Rapport Modélisation

Mariem Abdellaoui et Sami El Yaghmouri

20 février 2024

1 Introduction

Ce rapport à pour but de décrire le diagramme de classe et un diagramme de séquence du jeu Tetris, ainsi que de justifier les choix de modélisation.

2 Diagramme de classe

2.1 Description

2.1.1 Classe Brick

Cette classe représente une brique (justification choix modélisation cf 2.2.1). dans un jeu avec les attributs suivants :

1. shape : représente la forme de la brique, de type Shape (cf 2.1.7).
2. brickCoordinates : un vecteur de positions (de la structure Position cf 2.1.6) contenant les coordonnées relatives à un bloc représentant le centre de la brique.
3. boardCoordinate : la position du bloc central sur le plateau de jeu.

Cette classe possède des méthodes tels que :

rotate(r : Rotation) : permettant à la brique d'effectuer une rotation dans le sens horlogique ou anti horlogique (cf 2.1.9).
updateBoardCoordinate(d :Direction) : déplace la brique dans le plateau suivant une certaine direction (cf 2.1.8).

2.1.2 Classe Board

Cette classe est conçue pour représenter et gérer le plateau de jeu, en permettant des actions telles que l'insertion de briques, la suppression de lignes complétées et la vérification des mouvements possibles pour les briques.

En particulier, la méthode *brickBoardInitiator()* permet de commencer le jeu avec un plateau aléatoirement rempli, initialement.
Le plateau de jeu est représenté par un tableau 2D (justification cf 2.2.2).

2.1.3 Classe Bag

La classe "singleton" Bag représente un sac contenant les différentes formes possibles pour une brique (justification cf 2.2.3).
L'attribut index indique quelle sera la prochaine forme à piocher du tableau de formes bricks.
La méthode *shuffle()* permet de mélanger la sac dès qu'on pioche la dernière forme depuis le tableau bricks.

2.1.4 Classe Game

Cette classe est conçue pour être le moteur du jeu, permettant de manipuler les briques, de suivre l'évolution du score et de gérer la progression des niveaux, tout en contrôlant le déroulement de la partie.

Certains attributs et méthodes peuvent sembler moins évidents :

- L'attribut *linesCompleted* permet de comptabilisé le nombre de lignes supprimées afin de mettre à jour le score.

- La méthode *checkEndCollision()* vérifie si la pièce en déplacement vers le bas doit être immobilisée (dans le cas où elle touche une autre brique ou la fin du plateau). Si c'est le cas, elle génère une nouvelle brique et met à jour le score et le niveau.
- *addNewBrick()* génère une nouvelle pièce lorsque la pièce actuelle est immobilisée.
- *endGame()* détermine si le jeu est terminé.

2.1.5 Classe DataBrick

La classe "singleton" DataBrick permet de stocker les positions initiales pour chaque forme de brique différente. Étant donné que toutes les briques ont les même positions initialement (positions relatives par rapport au centre de la brique). Cela permet d'augmenter le niveau de performances.

Concernant les différentes positions possibles de rotation pour chaque brique, nous avons choisis de ne pas les "hardCoder" afin que la Classe DataBrick soit plus lisible et plus élégante. Et étant donné que la méthode de rotation consiste à simplement inverser des signes et des coordonnées.

2.1.6 Structure Position

Répresente les coordonnées d'un certain bloc.

2.1.7 Énumération Shape

Représente les différentes formes possible pour une brique. La valeur EMPTY est nécessaire pour la représentation du plateau de jeu (cf 2.2.2).

2.1.8 Énumération Direction

Représente des différentes directions de déplacement possibles.

2.1.9 Énumération Rotation

Représente des différents sens de rotation possibles.

2.2 Justifications

2.2.1 Représentation d'une brique

Héritage ou une classe générique ?

Les briques ayant toutes une forme différente, il est nécessaire de pouvoir les différencier.

Nous sommes confrontés à deux options : soit utiliser une classe mère pour les briques avec des formes génériques (chaque forme de brique étant un enfant par héritage), soit opter pour une classe plus générique capable de fournir la forme de brique désirée en fonction des paramètres fournis.

La première s'avère beaucoup plus restrictive en termes de maintenance. Si le jeu devait contenir 15 briques de forme différentes, il faudrait implémenter 15 classes différentes. De plus, si l'utilisateur peut créer une nouvelle forme de brique, l'implémentation de cette brique sera compliquée étant donné qu'il faudra créer une nouvelle classe. Par contre, son avantage est la lisibilité car chaque brique de forme différente possède sa propre classe.

L'utilisation d'une classe plus générique s'est avérée être une meilleure solution. En effet, en utilisant une énumération contenant les différentes formes des briques et une classe contenant les coordonnées des différentes briques. La classe générique devient plus facile en termes d'implémentation et de maintenance.

Matrice ou coordonnées relatives ?

Il existe plusieurs moyens de représenter une brique.

Soit utiliser une matrice de booléens où la position des blocs serait marquée par un true, soit représenter la position des blocs au moyen de coordonnées relatives par rapport à un bloc considéré comme le centre.

Le premier choix est la plus intuitif et a pour avantage d'être facile à comprendre par sa représentation visuelle, mais présente comme défaut un gaspillage d'espace : seules les cases où les briques sont positionnées seront utilisées, le reste est perdu. De plus, il est plus difficile de gérer les mouvements de rotation et il est nécessaire de savoir où ces briques sont dans la matrice.

