Puzzle Edition

Rapport de projet

VINCENT Leo 21805239
BELLEBON Alexandre 21808613
LEROY Clementine 21800424
DEROUIN Auréline 21806986

Table des matières

1 Présentation du projet **1**

A Présentation de l'application 1

2 Architecture du projet **2**

A Arborescence du projet 2

3 Aspects techniques **3**

A Librairie piecesPuzzle 3

B Modèle : Le plateau 3

C Controleur 4

i Play 5

D Vue 7

i InterfaceGraphique 7

ii ActionGraphique et MouseClicke 7

4 Conclusion **9**

5 Ressources utiles et sources utilisés **10**

1 Présentation du projet

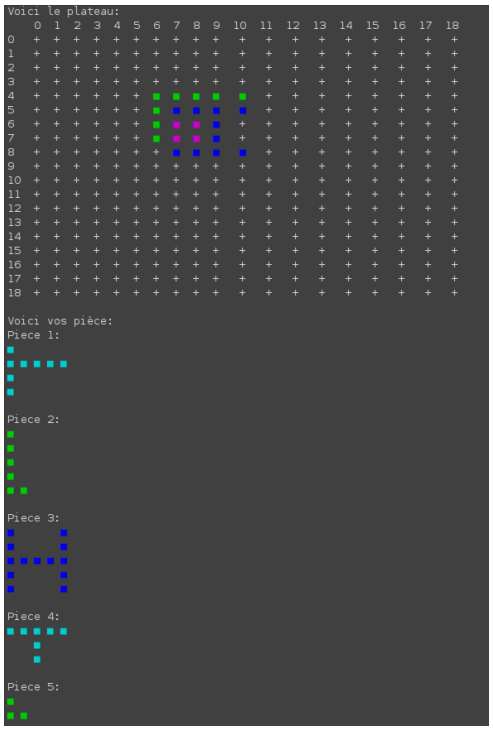
A Présentation de l'application

Puzzle Edition est une application de jeu, dotée d'une interface graphique, qui consiste à assembler des formes de sorte qu'elles occupent le moins de place possible. Cela s'apparente au tetrix, mais avec des modalités différentes. D'ailleurs, le joueur retrouvera certaines pièces de ce jeux. Ces pièces peuvent être placer, déplacer ou supprimer du plateau ou encore tourner.

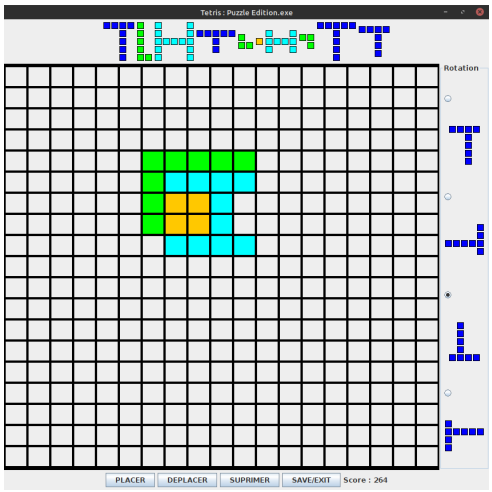
Le joueur peut créer ou charger une partie. La difficulté de la partie est mesurer en fonction de la taille du plateau choisis ainsi que le nombre de pièces générés. En effet, plus il y a de pièces généré, plus la partie est difficile, mais plus le score sera élevé. C'est pour cela qu'à chaque partie, il est possible soit de demander à obtenir une nouvelle configuration de départ, soit charger une configuration déjà créée et sauvegardée.

Le but du joueur est de minimiser l'espace occupé par l'ensemble des pièces. Plus précisément, la fonction d'évaluation sera l'aire du plus petit rectangle (parallèle aux axes) recouvrant l'ensemble des pièces. Lorsque le joueur considère avoir terminé (ou lorsque le nombre maximum d'actions autorisées est atteint), il clique sur un bouton est son score est alors calculé.

