Projet : Plumber

Programmation en Java

Gloria BENOIT & Eliott TEMPEZ
M1 BI-IPFB

Université Paris Cité
BQAAY190 Programmation avancée
10/01/2024

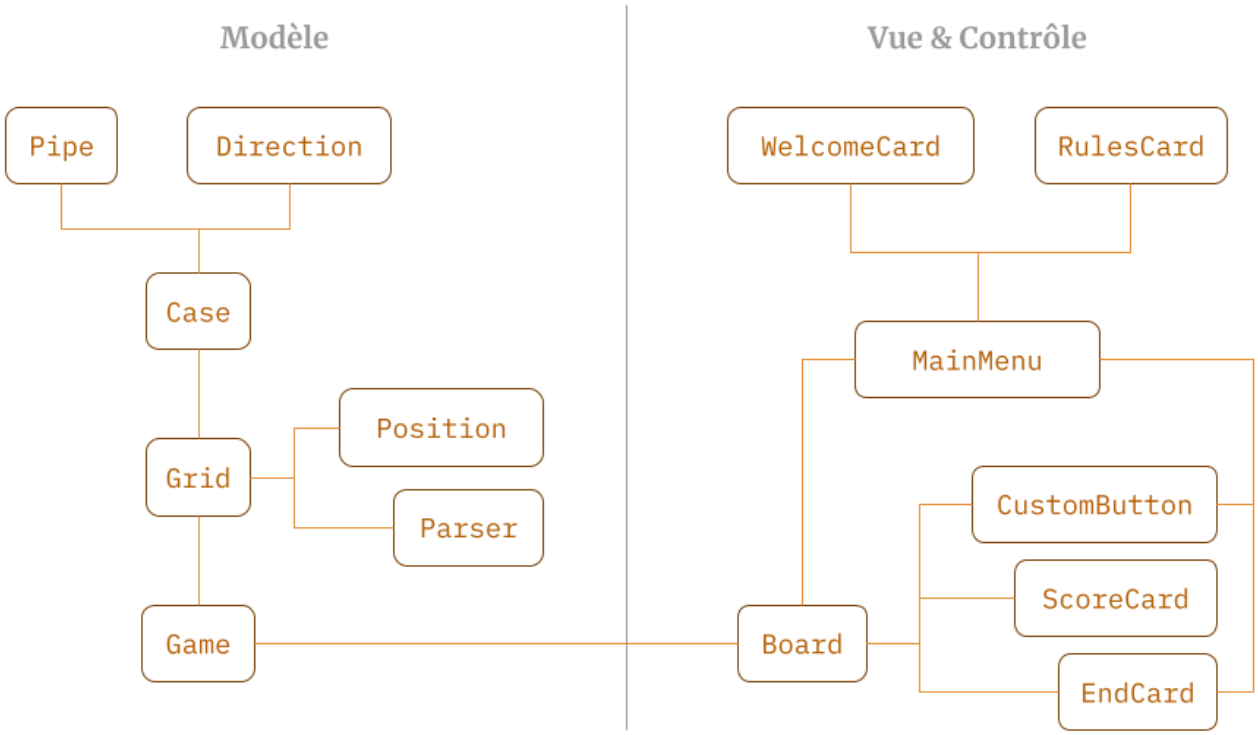
Introduction

Notre projet de programmation en Java consiste à l'implémentation du jeu *Plumber*, avec affichage, de manière à pouvoir y jouer uniquement avec la souris.

Le but du jeu est de relier une série de tuyau de manière à ce qu'ils soient tous connectés entre eux, sans qu'aucun ne soit ouvert sur la bordure du plateau.

Dans ce compte-rendu, nous allons détailler les différentes classes mises en places pour construire le jeu mais aussi son affichage. Dans un premier temps, nous verrons son implémentation minimale, puis dans un second, les différentes extensions que nous avons ajoutées.