Le second choix est plus logique parce qu'on utilise des coordonnées qui sont toutes relatives à un bloc central (0,0). Cela signifie que les positions des autres blocs sont déterminées en partant de ce bloc central. Cela rend les rotations et les déplacements plus simples à gérer. Étant donné que lorsqu'une brique brique se déplace, c'est uniquement un bloc qui se déplace et donc une coordonnée qui est modifiée.

Représentation d'une coordonnée

Initialement, nous avons choisi une paire d'unsigned char pour représenter la position de la brique sur le plateau. Car les dimensions du plateau de jeu sont raisonnables et qu'un unsigned char est codé sur 1 octet contrairement à l'unsigned int qui est codé sur 4 octets (selon la version). Cela permet ainsi de gagner de l'espace mémoire.

Par la suite, afin d'améliorer la compréhension, nous avons évolué de la paire d'unsigned char vers une structure Position.

2.2.2 Représentation du plateau de jeu

Lors de la conception d'une représentation du plateau de jeu, trois approches principales ont été envisagées :

1. *Tableau 2D (vecteur de vecteur de formes) :* Cette approche consiste à utiliser un tableau bidimensionnel pour représenter chaque case du plateau. Chaque case est associée à une valeur de type Shape, indiquant la forme présente dans cette case ou si elle est vide (EMPTY) . Bien que cette représentation soit intuitive et permette un accès direct aux cases sans nécessiter d'itération, elle présente l'inconvénient de gaspiller de la mémoire lorsque des cases ne sont pas utilisées. Cependant, elle offre des performances optimales en termes d'accès aux cases.

2. *Vecteur de positions :* Cette approche consiste à utiliser un vecteur de positions pour représenter les cases occupées du plateau. Chaque position est enregistrée dans le vecteur, indiquant ainsi les emplacements des briques sur le plateau. Contrairement à l'approche précédente, cette méthode économise de la mémoire en n'enregistrant que les positions occupées. Cependant, elle nécessite une itération à travers le vecteur pour accéder aux cases, ce qui peut entraîner une baisse des performances par rapport à l'approche du tableau 2D. Un autre inconvénient est que pour pouvoir différencier les formes des briques lorsqu'elles sont placées sur le plateau, il faudrait soit créer une nouvelle structure contenant une position et une forme, soit utiliser un vecteur de Brick ce qui n'est pas optimal.

3. *Liste de vecteurs de formes :* Cette approche consiste à utiliser une liste de vecteurs de formes pour représenter le plateau de jeu. Chaque vecteur de formes représente une ligne du plateau, et chaque élément de ce vecteur indique la forme présente dans la case. Cette méthode offre la possibilité de supprimer et de décaler plus facilement les éléments de la liste, ce qui peut être avantageux lorsqu'une ligne du plateau est complétée et qu'elle doit être supprimée. Cependant, elle peut entraîner des performances plus faibles en raison d'une utilisation moins efficace du cache, ce qui peut augmenter les échecs de cache et ralentir les performances.

Finalement, le choix s'est porté sur l'utilisation d'un tableau 2D pour représenter le plateau de jeu, étant donné que dans l'application, un accès fréquent aux cases du plateau est nécessaire et que la taille du plateau reste modérée.

2.2.3 Représentation du sac de briques

En ce qui concerne l'attribut bricks, nous avions initialement envisagé de créer un tableau d'éléments de type Brick. Cependant, avec l'introduction de la classe DataBrick, nous avons conclu qu'il serait meilleur d'opter pour un tableau de formes. Les classes Brick et DataBrick seront alors chargées de créer une nouvelle brique en utilisant les informations fournies par la classe Bag.

3 Diagramme de séquence

3.1 Description

Cette section concerne la description du diagramme de séquence des appels de méthode lorsqu'une brique se déplace vers le bas.

1. La méthode *move()* de la classe Game est appelée avec comme paramètre DOWN de l'énumération Direction.

2. La méthode privée checkEndCollision() est la méthode principale du diagramme de séquence, elle effectue les appels de méthodes suivants :

3. La méthode *canMove()* de la classe Board, qui reçoit en premier paramètre la brique courante afin d'obtenir sa position dans le plateau et en dexième paramètre la direction de déplacement (DOWN dans notre cas). Cette méthode permet de vérifier si la brique courante peut se déplacer vers le bas sans qu'il n'y ai collision ni avec une autre brique ni avec la fin du plateau.

4. Un booléen est retourné par l'appel de méthode *canMove()*.

5. Dans le cas où ce déplacement est possible (moveResult == true), la coordonnée de la brique courante dans le plateau est mise à jour grâce à l'appel de méthode *updateBoardCoordinate()* de la classe Brick, qui reçoit également la direction de déplacement par paramètre.

6. Dans le cas où le déplacement est impossible (il y a eu collision), la brique est insérée dans le plateau par l'appel de méthode *insertBrick()* de la classe Board.

7.8. Après que la brique ai été inséré, *deleteLinesCompleted()* supprime les lignes complétées et renvoie le nombre de lignes complétées (pour le calcul du score et du niveau).

9. Mets à jour le score du joueur uniquement au moyen du nombre de ligne complété. le calcul impliquant le drop est calculé séparement.

10. Augmente le niveau du jeu via *level* si *linesCompleted* atteint un palier de 10 lignes complétées.

11..18. *addNexBrick()* permet la création d'une nouvelle brique. *getBrickShape()* retourne à partir du sac la forme de la prochaine brique à générer. La position centrale du haut du plateau est déterminée au moyen de *getWidth() et getHeight()*.