Une option permet de faire jouer l'ordinateur sur une nouvelle partie, ou une partie déjà sauvegarder, permettant ainsi de comparer son score avec celui de l'ordinateur.



Vue console



Vue graphique

2 Architecture du projet

A Arborescence du projet

src : Arbo

A REVOIR

3 Aspects techniques

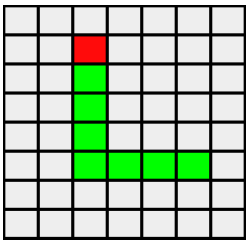
A Librairie piecesPuzzle

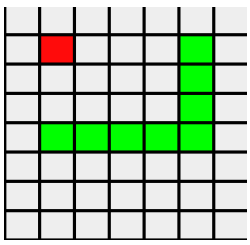
Pour la conception de ce projet, une librairie a été créer, réferencant ainsi toutes les pièces possibles. Cette librairie comporte 4 pièces différentes. Ces quatres pièces ont toutes une grille créer a partie de la rotation défini. Cette grille et cette rotation est créer à l'aide de la méthode *createGrid()* qui prend en paramètre un numéro qui correspond a la rotation dans le sens horaire. Cette méthode change la largeur/longueur de la pièce, en fonction de celle données de la pièce de départ, permettant ainsi la création d'une nouvelle grille en prenant en compte la rotation. Chaque pièce a une grille composé de valeur boolean, permettant ainsi de créer la pièce voulu (valeur "true" pour indiquer que la case est pleine), en fonction de sa rotation, largeur, longeur. Elle est aussi composé de coordonnées, permettant, en fonction du jeu, de pouvoir situé la pièce dans le plateau.

B Modèle : Le plateau

La librairie piecesPuzzle est utiliser par notre modèle. PlateauPuzzle permet de créer un plateau ainsi que d'utiliser les pièces pour les ajouter, supprimer... Pour la création de notre plateau, nous avons utiliser une HashMap de coordonnées (ArrayList de Integer) en clé et d'une pièce, s'il y en a une, ou "null" en valeur. Cela permet donc d'avoir une lisibilité absolut sur le plateau, ainsi que sur les différentes pièces présentes, sans être obliger d'avoir une hashmap trié. La HashMap est directement créer dès l'appel à cette classe.

Le plateau peut donc créer, placer, supprimer, déplacer, tourner la pièce sur le plateau. Pour la plupart des différentes méthodes, un appel de la méthode *validePlacement()* est nécessaires afin de savoir si la pièce, reçu par la méthode, peut être placer dans le HashMap. Pour cela, dès que la valeur de la grille parcouru est de boolean *true*, elle regarde si la valeurs de la clé coordonnées du HashMap (en fonction des coordonnées reçu par la méthode). Cela permet donc de pouvoir placer des pièces cote a cote, en ne placant que les case "true" de la pièce dans la HashMap. Les pièces ont donc des cases "true" et "false" en fonction de comment elles sont créer. Nous avons choisit une case de référence afin d'avoir les coordonnées de la pièce s'il faut, par exemple, la déplacer ou la faire tourner, ou encore, si on sauvegarde la partie. Cette case de référence, nous l'avons choisis en haut à gauche de la pièce. Sauf que cela posait problème si le joueur voulait posé la pièce a l'endroit où il a cliquer/indiquer les coordonnées (voir image**??**). En effet, la case est toujours en haut a gauche, même si cette case est en false.





Légende :
Rouge : coordonnées où le joueur veut placer la pièce, et indique aussi la case de référence
Vert : pièce L en rotation 0 (img 1) et rotation 3 (img 2)

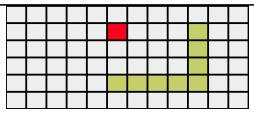
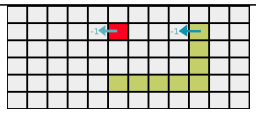
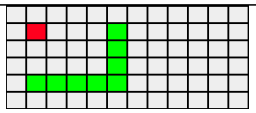
Pour régler ce problème, nous avons rajouter un changement de coordonnées, dès que le joueur indiquait, par un clique ou par indication de coordonnées en console, la case où il voulait placer la pièce.

