INFO-F202: Projet CandyCrush

Vinovrski Alexandre (501.157)

Janvier 2022

Contents

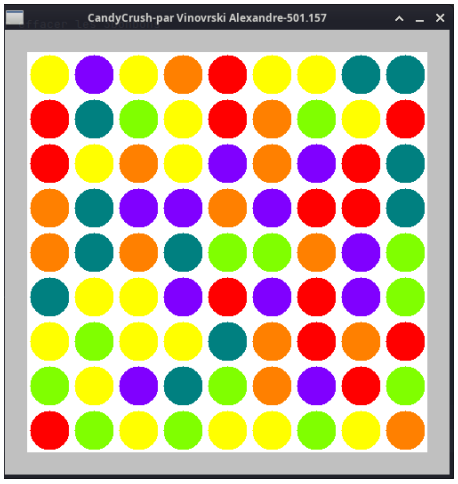
1 Introduction

Dans le cadre du cours de INFO-F202, il nous à été demandé d'appliquer la théorie du paradigme orienté objet dans un projet conséquent à faire seul ou en groupe, ayant pour but de reproduire le jeu Candy Crush.
Dans ce contexte ci, le projet à été réalisé seul avec le moins d'aide possible provenant de l'extérieur (hormis les travaux pratiques). Vous pouvez retrouver le projet sur ici : CandyCrush

1.1 Le jeu

Candy Crush est un jeu développé par la société King et se basant sur un principe simple. Le joueur fait face à un tableaux de bonbons de plusieurs couleurs. Lorsque 3 bonbons de même couleurs (ou plus) sont alignés, ces bonbons disparaissent et entraîne la chute des bonbons se situant au dessus. Il est possible selon la combinaison de bonbons d'obtenir des bonus permettant d'effacer le tableau plus facilement.

(a) Plateau du jeu original.



(b) Plateau du jeu implémenté.

2 Tâches du projet

Une dizaine de tâches ont été proposées allant des fonctionnalités de base du jeu jusqu'à l'édition de niveau, l'implémentation de bonus, ainsi que la gestion du score.

2.1 Tâches implémentées

Malheureusement de par un manque de temps issus de 2 semaines de maladie ainsi que le choix de faire ce projet seul sans aides extérieures, seule la première tâche proposée à su être développée entièrement.

2.2 Tâches en cours d'implémentation

Le projet à cependant des classes et des structure de classe permettant l'ajout de bonbons spéciaux et la possibilité de crée son propre niveau à partir d'un fichier texte. Seulement, ces fonctionnalités ne sont pas entièrement implémentées

2.3 Tâches non implémentées

L'ensemble des tâche restantes n'ont pas pu être structurées et implémentées dans les temps.

3 Structure du projet

3.1 Structure des répertoires

Le projet se structure en une série de répertoire, nous retrouvons l'arborescence suivante (Chaque fichier du projet représente une classe ou un ensemble de classes héritées.):

1. **src** : Dossier contenant les codes sources du projets ainsi que les exécutables et makefile.
   (a) **logic** : Dossier contenant les codes sources concernant la logique du jeu.
   (b) **gui** : Dossier contenant les codes sources concernant l'interface du jeu.
2. **srd** : Dossier contenant le rapport ainsi que les fichier *.uxf* des diagrammes UML.

3.2 Structure des objets, diagramme de classe

L'ensemble des classe implémentées suivent le paradigme orienté objet et sont agencées comme suit (Une explication avancée de chaque classe ainsi que leurs méthodes sera proposée dans les points à venir):