Architecture



Implémentation minimale

Modèle

Pipe: La classe *Pipe* est une énumération des différents types de tuyaux retrouvés dans le jeu. Chaque tuyau possède une forme définie, décrite par le nombre d'ouvertures qu'elle possède ainsi que ses emplacements. Pour cela, chaque type de tuyau a comme attributs le nom de la forme du tuyau, ainsi qu'un tableau de 4 booléens correspondant à la présence ou absence d'ouverture au nord, à l'est, au sud, et à l'ouest respectivement. Cette classe contient des méthodes d'accession, permettant d'obtenir les différentes informations caractérisant le tuyau (son nom et si il est ouvert dans une certaine direction par exemple).

Direction: La classe *Direction* est une énumération des différentes directions rencontrées. Celle-ci associe un point de coordonnées spécifique au nord, à l'est, à l'ouest, et au sud. Ces valeurs permettent ainsi de déterminer dans quelle direction pointe l'ouverture d'un tuyau par exemple. Elle comprend des méthodes d'accession à ces coordonnées, aux coordonnées correspondant à la direction inverse, ainsi que des méthodes permettant de gérer les rotations horaires et anti-horaires.

Case: La classe *Case* représente une case individuelle sur la grille du jeu. Elle comprend trois descripteurs : le tuyau présent sur la case, la direction d'entrée de l'eau dans la case, et un booléen correspondant à la présence d'eau ou non au sein de la case. Cette classe possède des accesseurs, des méthodes permettant d'effectuer des rotations, ainsi que des

méthodes permettant de réinitialiser ou d'activer la présence d'eau.

Position: La classe *Position* représente les coordonnées d'une case au sein d'une grille ; elle a comme descripteurs le numéro de la ligne et de la colonne correspondante. Elle comprend des accesseurs, ainsi qu'une méthode de "voyage" dans une direction donnée. On l'utilise de manière à regarder la case la plus proche sur laquelle on peut se déplacer (qui est connectée à notre case actuelle).

Grid: La classe *Grid* représente la grille du jeu. Elle est caractérisée par une hauteur et une largeur, ainsi qu'un tableau de cases de dimension hauteur * largeur. Cette classe permet la gestion des mécanismes de jeu par le biais de plusieurs méthodes :

- *shuffleGrid()* permet de mélanger aléatoirement les directions des tuyaux de chaque case
- *rotate(i, j)* et *invRotate(i, j)* permettent d'effectuer un quart de tour dans le sens horaire ou anti-horaire pour une case en particulier
- *resetFlow()* réinitialise le courant d'eau dans la grille, permettant de mettre à jour le courant d'eau à chaque tour effectué
- *initPropag()* initialise la propagation du courant d'eau à partir du centre de la grille, puis fait appel à *propagation()*, qui propage ce courant dans les cases connectées à une case comportant de l'eau, de manière récursive. Si la hauteur et/ou la largeur de la grille est pair, le centre de la grille est la valeur arrondi supérieur à l'entier du milieu réel du plateau
- *win()* vérifie si les conditions de victoire du jeu sont remplies et renvoie le booléen correspondant. Elle vérifie donc qu'aucune case ne soit ouverte sur du vide et que tous les tuyaux soient connectés ensemble en passant par le centre

Parser: La classe *Parser* est responsable de la création d'un niveau de jeu à partir d'un fichier. Pour cela, elle prend en entrée le fichier explicitant la solution à ce niveau et construit la grille correspondante. Elle renvoie ainsi la grille de solution au niveau, que l'utilisateur doit retrouver pour gagner.

Game: La classe *Jeu* permet d'assembler toutes les autres classes pour rendre le jeu fonctionnel. Ses champs sont la grille de jeu, la grille de solution du jeu, statique, un booléen qui indique si le jeu est fini ou non, les fichiers de niveaux, le numéro du niveau actuel et des informations sur le retour arrière. Son constructeur permet de lire un niveau, en créer la grille et la solution puis mélanger la grille et lancer la première propagation.

Elle contient différentes méthodes : jouer un tour de jeu, annuler le tour précédent, mélanger la grille actuelle et transformer la grille actuelle en sa solution. La méthode *play()* renvoie un entier, ce qui permettra d'afficher différents messages selon les coups effectués : -1 si on est hors du plateau, 0 si la case visée est vide, 1 si le tour joué est valide, 2 si le jeu est fini pour la première fois et 3 si le jeu est déjà fini.