Tant que les coordonnées Y de la grille de la pièce était false, les coordonnées (*coo*) en position Y était diminuer de -1, permettant ainsi de placer la grille de la pièce a partir des coordonnées indiqué -1, si la case était "true" .

Algorithm 1: finished() :boolean

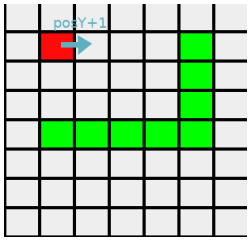
```
1 posY ← 0;
2 xx ← 0;
3 yy ← 0;
4 while !piece.getGrid()[xx][yy+posY] do
5 |   coo.set(1,coo.get(1)−1);
6 |   posY ← posY+1;
7 end
8 return true
```

Cela permet de changer la "case", donc les coordonnées indiquer par le joueur, jusqu'à ce que la valeur des coordonnées de la grille de la pièce est égal a "true".

		
Pris en compte des coordonnées	Boucle while	Application de l'ajout ou déplacement de la pièce

Légende :
Rouge : Case de référence et variable "coo" qui, sur l'image 1, sont les coordonnées renseigner par le joueur (modifier par la suite dans la while)
Flèche : coo modifier dans la while
Vert/jaune : Pièce "fantome" où la pièce devrait être normalment sans cette while

Cela permet de changer la "case", donc les coordonnées indiquer par le joueur, jusqu'à ce que la valeur des coordonnées de la grille de la pièce est égal a "true".
Pour cela, pendant que la valeurs des coordonnées Y de la variable *coo* diminuent, le parcours de la grille se fait jusqu'à ce que elle rencontre la case plus en haut a gauche possible de la pièce. Donc elle ajoute +1 a la variable *posY*.



Légende :
Rouge : Case de référence
Flèche : vérification de la prochaine case si sa valeur est "true"
Vert : pièce L rotation 3

C Controleur

Le controleur est séparrés en plusieurs classes :

classe *PlayMenu* : Permet de choisir entre la vue console et la vue graphique.

classe *Play* : Fait appel aux méthodes du modèle,et appels les méthodes de "choix" en fonction de la vue/joueur actuel.

classe *InterfacePlay* : Méthodes communes aux choix des joueur : ia ou joueur.

classe *PlayJoueur* : Méthodes de choix pour joueur Console, grâce aux Scanner.

classe *PlayIa* : Méthodes de choix pour ia.

enum *EnumAction* : Enumère les actions possibles en jeu.

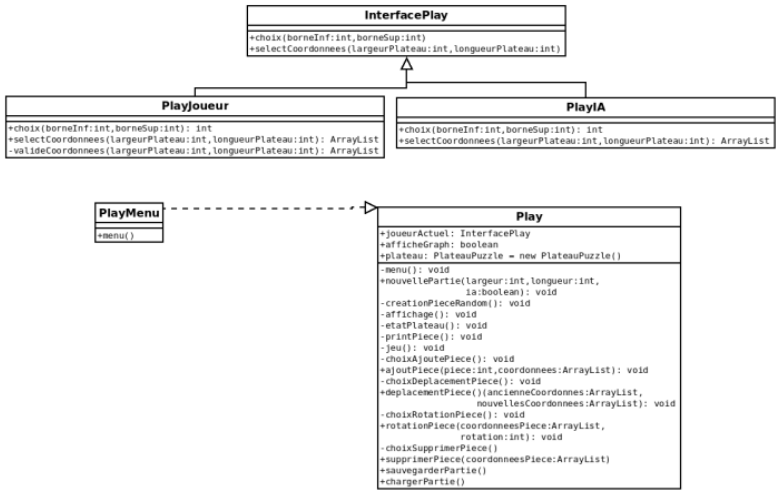


FIGURE 3.1 – Controleur

Cette séparations permet simplifier et séparer les choix de jeu. Cela évite la redondance de code.