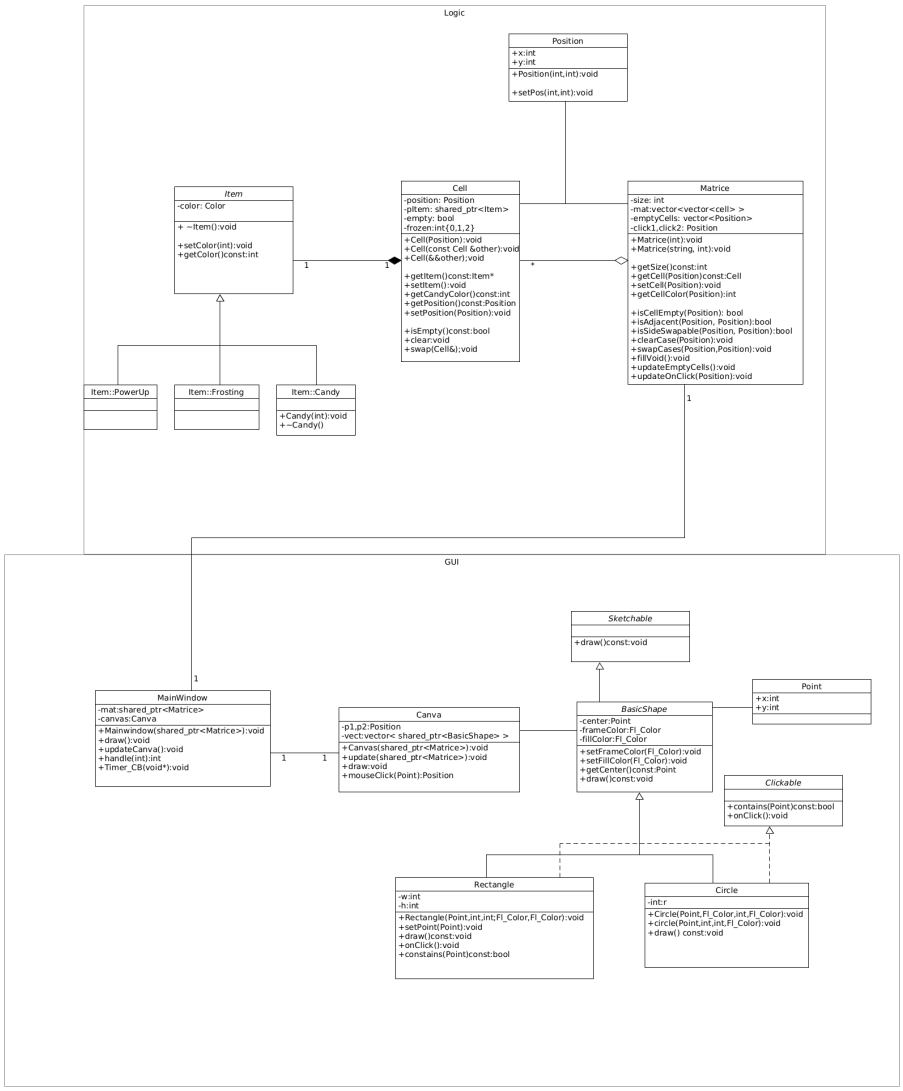


Figure 2: Structure des différentes classes classes

3.3 Structure des imports

Les imports des différents fichiers et librairies extérieures est représenté par la figure suivante:

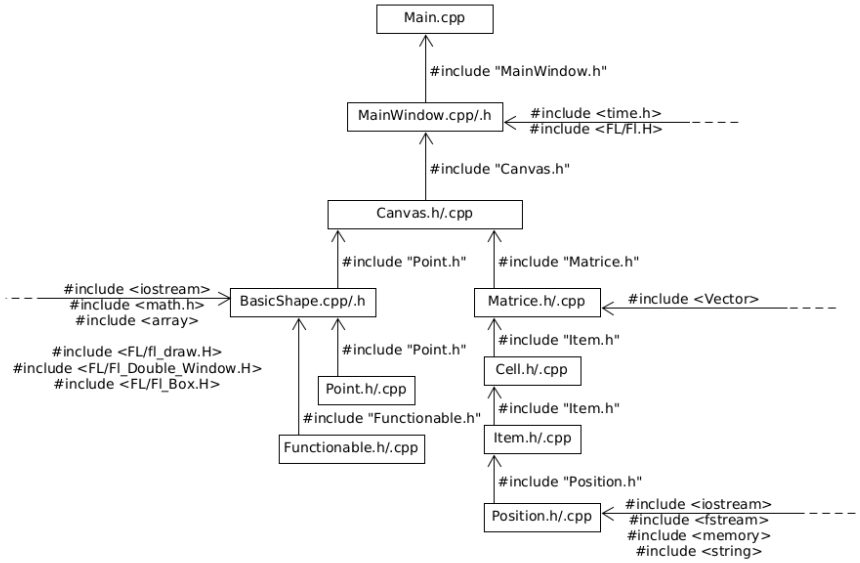


Figure 3: Gestion des imports

4 Détails des Classes

Dans cette partie, nous détaillons l'utilité de chaque classe implémentée.

4.1 Les classes du dossier *logic*

- **Matrice.cpp/.h**: Classe principale regroupant les autres classe du dossier et faisant le lien avec l'interface graphique selon le patron de conception MVC. Elle construit une matrice de taille fixe composée de cellule possédant chacune un bonbons d'une couleur. C'est à partir de la classe matrice que le échanges entre bonbons, les vérification d'alignement et autres sont exécutés.

