Projet Yellow

Pac MAIN

Auteur : LIU Vincent
MAIN 4, 2018 - 2019

Encadrants : Cécile Braunstein
Jean-Baptiste Brejon

17 février 2021

Ce document est un compte rendu du projet "Yellow" dans le cadre du cours de "N7-IOB Langage objet C++". Il a été réalisé par **Vincent LIU**, un étudiant MAIN 4 de Polytech Sorbonne lors de l'année scolaire 2018 - 2019. Les encadrants pour ce projet sont : **Cécile Braunstein** (LIP6) et **Jean Baptiste Brejon** (LIP6).

Ce projet final a pour but de mettre en application les notions théoriques acquises dans ce cours. C'est pourquoi, l'application réalisée doit satisfaire certaines contraintes :

- 8 classes,
- 3 niveaux de hiérarchie,
- 2 fonction virtuelles différentes,
- 2 surcharges d'opérateurs,
- 2 conteneurs différents de la STL,
- le diagramme de classe UML doit être fourni et complet,
- le code doit être commenté,
- pas d'erreur avec valgrind,
- un makefile doit être associé à l'application.

Table des matières

1 Description

Pour ce projet, j'ai choisi de modéliser le célèbre jeu d'arcade **Pac Man** à ma manière.

But du jeu [1] :
Le jeu consiste à déplacer un personnage qui se trouve dans un labyrinthe afin de lui faire manger toutes les ressources qui s'y trouvent en évitant d'être touché par des fantômes.
Chaque ressource permet au joueur de gagner des bonus qui permettent d'augmenter le score du joueur.
Lorsqu'un joueur est touché par un fantôme, il perd une vie : le jeu s'arrête lorsqu'il n'a plus de vie restante.

Mon implémentation :
Mon application s'est donc fortement inspiré de ce jeu classique, malgré tout, certaines fonctionnalités concernant le comportement des fantômes ne sont pas implémentées.

Cahier des charges :
Le jeu doit pouvoir être joué :
- en mode 1 joueur facile,
- en mode 1 joueur difficile.

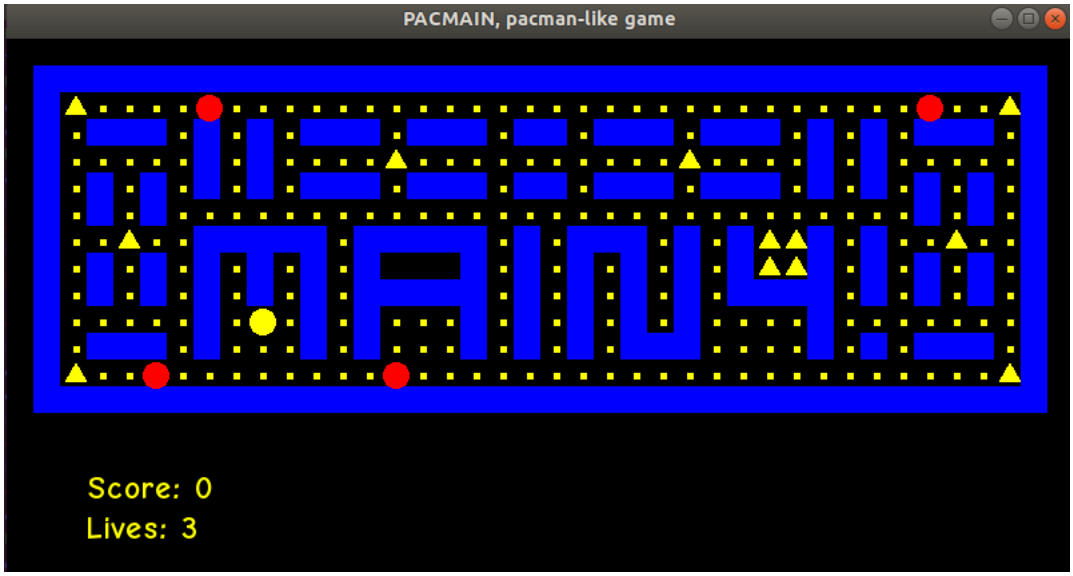

FIGURE 1 – Aperçu de l'interface - Le joueur contrôle le personnage "rond jaune", et doit éviter les "ronds rouges"

2 Présentation de l'implémentation

Afin de modéliser le jeu, 12 classes ont été utilisées.

2.1 Éléments du jeu

9 classes permettant de représenter les éléments du jeu.
Les codes sources correspondant se trouvent dans le répertoire *lib*.