i Play

En vue console, les choix se font grâce à la variable *joueurActuel* qui est une instance de *InterfacePlay*. Cela permet, en fonction du choix du joueur (option ia ou non), d'appeller les même méthodes de choix, permettant ensuite d'appeler les méthodes communes a toutes les vues pour accomplir l'action prévu (placer, déplacer...). Tout cela est controler par la méthode *jeu()*.

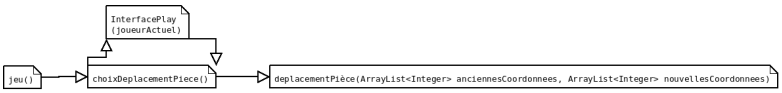


FIGURE 3.2 – Explication choix en fonction du joueur - vue console

En vue graphique, les choix se font grâce à des actions sur des cases/boutons, détecter par des Observeur/Observable voir...partie graph

Joueur Ia

Une méthode *jeuIa()* est appeler dès que le joueur choisis de regarder l'ia joué une partie chargé ou une nouvelle partie (en vue graphique ou console). Cette option permet donc de savoir la meilleurs composition de jeu. Cette méthode permet de lancer des *fourmis* (joueur ia qui joue une partie avec des choix random et guidées) afin de faire un certain nombre de partie. Le meilleurs score et le meilleurs plateau est enregistrer. Cela permet donc d'avoir plusieurs possibilités aléatoires, et d'essayer de trouver la meilleurs composition possible. Cela évite donc de n'avoir qu'une possibilité qui regarde jusqu'à une certaine profondeur.

La fourmis appel donc la méthode *jeu()*, commune si le joueur actuel est le joueur en vue console. Cette méthode permet de choisir l'action que le joueur va effectuer. C'est un choix, donc cette mathode appel l'attribut *joueurActuel*, permetant de faire un choix random pour l'ia. Pour éviter d'avoir un simple choix aléatoire où toutes les actions ont le même niveau de piorité, une liste d'actions a été créer avec plusieurs même actions à l'interieur, permettant de signifier qu'une action pour "avancer" (placer et déplacer) dans le jeu est plus important que de "rester sur place" (rotation dans le plateau et rotation des pièces a jouer) et est plus important qu'une action qui "recule" l'avancement du jeu (supprimer une pièce). Une fois l'action choisis, d'autres méthodes de choix sont appelés en fonction du choix de l'action.

Pour les actions pour "avancer" dans le jeu, les choix ne sont pas de simple random. En effet, si la fourmis a choisi l'action de PLACER ou DEPLACER une pièce, la méthode *choixDeplacement()* ou *choixAjout()* est appelé. Ces deux méthodes prenent en compte la largeur, la longueur et le plateau. Pour *choixDeplacement()*, l'ia choisis d'abord une pièce au hasard sur le plateau, grâce à la méthode selectPiece() (méthode commune de l'InterfacePLay) qui sélectionne une pièce dans la liste des pièces placer. Les coordonnées récupéré sont les coordonnées de la case de référence de la pièce (cf : voir **??**). Une boucle while, presque similaire à ici1, est donc nécessaire pour avoir les coordonnées qui aurait pu être cliquer ou indiquer par un véritable joueur. Une fois les coordonnées obtenus, la pièce est supprimer afin d'exercer une simulation de possibilité sans avoir une pièce sur ces cases dans le plateau. A partir de la, *choixDeplacement()* exerce comme *choixAjout()*. Elles créer une copie du plateau actuel, puis parcours tout le plateau à l'aide des coordonnées placé en clé du HashMap. Si la coordonnée est vide et au moins une pièce est situé à 1 ou 2 cases autour de celle ci, alors un test de placement est fait (grâce à la méthode *validePlacement()*B). Si ce test est réussi, un deuxième test est necessaire : faire un ajout de la pièce aux coordonnées et calculer le score grâce aux méthodes dans la classe *PlateauPuzzle*). Si le score est supérieur au score du test précédent, une sauvearde du score et des coordonnées est faites. Sinon, cela passe au test pour les coordonnées suivantes. Le *validePlacement()* est déjà appeller dans la méthode d'ajout ou de déplacement, mais cela diminue de temps d'execution en vérifiant le placement avant d'appeler la méthode d'ajout qui vérifie le placement et qui place si c'est possible.