Vue et Contrôle

CustomButton: Cette classe est une extension de *JButton* qui permet de changer dynamiquement la couleur du bouton. Tous les boutons utilisant une couleur spécifique, cela permet de raccourcir le code, en spécifiant directement à la création du bouton sa couleur. Il est possible de créer un *CustomButton* en spécifiant la taille de la police à utiliser.

MainMenu: Cette classe est le menu principal du jeu. C'est une extension de JPanel, qui affiche une banière d'introduction (grâce à *WelcomeCard*, expliquée plus bas) en haut de la fenêtre, les boutons de niveaux au centre et les boutons utilitaires en bas, qui sont les règles du jeu (grâce à *RulesCard*, expliquée plus bas) et le bouton de sortie. Elle reste affichée tout au long du jeu, lorsque les différents niveaux sont choisis.

Le bouton de *Règles* affiche une nouvelle fenêtre, qui disparait uniquement si fermée. Cela permet de pouvoir lire les règles en même temps que de jouer au jeu, pour être sûr de bien jouer.

Les constructeurs des différents affichages prennent tous en entrée une couleur, ce qui permet d'unifier les différentes fenêtres et de modifier directement l'identité visuelle du jeu. Les couleurs du jeu sont des champs statiques finaux de cette classe.

Nous avons construit deux niveaux en plus de ceux fournis : les niveaux 1 et 2. Tous les niveaux utilisés ont été rangées selon la difficulté, calculée selon le nombre de cases total du plateau.

WelcomeCard: La classe *WelcomeCard* est une extension de JPanel qui permet d'afficher un message de bienvenue sur notre jeu.

RulesCard: La classe *RulesCard* est une extension de JPanel qui permet d'afficher les règles du jeu. La séparation du texte a été effectuée manuellement, n'ayant pas trouver de technique simple à utiliser pour le faire automatiquement.

Board: La classe *Plateau*, extension de *JPanel*, permet l'affichage des niveaux. Elle possède de nombreux champs, distingués en plusieurs groupes :

- Dimensions la fenêtre
- Stockage des sprites
- Spécificités du niveau
- Animation de la rotation
- Retour arrière
- Spécificités de l'affichage

Son constructeur permet d'afficher une fenêtre selon la taille du niveau, en ajoutant un espace en haut de la fenêtre, dédié au différents boutons nécessaires. On en compte quatre : *Retour au menu*, *Retour arrière*, *Affichage de la solution* et *Redémarrage du niveau*.

Le bouton de *Retour au menu* est en blanc, pour se distinguer des autres, qui ont une couleur spécifique, trouvée dans les champs du plateau. Cette couleur est *static* et *final* car commune et inchangée dans tout le plateau.

Le bouton de *Retour arrière* a une spécificité, il apparaît en gris lorsque le retour en arrière est impossible : aucun tour n'a encore été effectué, la solution a été affichée, le niveau a été redémarré ou tous les coups effectués ont été annulés. Cela permet de traduire visuellement son utilisation : s'il est utilisable, le joueur le comprendra. Cela est rendu possible par la classe *CustomButton*, expliquée plus haut.

Dans le constructeur, les différents sprites des tuyaux sont stockés dans un tableau de format $3x5x4$, spécifique au fichier de sprites fourni, à l'aide d'une méthode construite. La troisième dimension permet de stocker les différentes rotations de chaque tuyau, et l'utilisation d'une méthode externe permet d'utiliser d'autres fichiers de sprites, tant que leur structure globale est la même.

À chaque mise à jour du plateau, la grille de jeu est parcourue et les tuyaux sont affichés de nouveau. Cela permet de suivre à chaque tour le courant d'eau, et donc l'avancée du jeu. Les différents tuyaux ont une taille initiale de $120x120$, mais cela est réduit à $100x100$ de manière à mieux localiser le clic de souris. Ainsi, à chaque clic, une rotation est effectuée sur le tuyau correspondant. Le clic est valide ou non, selon les résultats de *play()*, spécifiés précédemment, et le type d'action est affiché en ligne de commande. Un coup valide affiche la grille, tout autre coup affiche un message spécifiant pourquoi le coup n'est pas accepté. La rotation s'effectue avec 5 degrés dans le sens horaire toutes les 5 milisecondes. Si une case est en mouvement, il ne sera pas possible de tourner une autre.