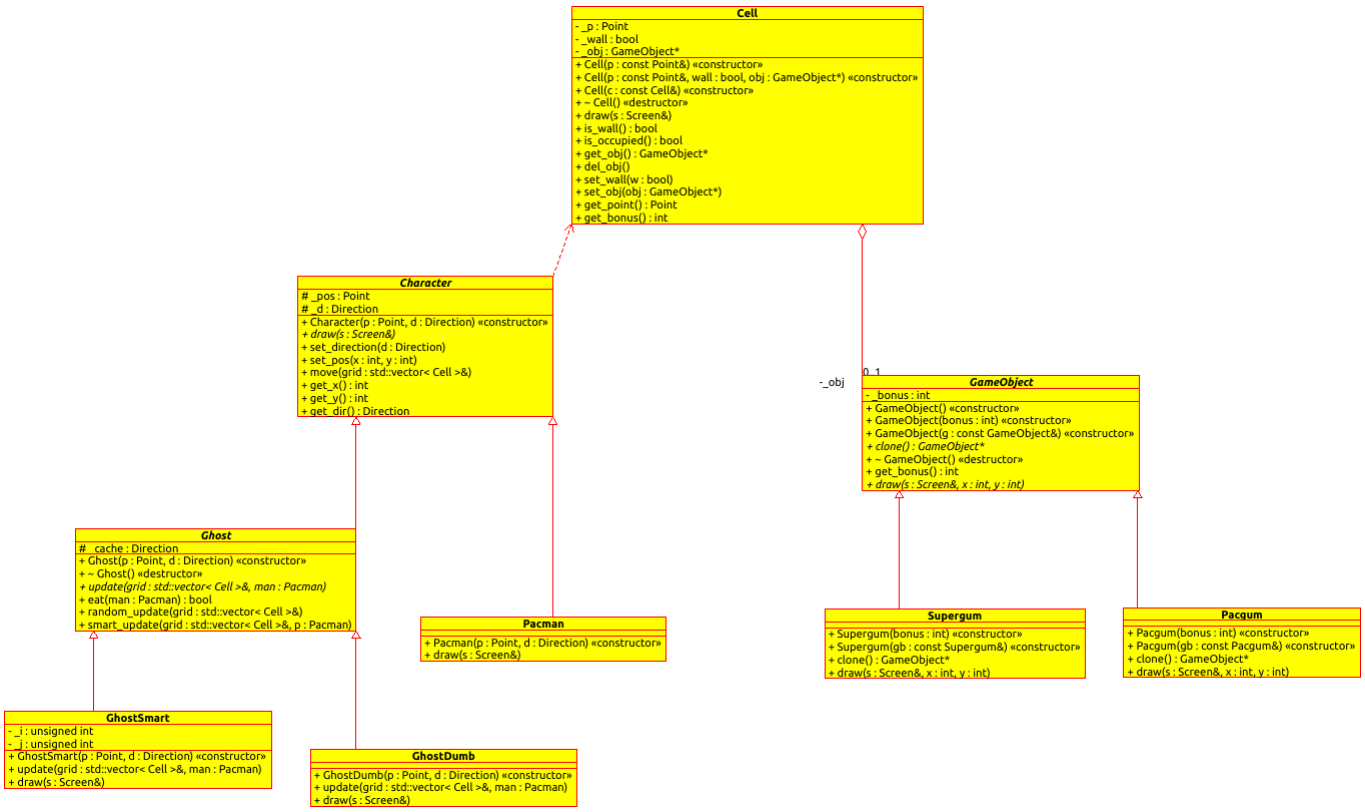

FIGURE 2 – Diagramme UML Réduit - La version complète se trouve en annexe

Deux types de classes se dégagent de cette modélisation :
— la classe *Personnage* ou "***Character***", qui va regrouper les éléments qui peuvent se déplacer dans le labyrinthe.
On aura besoin de leur *direction* de déplacement ainsi que de leur *position*.
— la classe *Objet du jeu* ou "***GameObject***", qui va regrouper les éléments qui peuvent être consommés par le héros du jeu, ils apportent des *bonus* qui augmentent le *score* du joueur.

Ces classes sont donc des classes abstraites, c'est-à-dire qu'on ne pourra pas crée d'instance de ces classes. Cependant, elles servent de fondations pour leurs classes filles. On a procédé à un ***héritage*** afin de regrouper ces deux principaux types d'objets.