Pour les autres actions (Rotation, Supprimer), le choix est défini en aléatoires.

Une fois que toutes les fourmis ont finit leurs parties, *afficheIa()* affiche le plateau ayant obtenus le meilleurs score. L'affichage déplace d'abord les pièces déjà placer sur le plateau initiale(si c'est une partie charger), puis place ensuite les pièces a jouer en fonction du plateau obtenus par les fourmis. Pour cela, elle utilise les méthodes communes pour effectuer des actions.

joueur Graphique

Si la vue graphique est choisis, une InterfaceGraphique, une MouseClicker ainsi qu'un ActionGraphique est créer a partir du constructeur de la classe *Play()*, permettant ainsi d'afficher la Vue. Ensuite, la méthode *menuGraph()* est appeler, permettant d'appeler *jeuIa()* ou *jeuVue* en fonction de ce que veux le joueur. Le choix se fait grâce aux différents boutons. Le joueur peut donc sélectionné la taille de son plateau (via le bouton ligne et colonne qui vont de 5 à 20), il peut également charger une partie, afficher le tableau des scores ou encore les règles du jeu. Le fonctionnement des boutons sont assez simple. Le controleur attend que ActionGraphique le notifie, pemrettant ainsi de récuperer le choix du joueur (via la méthode *getChoix()*) et ordonne ainsi à la vue ce qu'elle doit faire (afficher la grille, les scores, demander au joueur de selectionné une pièce,...).

FIGURE 3.3 – Bouton du menu principal

D Vue

La vue est composé de 3 classes :

classe *InterfaceGraphique* : Construit et affiche la vue graphique

classe *ActionGraphique* : Gère les interactions entre le joueur et la vue

classe *MouseClicker* : Permet de récupéré les clique souris du joueur sur la vue

i InterfaceGraphique

L'interface Graphique est composé de nombreuses méthodes pour afficher la vue. La méthode principale est *buildContentPane()*.Elle permet à la fenêtre, dans un premier temps, d'afficher le menu principale lorsqu'elle n'a pas de modèle. Ce menu est composé de divers bouton (cf : voir **??**). Et quand nous avons un modèle, on affiche la grille en parcourant le plateau en vérifiant la présence de pièce. Cette grille est une grande fenêtre de couleur noir composé de plusieurs petites fenêtre grise espacé entre-elles, donnant cette impression de grille. On oublie pas d'ajouter à cela les pièces à placer, les boutons de jeu et les rotations des pièces, nous obtenons ceci **??**. Cette class permet aussi l'affichage des scores, des explications de jeu, des aides, du formulaire pour rentrer son pseudo et des fenêtre de choix.