- **Cell.cpp/.h**: Classe secondaire représentant une cellule possédant un pointeur intelligent vers un objet bonbon. Cette classe possède un attribut *frozen* qui aurait du servir à l'implémentation du glaçage.

- **Item.cpp/.h**: Ensemble de classe héritées donc la classe mère est une classe abstraite Item et dont l'objet Candy est issus. Cette ensemble de

classe héritée aurait du servir à l'ajout de bonbons spéciaux tel que les bombes chocolatées et les bonbons verticaux/horizontaux.

- **Position.cpp/.h**: Structure représentant la position d'une cellule dans la matrice. Cette structure est strictement identique à la structure *Point* de la section suivante. Cependant nous considérons nécessaire le fait de séparer entièrement l'interface de la logique.

4.2 Les classes du dossier *GUI*

- **MainWindow.cpp/.h**: Classe héritée de la classe *Window* de la librairie *FLTK*. Cette classe est la principale du répertoire. Elle permet de faire le lien et d'afficher la matrice ainsi que gérer les cliques sur cette dernière via un pointeur intelligent initialisé par le constructeur.
- **Canvas.cpp/.h**: Classe regroupant les forme de bases pour afficher l'interface de la matrice. Elle récupère aussi les signaux de cliques
- **BasicShape.cpp/.h**: Ensemble de classe héritées donc la classe principale est une classe abstraite *BasicShape* donc les classes *Circle* et *Rectangle* sont issue.
- **Functionable.cpp/.h**: Ensemble de classe non héritée reprenant les classes *Clickable* et *Sketchable* permettant de cliquer et dessiner chaque *BasicShape*
- **Point.cpp/.h**: Classe représentant une coordonnées sur une fenêtre et non une coordonnée au sein d'une matrice (d'où la différentiation entre la classe Point.cpp et Position.cpp).

5 Détails des méthodes de classes principales

Dans cette section, nous décrivons les méthodes les plus importantes de chaque classe. Nous ne détaillerons pas les différents constructeurs et surcharge d'opérateur afin d'éviter un travail trop long.

5.1 Classe *Cell*

Une cellule est un objet qui détient un objet *item*. Dans ce cas de figure, l'attribut item est un pointeur intelligent vers un objet *item*.

- **void Cell::clear()**: Méthode qui efface le contenu de la cellule en réinitialisant le pointeur intelligent à n*ullptr*
- **void Cell::swap(Cell& other)**: Méthode qui à l'aide du constructeur de déplacement, échange le contenu de deux cellules sans passer par des copies.

5.2 Classe *Matrice*

Cette classe est la représentation d'une matrice, alias un vecteur de vecteur de cellule. C'est à ce niveau que les vérification d'échange de cellules, de suppression, etc sont effectuées.Cette classe contient plusieurs vecteurs importants dont celui des cases vides et celui des cases à supprimer.

- **bool Matrice::isAdjacent(Position p1, Position p2)**: Méthode qui retourne un booléen lorsque deux cellules aux posistion p1 et p2 sont adjacente
- **bool Matrice::isSwapable(Position p1, Position p2)**: Méthode quiretourne yun booléen lorsque deux cellules sont échangeables. Il faut vérifier que la première cellules est à sa nouvelles position des cellules adjacentes de même couleurs, et faire de même pour la seconde cellule.
- **void Matrice::clearCase(Position p)**: Efface le contenu d'une cellule via l'appel à *void Cell::clear()* et mets à jours le vecteur des position vides de la matrice
- **void Matrice::swapCases(Position p1, Position p2)**: Echange les contenus des case aux deux positions à l'aide de l'appel à *void Cell::swap()*. Lorsque les deux cases sont vide, il est inutile de les échangées. Cependant, il faut veiller à garder le vecteur des cases vide à jours.
- **int Matrice::updateToDelete()**: Méthodes mettant à jours le vecteurs des cases à vider. Il retourne un entier représentant le nombre de case ajoutées au vecteur.Lorsque aucune case à été ajoutée, nous pouvons terminer une certaine boucle.
- **void Matrice::updateOnClick(Position p1)**: La classe matrice possède en attributs deux objets positions étant mis à jours lorsque le joueur sélectionnes les deux cases à échanger. Une fois les case sélectionnées cette méthode met à jours plusieurs élément au sein de la matrice
- **void Matrice::fillVoid()**:Méthode se chargeant de remplir les vides laissés dans la matrice selon un algorithme simple. On parcours la matrice de bas en haut, et à chaque vide on échange la cellule avec sa voisine du dessus à l'aide de l'appel à *void Cell::swap()*. Si il n'y à pas de cellule au dessus, alors la cellule est remplie avec un bonbon aléatoire.