2.1.1 Classe Personnage

Parmi les éléments qui peuvent se déplacer, nous pouvons distinguer "**Pac-man**", le personnage que contrôle le joueur, ainsi que **les fantômes**. Les fantômes peuvent avoir deux comportements différents, qui correspondent à leur manière de se déplacer. On a donc procédé à un nouvel ***héritage*** afin de distinguer ces deux types de fantômes.

Personnage	**Règle de déplacement**	**Représentation**
Pacman	Contrôle de l'utilisateur	Rond jaune
Fantôme aléatoire	Direction aléatoire, mise à jour lors de la rencontre d'un obstacle	Rond blanc
Fantôme prédateur	Parfois comportement aléatoire, parfois direction la plus proche	Rond rouge

TABLE 1 – Détail des personnages instantiables

2.1.2 Objets du jeu

Parmi les consommables, nous trouvons simplement les **pac-gommes** et les **super-gommes**, qui diffèrent par leur bonus gagné par le joueur.

Consommables	**Bonus**	**Représentation**
Pac-gommes	10 points	carré jaune de petite taille
Super-gommes	100 points	triangle jaune de moyenne taille

TABLE 2 – Détail des consommables instantiables

2.1.3 Classe Cellule

Le labyrinthe va être représenté par un tableau de ***cellules***.
Les *cellules* contiennent toutes les informations nécessaires relatives à une case.
C'est une des classes les plus importantes, elle permet de donner l'information
— de l'**accessibilité d'une case** : les personnages peuvent-ils se déplacer dans cette direction ?
Cette question est répondue par un attribut booléen *wall*.
— du **contenu d'une case** : quelle est le consommable qui se trouvent à la cellule C_i ?
La classe contient un attribut de type "*Pointeur sur GameObject*" pour répondre à cette question.
Lorsque ce pointeur est le pointeur *NULL*, la cellule correspondant est vide.
Sinon, elle contient soit un pac-gommes, soit un super-gomme.

Nous avons donc utilisé une technique de ***polymorphisme*** car le type de l'objet contenue dans la cellule n'est connue seulement à l'exécution du programme.

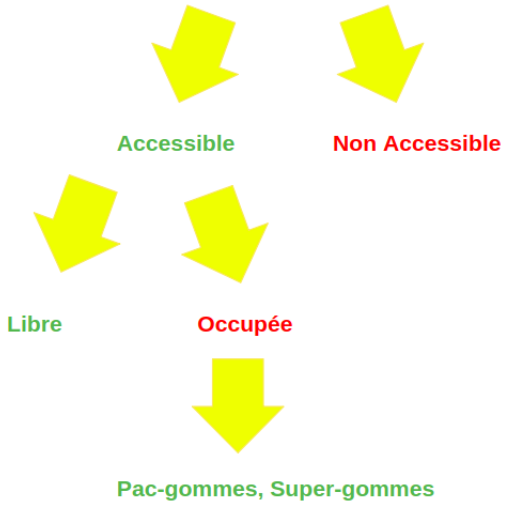

FIGURE 3 – Information de la cellule

2.2 Structure de l'application

3 classes permettent de structurer l'application.
Les codes sources correspondant se trouvent dans le répertoire *src*. Il s'agit des classes :

- **Screen**, qui gère l'interface graphique du jeu. La bibliothèque graphique utilisée est **SFML**.
- **Player**, qui gère les informations principales à propos du joueur.
- **Game**, qui gère le fonctionnement du jeu en général.

Attribut	**Type**	**But**
h, w	uint16_t	Dimensions de la fenêtre graphique
win	sf : :RenderWindow *	Fenêtre graphique en 2D

TABLE 3 – Description des attributs de la classe Screen

Attribut	**Type**	**But**
score	unsigned long	Score du joueur
life	int	Vie restante du joueur (0 à 3)
Man	Pacman	Personnage associé au joueur

TABLE 4 – Description des attributs de la classe Player

2.3 Contraintes

2.3.1 Méthodes virtuelles

Les méthodes virtuelles utilisées sont assez simple :
— la méthode ***draw*** pour les éléments du jeu.
Elle prend en argument la classe *Screen*, et dessine donc son rendue sur la fenêtre graphique.
— la méthode ***update*** pour les fantômes afin de mettre à jour la direction de déplacement. La règle de déplacement sera détaillée dans la partie **Fierté**.
— la méthode ***clone*** pour permettre un constructeur de copie virtuel.
Il a été nécessaire de l'implémenter car la copie virtuel était nécessaire. En effet, le type de ressources est connue seulement à l'exécution. La copie virtuel permet d'appeler une bonne version de constructeur par copie lorsque le type connu à la compilation est seulement la classe *GameObject*, qui est la classe mère.

