaque l'ennemi en se déplaçant vers lui. Pour programmer cela, on se sert des événements mouseMoveEvent et mousePressEvent.

```
//Permet de déplacer le personnage à l'aide de la souris
void HazelGame::mouseMoveEvent(QMouseEvent *event)
{
    int x = event->x();
    int y = event->y();
    sprite->setPos(x,y);
    collisionsObjets();
    collisionsMonstres();
}
```

mouseMoveEvent

Les fonctions collisions étant liées à la position du joueur, on appelle les fonctions collisionsObjets et collisionsMonstres dès que le joueur se déplace.

Fonctions collisions

La fonction collisionsObjets verifie si le joueur est entré en contact avec l'un des objets des vectors objetsVie et objetsMana et appelle respectivement recupVie ou recupMana si oui.
La fonction collisionMonstres verifie la même chose mais avec le vector monstres. En accordance avec les règles du jeu, le joueur ne peut attaquer que s'il est transformé et sa mort résulte en une fin de partie. Lorsqu'un monstre attaque le joueur, celui-ci est invincible pendant une seconde afin qu'il ait le temps de fuir.

```
void HazelGame::collisionsMonstres(){
    int nbMonstres = monstres.size();
    Joueur* persoJoueur = sprite->getJoueur();
    Monstre* monstre;

    for(int i = 0; i < nbMonstres; i++){
        monstre = monstres[i]->getMonstre();
        if(sprite->collidesWithItem(monstres[i])){
            if(persoJoueur->estTransforme()){//Le joueur attaque
                persoJoueur->attaque(monstre);
                monstres[i]->setMonstreHP(monstre->getVie());
            }
            else{//Le Monstre attaque
                monstre->attaque(persoJoueur);
                sprite->setHP(persoJoueur->getVie());
                if(persoJoueur->estMort()){
                    finDePartie(0);
                    return;
                }
                else if(!persoJoueur->estInvincible()){ //On rend le joueur invincible pendant 1 seconde
                    joueurInvincible();
                    QTimer::singleShot(1000, this, SLOT(joueurInvincible()));
                }
            }
        }
    }

    effacerMorts();
}
```

Fonction collisionMonstres

La fonction effacerMorts() efface les Monstres morts du vector monstres et de la scène. Un objet peut apparaître à l'endroit où le monstre est mort.

Main

Notre fonction main permet d'ouvrir la fenêtre de jeu. La variable app de type QApplication permet d'initialiser la fenêtre et de la modifier ensuite. On appelle un objet de la classe HazelGame qui contient les différentes données du jeu afin de les afficher sur la fenêtre du jeu. Celle-ci apparaît sur l'écran de manière à occuper tout l'espace disponible.

Nos ressources graphiques

Toutes les images utilisées pour notre jeu proviennent d'Internet. Cependant, la plupart d'entre elles sont un mix de plusieurs images. Pour adapter ces images, nous avons utilisé des logicielles de retouche d'image comme PhotoFiltre 7 ou GIMP.

Installation

Après avoir cloné le dossier avec la commande :

```
git clone https://github.com/YounesChefou/Hazel.git
```

Il faut aller dans le dossier Installation et entrer la commande make.

```
cd Installation/
make
```

Il suffit ensuite de rentrer cette dernière commande pour lancer le jeu !

```
./hazel
```

Conclusion

Ce projet nous a permis d'appliquer toutes nos connaissances en programmation objet et en C++ mais aussi d'en apprendre au fil du développement. Nous sommes particulièrement fiers de l'inclusion des Timers ajoutant à la dynamique du jeu avec la gestion de la Mana et le choix du bon élément afin de vaincre les ennemis.