

# Projet de Langage de Programmation: Sokoban

Andrius Ezerskis & Moïra Vanderslagmolen

January 12, 2023

# 1 Introduction

Nous avons séparé ce rapport en plusieurs parties. Tout d'abord, nous présentons les tâches accomplies. Ensuite, nous verrons les différentes classes implémentées et nous les expliquerons. Par après, nous parlerons de la logique du jeu, nous décrirons le déroulement d'un début de jeu, d'une fin de jeu, de l'enregistrement des steps et enfin des widgets. Pour finir, nous discuterons du modèle MVC dans notre projet.

## 2 Tâches Accomplies

Nous avons accompli 10 tâches. Nous avons implémenté :

- Les boîtes de couleur
- Les cases de téléportation
- Les boîtes légères
- Le compteur de steps
- Le meilleur score de steps
- L'écran d'accueil
- Les niveaux et sélection de niveaux
- La limite de steps
- Le déplacement automatique à la souris
- La détection d'échec

## 3 Classes

Nous avons séparé nos classes en trois fichiers distincts : Model, Controller et Vue.

### 3.1 Classes du Modèle

#### 3.1.1 BoardModel

BoardModel est la classe qui s'occupe de gérer la logique du plateau. Elle fait bouger le personnage, compte le nombre de pas, détermine si la partie est terminée ou si elle ne peut plus être résolue. Elle peut aussi téléporter le personnage d'une case de téléportation à une autre et charger les niveaux en mémoire et s'occupe de sauvegarder le meilleur score en mémoire.

Nous avons mis les niveaux dans des fichiers, nous avons donc aussi implémenté une fonction qui permet de vérifier si le plateau de jeu est fonctionnel. Nous vérifions si le plateau

de jeu a bien un joueur, si les téléportations ont bien une case arrivée et si le plateau de jeu a des lignes de même taille.

Les niveaux sont affichés comme ceci : la première ligne indique la limite de pas. Elle affiche un l et un nombre ensuite. La deuxième ligne indique le meilleur score, le fichier affiche un m suivi du meilleur score.

### **3.1.2 LogicCell**

Nous avons implémenté deux structures : Player et Box, que nous avons placé dans le fichier de LogicCell. Player contient seulement deux attributs, x et y, qui représentent la ligne et la colonne dans laquelle est placé le player. Box contient trois attributs, la couleur de la boîte, un booléen indiquant si elle est légère et un autre booléen indiquant si la boîte est bloquée.

Cette classe représente une cellule ou une case. Elle a une position, et contient un joueur ou une boîte. La case a aussi une couleur et un type. Elle peut avoir 4 types différents (mur, vide, téléportation, position finale d'une boîte).

Nous avons écrit une méthode qui permet de vérifier si la case est "complète". En effet, si le type de la case est la position finale d'une boîte, alors elle est complète que si elle possède une boîte et si cette boîte a la même couleur que la cellule qui la contient. Elle est aussi complète lorsque le type n'est pas la position finale d'une boîte. Cette fonction est utile lorsque nous voulons vérifier si le joueur a terminé son niveau, nous itérons alors dans le vecteur contenant toutes les cases et nous vérifions si chacune des cases est complète. La cellule est bloquée si son type est un mur ou si la boîte sur la case est bloquée.

### **3.1.3 Téléportation**

La classe téléportation prend deux LogicCell en paramètre, la case de départ et la case d'arrivée. Elle contient aussi une méthode qui prend un tuple d'int (la position du joueur), et qui renvoie un tuple de la position de la case d'arrivée de la téléportation si le joueur est sur une case de téléportation ou un tuple de -1 sinon.

## **3.2 Classes de la Vue**

### **3.2.1 CellDisplay**

Cette classe s'occupe de dessiner chaque cellule en fonction de son type. Elle contient aussi une méthode qui renvoie la position de la cellule si elle contient la position de la souris.

### **3.2.2 DisplayBoard**

Le DisplayBoard va itérer à travers le vecteur de LogicCell et créer des instances de CellDisplay. Enfin, lorsque l'utilisateur clique sur le DisplayBoard, il va demander à chaque

instance de CellDisplay si elle contient la position de la souris et renverra le résultat au MainWindow.

### **3.2.3 HelpWindow**

Cette fenêtre s'ouvre quand l'utilisateur appuie sur le bouton Help. Elle affiche toutes les commandes disponibles. Cette fenêtre permet aussi d'afficher les niveaux disponibles, lorsque l'utilisateur appuie sur le bouton Levels.

### **3.2.4 StartWindow**

Cette fenêtre permet d'afficher un écran d'accueil et affiche le nom des auteurs. C'est la première fenêtre qui s'affiche et disparaît après 10 secondes.

## **3.3 Classes du contrôleur**

### **3.3.1 MainWindow**

Cette classe sert de contrôleur. En effet, elle s'occupe de faire le lien entre le modèle et la vue. Elle va aussi dessiner les limites de pas, le compteur de pas et le meilleur score. Cette classe gère aussi les commandes entrées par l'utilisateur et s'occupe aussi des callback de chaque bouton. Chaque callback change l'information dans le modèle et met à jour le display en fonction. Il affiche aussi la fenêtre principale du jeu.

## **4 Logique du jeu**

### **4.1 Démarrer le jeu**

Lorsque nous démarrons le jeu, nous donnons un fichier au BoardModel, qui va créer un vecteur de LogicCell, vérifier si le niveau est fonctionnel et enregistrer la limite de pas ainsi que le meilleur score dans une variable.