2.3.2 Conteneurs

Comme evoqué dans la partie 2.1.3, le labyrinthe va être représenté par un tableau de ***cellules***.
Le choix d'un conteneur de la STL parait donc ainsi naturel, la difficulté réside dans le choix entre un *vector* et une *list*.
D'après [3] et le cours de C++, l'avantage de la *list* est de retirer un élément en temps constant ($O(1)$). Par contre, l'accès à un élément est en temps linéaire en la taille du tableau pour une *list* ($O(n)$) alors qu'elle est en temps constant pour un *vector* ($O(1)$).
Comme la taille du labyrinthe est fixe, l'avantage de la *list* est négligeable. De plus, dans le programme, nous allons souvent accéder à un élément pour vérifier l'accessibilité, le choix d'un ***vector*** de ***cellules*** est donc plus logique.

Les conteneurs ont également également beaucoup servi dans les méthodes *update*, qui mettent à jour la règle de déplacement des fantômes.
— le conteneur ***vector*** pour stocker les directions possibles à un instant donné,
— le conteneur ***unordered_map*** où on associe une direction à une valeur d'heuristique par rapport au personnage principal. Cela permet aux fantômes de se déplacer vers *Pac-man* avec un chemin court.

2.3.3 Surcharges d'opérateur

L'opérateur $<<$ a été surchargé pour l'affichage des informations dans la console.
— pour la classe joueur, cela affiche le nombre de vie restante si le jeu tourne encore. Sinon, si le joueur n'a plus de vie, cela affiche son score final.
— pour la classe jeu, cela affiche l'état de jeu.

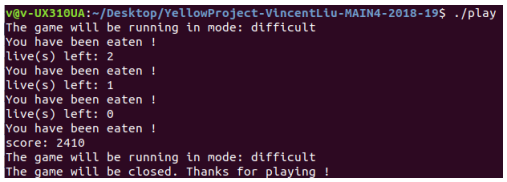

FIGURE 4 – Affichage des informations du jeu

3 Fierté

3.1 Classe Game

Le fait d'avoir bien structurer l'application est une de mes fiertés principales quant à la création du jeu. La classe ***Game*** sert à assurer le bon fonctionnement du jeu. Elle possède un attribut très important : le GameStates qui permet d'effectuer les transitions entre les différents modes de jeu.

Valeur	**But**
STATE_START	Mode menu : l'utilisateur choisit la difficulté
STATE_GAME	Mode jeu classique : l'utilisateur joue
STATE_DEAD	Mode arrêt de jeu : le personnage est mort
STATE_END	Mode arrêt de l'application : la fenêtre se ferme

TABLE 5 – Valeur de l'enumeration GameStates

Lorsqu'on appelle la méthode ***run***, on regarde simplement l'état actuel pour appliquer les fonctions adaptées à la situation.
Dans une boucle de jeu, en *C++*, cela ressemble à [9] :

```
switch( gameState )
      case STATE_MENU:
          // Fonctions
      case STATE_GAME:
          // Fonctions
      case STATE_DEAD:
          // Fonctions
      case STATE_END:
          // Fonctions
```

Les fonctions à appeler sont :
- *handle_event* : On va attendre les évènements comme un clic de souris, un bouton appuyé etc.
- *logic* : On va changer les éléments du jeu en fonction de l'événement apparue (ou non).
  Ces modifications comprennent le changement de direction, le déplacement, la consommation des ressources lorsqu'on est en *STATE_GAME* par exemple.
- *render* : On va réactualiser le rendue graphique dans la fenêtre en prenant en compte les nouvelles modifications.

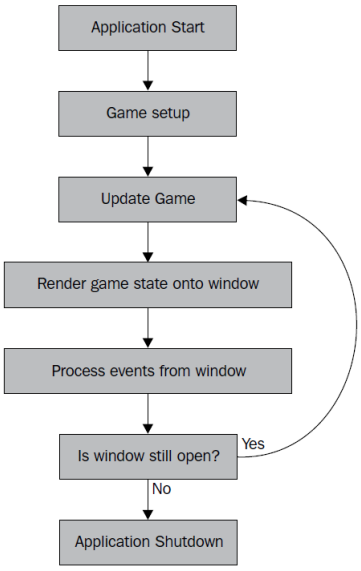

FIGURE 5 – Boucle du jeu d'après Game Class - SFML Tutorials and Practice [9]

3.2 Règle de déplacement

Les règles de déplacement n'ont pas été faciles à implémenter. Simplement 2 algorithmes de déplacement ont été implémentés, alors que dans le vrai **Pac-man**, chaque fantôme a son propre algorithme.