5.3 Classe *Canvas*

La classe Canvas permet de dessiner la matrice grâce à des objet *Rectangle* et *Circle*. Elle reprend en partie le principe du patron de conception MVC et comporte donc un vecteur de forme qui doit être mis assez souvent à jour.

- **void Canvas::update(std::shared_ptr ¡Matrice¿ mat)**: A chaque mise à jours de la matrice, cette méthode va passer en revue via un pointeur intelligent le contenu de cette matrice pour mettre à jours les formes.
- **Position Canvas::mouseClick(Point mouseLoc)**: Pour chaque formes dans le vecteurs, vérifie que le clique est effectué sur la forme puis appelle la méthode *BasicShape::on_click()*.

6 Logique du Jeu

Lorsque le jeu est compilé puis lancé, différents objets sont crées et nous rentrons dans une boucle.

6.1 Construction du plateau

Un objet *Matrice* de taille fixée est crée. Comme spécifié plus tôt, la matrice est un objet composé d'objets *Cell* étant composé de pointeur vers *Item*. Lorsque nous créons donc un objet Matrice de taille n, il y à n^2 objet Cell qui sont crée ainsi que n^2 objet Item de couleur aléatoire.

6.2 Création de la fenêtre

Une fois la matrice générée, un objet *MainWindow* est crée et lié à cette matrice ou moyen d'un pointeur intelligent. La création de l'objet MainWindow implique la création d'un objet *Canva* comprenant un vecteur de forme tenu à jours à l'aide du patron de conception de l'observateur. Une fois la fenêtre affichée la boucle principale est lancée et la fenêtre attends les entrées de l'utilisateur pour pouvoir avancé dans le programme.

6.3 Gestion des entrées

A chaque clique de l'utilisateur, la position de ce dernier sur le canvas est convertie en coordonnée de matrice et envoyé à l'objet matrice (d'où l'importance de séparé les structures Position et Point). Le clique est vérifié, et si celui-ci est valide, la coordonnées est retenue. Une fois deux coordonnées valides retenues, nous passons à l'étape suivantes

6.4 Vérification des coordonnées matrice

Lorsque les deux coordonnées ont été retenue et vérifiées, une série d'autre vérification sont effectuées. Nous vérifions que, d'abord les deux cellules sélectionnées soient voisine, et ensuite nous vérifions qu'elles soient interchangeables. Si ces conditions sont remplies, les cellules sont inter-changées.

6.5 Suppression des triplés (ou plus)

Une fois les cellules inter-changées, un algorithme passe en revue les triplés de la matrices. Pour chaque coordonnées de la matrice, cet algorithme vérifie la couleurs des cellules voisines (haut gauche droite bas). Si le contenu des cellules sont de même couleurs, alors les coordonnées sont ajoutées à un vecteur. Après le passage en revue de la matrice, toute les cellules correspondantes aux coordonnées du vecteur sont vidées.

6.6 Remplissage des cellules vides

Une fois les cellules vidées, un autre algorithme passe en revue les cellule vide, et décide de les échangés avec leurs voisines du dessus. Si les cellules sont sur la ligne la plus haute, alors elle sont remplies aléatoirement.

6.7 Répétition

Même après un premier passage de la méthode du point 6.5, il se pourrait que le hasard rajoute à nouveau des triplés ou plus. Nous répétons donc les algorithmes du 6.5 et 6.6 tant que des cases sont ajouté au vecteur de suppression

7 Patron de conception MVC

Le patron de conception est une méthode de structuration d'un code orienté objet qui vis à séparer les différentes parties d'un programme (ici le modèle la vue et le contrôleur). Dans notre programme cette séparation est faite juste au niveau modèle et vue en séparant la logique de l'interface graphique. En effet, notre classe nous disposons de deux dossier réservés à chacune des tâche et seule les classes *Matrice* (alias le modèle) et *MainWindow* (alias la vue) communiquent entre elles. Cela est possible grâce à l'ajout d'un attribut à la classe *MainWindow* de type pointeur intelligent vers matrice. De plus, nos adoptons le patron de conception d'observation où la classe *MainWindow* agit comme observateur de la classe *Matrice* qui se charge de lui indiqué quand cette dernière est modifiée afin de mettre à jour le canvas.