Ensuite, nous allons créer une instance de HelpWindow, et nous les passons ensuite en paramètre à MainWindow. MainWindow va dessiner tous les widgets du jeu(bouton d'aide, changer de niveau, recommencer le niveau et remettre à 0 le meilleur score). MainWindow va aussi dessiner le nombre de pas, la limite de pas ainsi que le meilleur score. Le DisplayBoard, créé dans mainWindow, va dessiner toutes les CellDisplay une par une grâce au vecteur de LogicCell créé dans BoardModel.

### **4.2 Jouer**

La classe MainWindow s'occupe de gérer chaque événement. Par exemple, lorsque l'utilisateur appuie sur la flèche de droite et que la partie n'est pas terminée, la classe MainWindow va tout d'abord appeler la classe BoardModel, pour lui indiquer que le joueur veut aller à droite. Le BoardModel va mettre à jour son vecteur de LogicCell et le nombre de

pas. MainWindow va demander à DisplayBoard de se mettre à jour, et va donc recréer un nouveau vecteur de CellDisplay grâce au vecteur de LogicCell de BoardModel. MainWindow va aussi redessiner les pas.

Si l'utilisateur clique sur le plateau de jeu, MainWindow va demander à DisplayBoard quelle est la position de la case cliquée. Celui-ci va demander à chaque CellDisplay si la position de la souris est dans cette CellDisplay. Si la CellDisplay renvoie true, il envoie le tuple de cette Cell à MainWindow, qui enverra ce tuple à BoardModel. BoardModel va mettre à jour le vecteur de logicCell et la position du joueur. MainWindow vérifie ensuite si la partie est terminée ou pas.

Si la partie est terminée, les événements liés au plateau de jeu ne sont plus reconnus par MainWindow et celui-ci affichera un message indiquant à l'utilisateur s'il a gagné ou perdu.

### **4.3 Widgets**

#### **4.3.1 Help**

Si l'utilisateur clique sur le bouton Help, une nouvelle fenêtre va s'ouvrir. Nous mettons l'attribut de help à true, pour indiquer à helpWindow d'afficher l'aide. Ensuite, nous appelons la fonction show de Fl\_Window sur l'instance de HelpWindow.

#### **4.3.2 Levels**

Si l'utilisateur appuie sur le bouton changeLevel, une nouvelle fenêtre va être affichée grâce à la fonction show de FLTK. Tant que la fenêtre est affichée (la fonction shown de Fl\_Window), nous appelons la fonction Fl::wait. La fenêtre est modale, c'est à dire que l'utilisateur ne peut plus faire aucune autre action dans une autre fenêtre tant qu'il n'a pas cliqué sur le bouton pour choisir le niveau. Lorsque l'utilisateur a cliqué sur un des niveaux, le callback de ce bouton va enregistrer dans une variable quelle niveau a été cliqué et va fermer la fenêtre. Ensuite, MainWindow va demander à BoardModel d'enregistrer le meilleur score va récupérer l'information de HelpWindow et enverra l'information à BoardModel, qui va recréer un nouveau board. Enfin, MainWindow va demander à DisplayBoard de se mettre à jour.

#### **4.3.3 ResetMinPas**

Si l'utilisateur appuie sur le bouton pour remettre à 0 son meilleur score, MainWindow va appeler boardModel et mettre le nombre de pas à 0. BoardModel va écrire dans le fichier que le meilleur score est égal à 0. Pour écrire dans le fichier, il cherche m + l'ancien meilleur score. Lorsqu'il le trouve, il le remplace par m + le nouveau meilleur score. MainWindow va appeler sa fonction draw pour redessiner le meilleur score.

#### 4.3.4 ResetLevel

Pour recommencer un niveau, nous avons décidé de recharger le niveau à partir du fichier. Il agit donc comme un changement de niveau.

### 4.4 Quitter le jeu

Lorsque nous quittons le jeu, le callback s'occupant de fermer la fenêtre va enregistrer le meilleur score de l'utilisateur.

## 5 Modèle-Vue-contrôler

Nous avons implémenté le modèle MVC uniquement pour le board. Premièrement, nous avons d'abord essayé d'implémenter tout le jeu en modèle MVC, mais FLTK n'est pas adapté au modèle MVC. En effet, les callback des widgets sont statiques et nous étions donc obligés d'appeler notre contrôleur dans le MainWindow. Nous avons donc décidé que la fenêtre principale serait le contrôleur du board vu qu'il gère les commandes entrées par l'utilisateur et met à jour le boardModel et le display en fonction et il s'occupe aussi de l'affichage des boutons et de leur callback.

Nous avons aussi décidé par souci de performance d'implémenter la vue dans le contrôleur. En effet, nous avons implémenté une fonction qui rafraîchit le board 60 fois par seconde, et il n'était donc pas nécessaire d'avoir cette classe dans le contrôleur. Seulement, cela ralentissait beaucoup notre programme et le jeu clignotait. Nous avons donc mis DisplayBoard dans MainWindow. Lorsque nous avons une commande entrée par l'utilisateur, nous mettons à jour le plateau de jeu (le modèle) et nous appelons la fonction update de display pour lui dire de se mettre à jour.

## 6 Conclusion

En conclusion, nous avons implémenté les 10 tâches requises, nous avons écrit plusieurs classes, séparés dans des fichiers différents et dans trois dossiers différents : Modèle, Vue et Contrôleur. Nous avons essayé de respecter le modèle MVC, mais par souci de performance, nous avons décidé de mettre la vue dans le contrôleur, pour qu'il puisse être notifié des changements du boardModel.