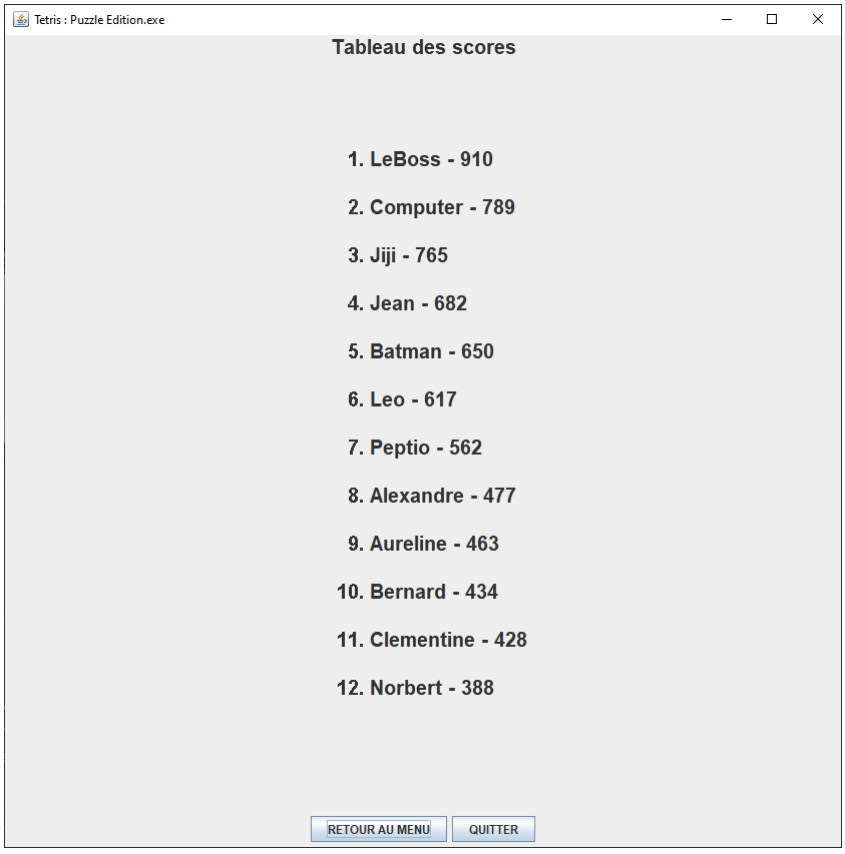


FIGURE 3.4 – Tableau des scores

ii ActionGraphique et MouseClicke

Ces 2 méthodes sont liés car MouseClicker est utilisé quand les méthodes d'interaction d'ActionGraphique sont appelés. La tache d'ActionGraphique est d'attribuer un rôle à chaque bouton et de gérer l'ordre des évènements quand un joueur veut réaliser une action. Par exemple, le bouton "PLACER" va

appelé la méthode *actionBouton()* pour vérifier si aucune autre action est en cours. Si aucune action est en cours, on appelle la méthode *placementPieceVue()* qui va demander au joueur de sélectionner une pièce. Il va attendre que MouseClicker lui envoie la pièce sélectionner pour ensuite demander au joueur de choisir sa rotation et son emplacement. Il va de nouveau attendre que MouseClicker lui envoie la case sélectionné pour pouvoir demander au contrôleur de placer la pièce dans la rotation souhaité via les méthodes vu précédemment. Pendant les 2 périodes où ActionGraphique attend, on vérifie si le joueur n'a pas décidé d'annuler son action, sinon on coupe l'attente est demande à la vue d'afficher son état initiale. Ce processus est semblable pour les méthodes *deplacementPieceVue()* et *supprimerPieceVue()*. Cette class gère aussi les listener des pièces et des case de la grille, pour que le joueur ne fasse pas de mauvaise sélection.

4 Conclusion

Ce projet était interessant dans l'ensemble. Nous en somme satisfait, même si l'application peut encore être amélioré, nous avons une vue console et graphique fonctionnelle ainsi qu'un joueur IA un peu plus performant que de simple choix aléatoire. Nous avons respecter, dans l'ensemble, ce qui nous avez été demandé, même si nous n'avons pas implementer de patern observeur/observable, nous avons un MVC correct et respecter.

5 Ressources utiles et sources utilisés