Si le niveau est réussi, une nouvelle fenêtre est affichée en bas de l'actuelle. Elle est construite à l'aide de la classe *EndCard*, explicitée plus bas. Quand le jeu est fini, les coups ne sont plus acceptés. La rotation est toujours affichée mais la case n'est pas tournée. Nous avons décidé de garder l'affichage de la rotation, pour traduire que la case est bloquée. Il est néanmoins possible d'utiliser les boutons *Solution* et *Redémarrage*. Si le niveau est redémarré, la fenêtre de fin persiste malgré tout. Il faudra revenir au menu pour refaire le niveau sans cette fenêtre.

À droite du plateau, on affiche le score a l'aide de *ScoreCard* (expliquée plus bas). Le score est initialisé à 0 depuis *MainMenu*, et est incrémenté directement depuis l'appel du niveau suivant dans *EndCard*. Cela permet de réinitialiser le score à 0 lorsque l'on revient au menu.

ScoreCard: Cette classe, extension de *JPanel*, permet d'afficher le score actuel.

EndCard: Cette classe est une extension de *JPanel*. Elle permet d'afficher une fenêtre de fin de niveau, spécifique au niveau actuel. En effet, elle affiche un message de réussite de niveau et un bouton de passage au niveau suivant pour tous les niveaux, sauf le dernier. Si le niveau actuel est le dernier, elle affiche un message de fin de jeu et un bouton de retour au menu.

Sa couleur, les différents niveaux, le niveau actuel et le score actuel sont spécifiés dans son constructeur. La couleur, le numéro de niveau actuel et le numéro du dernier niveau sont stockés de manière statique dans ses champs, pour les utiliser à tout niveau du code et ainsi changer le rendu selon le niveau considéré. Comme expliqué précédemment, la présence du score dans le constructeur de cette classe permet de l'incrémenter si l'on passe au niveau suivant.

On obtient deux messages différents selon si le dernier niveau a été réussi avec un score maximal ou non.

Extensions mises en places

Cases vides

Nous avons implémenté l'utilisation de cases vides, spécifiée par un point (*'.'*). Lorsqu'une grille est lue, avec la classe *Parser*, une case vide prendra comme initiale un point, et comme direction l'Est. Cette direction est aléatoire et n'aura pas d'effet sur l'utilisation de cette case. En effet, dans le classe *Pipe*, la case vide n'a aucune ouverture, sa direction n'est donc pas importante puisqu'il sera impossible de l'atteindre de toute manière. Cela permet de bloquer la propagation sur cette case. Il est aussi impossible de tourner cette case. En effet, comme dicté précédemment, *play()* empêche de jouer sur une case vide.

Par défaut, une case vide est considérée comme recevant le courant d'eau. Cela permet de garder les mêmes conditions de victoire, en spécifiant dans *win()* qu'une ouverture sur une case vide est équivalente à une ouverture sur du vide.

Le niveau 2 est un exemple de niveau utilisant une case vide.

Historique

Nous avons aussi implémenté la possibilité d'effectue un retour arrière des coups effectués. Pour cela, on stocke les coordonnées de chaque coup dans un tableau de *Position*, et on utilise une méthode pour inverser la rotation, trouvée dans *Direction*. Le tableau fonctionne comme une pile, avec un compteur permettant de savoir où placer le prochain coup. Le tableau double de taille si jamais il est totalement rempli est un coup est ajouté, permettant de ne pas allouer trop de mémoire si jamais il n'est pas utilisé.

La classe *Game* possède un champ booléen *possibleUndo* qui, une fois lu, permet d'empêcher d'annuler des coups non existants et de continuer à changer dynamiquement l'apparence du bouton.

Si le niveau est redémarré, l'historique des coups est réinitialisé, de manière à ne pas annuler des coups effectués sur une version antérieure du plateau.

Nous avons décidé de ne pas implémenter le retour en avant, mais il suffit de garder en mémoire l'indice du dernier coup joué et de remonter le tableau. Ici, quand on annule un coup, on ne l'efface pas du tableau, ce qui permettrait donc l'implémentation du retour en avant.