3.2.1 Aléatoire

La règle adoptée a été la suivante :

1. La règle 1 permet de déplacer le fantôme lorsqu'il rencontre un obstacle : on va changer de direction, et il sera préférable de ne pas revenir en arrière. On revient en arrière si il s'agit de la seule direction possible.
2. Si il n'y a pas d'obstacle, on applique la règle 2 qui permet de changer de direction avec proba 0.2 pour qu'il n'ait pas trop un comportement monotone.

3.2.2 Prédateur

Pour l'algorithme du fantôme prédateur ou malin, le déplacement de celui-ci se fait en fonction de la position actuel du personnage principal.
On aurait pu implémenter l'algorithme $A*$, mais par manque de temps, on calcule seulement l'heuristique des fantômes et on se déplace simplement dans la direction où l'heuristique est le plus faible.

L'unique mode prédateur rend le jeu beaucoup trop difficile, donc les fantômes ont deux comportements différents en fonction du temps, ils ont une période

— aléatoire de 40 frames,
— prédateur de 20 frames.

4 Procédure d'installation et manuel d'utilisation

4.1 Procédure d'installation générale

Afin de télécharger la bibliothèque **SFML**

```
$ sudo apt-get install libsfml-dev
```

Afin de compiler le projet :

```
$ make
```

Afin de lancer le jeu :

```
$ ./play
```

4.2 Manuel d'utilisation

Menu

Afin de sélectionner le mode de jeu, deux possibilités :

1. flèche haut ou bas pour sélectionner, puis entrée pour confirmer,
2. Appuyer avec la souris directement.

Jeu

Le déplacement du personnage principal se fait avec les flèches.

Conclusion

Ce projet a été très riche en connaissances. Ce fût le premier jeu que j'ai conçu et je suis très fier du résultat.
Il m'a permis de gérer un petit projet en C++ pour mettre en avant les connaissances acquises lors de cette unité d'enseignement.
Les points à travailler serait :

- l'affichage : créer un design pour ses propres personnages. Les sons et images de Pac-man et des fantômes appartiennent à l'entreprise japonaise Namco, c'est pourquoi je ne les ai pas utilisé.
- les algorithmes de déplacement : ajouter plusieurs règles de déplacement.
- permettre à un deuxième utilisateur de déplacer les fantômes pour un mode 2 joueurs.
- ajouter des nouveaux consommables avec de nouveaux effets.

Références

[1] Pac-man. Wikipedia. Consulté en Decembre 2018.
Disponible sur : https://fr.wikipedia.org/wiki/Pac-Man

[2] How to Read from a Text File, Character by Character in C++, stackoverflow.
Consulté en Decembre 2018. Disponible sur : https://stackoverflow.com/questions/12240010/how-to-read-from-a-text-file-character-by-character-in-c

[3] When should vector list be used, StackExchange.
Consulté en Decembre 2018. Disponible sur : https://gamedev.stackexchange.com/questions/4887/when-should-vector-list-be-used

[4] Segmentation fault when push_back to vector c++, stackexchange.
Consulté en Decembre 2018. Disponible sur : https://stackoverflow.com/questions/41712386/segmentation-fault-when-push-back-to-vector-c

[5] C++ : Could Polymorphic Copy Constructors work ?, stackoverflow.
Consulté en Decembre 2018. Disponible sur : https://stackoverflow.com/questions/1021626/c-could-polymorphic-copy-constructors-work

[6] Copying a Polymorphic object in C++, stackoverflow.
Consulté en Decembre 2018. Disponible sur : https://stackoverflow.com/questions/5148706/copying-a-polymorphic-object-in-c

[7] Documentation of SFML 2.5.1, SFML.
Consulté en Decembre 2018. Disponible sur : https://www.sfml-dev.org/documentation/2.5.1

[8] Tutorial : Create and Use SetIcon, SFML
Consulté en Decembre 2018. Disponible sur : https://github.com/SFML/SFML/wiki/Tutorial:-Create-and-Use-SetIcon

[9] maksimdan, Game Class - SFML Tutorials and Practice
Consulté en Janvier 2019. Disponible sur : https://maksimdan.gitbooks.io/sfml-and-gamedevelopement/content/game_class.html

[10] Lazy Foo' Productions, State Machines
Consulté en Janvier 2019. Disponible sur : http://lazyfoo.net/articles/article06/index.php

A UML Complet

FIGURE 6 – Diagramme UML Complet du projet