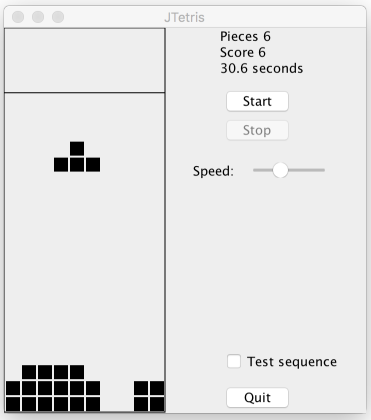
Tetris *

Guillaume Wisniewski
guillaume.wisniewski@limsi.fr

octobre 2017



L'objectif de ce projet est de développer un clone du jeu Tetris. Le programme comportera 7 classes :

1. la classe `TPoints` représentant un point dans un espace 2D ;
2. la classe `Piece` représentant les différentes pièces du jeu ;
3. la classe `Board` représentant la grille sur laquelle les pièces vont évoluer ;
4. la classe `JTetris` gérant toute l'interface graphique (affichage et chute des pièces), interaction avec l'utilisateur (déplacement et rotation des pièces) ;
5. l'interface `Brain` qui définit la manière dont une intelligence artificielle peut interagir avec le programme ;
6. la classe `DefaultBrain` qui implémente une première intelligence artificielle ;
7. la classe `JBrainTetris` qui permettra de faire jouer une intelligence artificielle.

*D'après une idée originale de Osvaldo Jiménez.

Le code d'une partie de ces classes vous est donné.

Vous trouverez également sur le site du cours des *tests unitaires* vous permettant de tester les différentes méthodes que vous aurez à développer. Les tests unitaires permettent de tester de manière répétée et rapide les différentes méthodes à écrire.

Les fichiers contenant les tests unitaires doivent être mis dans le même répertoire que votre code source. Il faut également ajouter la bibliothèque `JUnit` à votre projet (par exemple en utilisant l'option « Fix project setup » du `QuickFix`).

Chaque classe de test comporte plusieurs méthodes définissant chacune un ou plusieurs tests unitaires. Le principe d'un test unitaire est toujours le même :

— Initialisation de différents objets pour reproduire un « contexte d'exécution » (par exemple en plaçant différentes pièces sur une grille) ;

— Appel de la méthode à tester ;

— Comparaison du résultat de cette méthode (soit la valeur renvoyée, soit une modification des attributs) à l'aide de la méthode `assertEquals`.

La Figure 1 donne un exemple de test unitaire.

```java
// l'annotation Test permet d'indiquer que la méthode va réaliser
// un test unitaire
@Test
public void testFallingMaxHeight() {
  // définition du contexte
  Board b = new Board(3, 6);
  Piece p = new Piece(Piece.STICK_STR);
  b.place(p, 0, 1);

  // appel de la méthode à tester et comparaison au résultat
  // attendu
  assertEquals(5, b.getMaxHeight());
}
```

FIGURE 1 – Exemple d'un test unitaire de la méthode `getMaxHeight`.

En exécutant les tests unitaires (de la même manière qu'en exécutant un programme java « normal »), il est possible de savoir de manière automatique (la bibliothèque `JUnit` est capable de déterminer quelles fonctions ne renvoient pas la valeur attendue) et répétée (l'ensemble des tests peut être exécuté à n'importe quel moment) quelles parties du programme fonctionnent et quelles parties ne fonctionnent pas.

1 Les pièces

Il y a sept pièces dans la version standard de Tetris qui sont représentées Figure 2. Chacune des pièces est composée de 4 blocs et peut subir une rotation de 90° dans le

sens anti-horaire pour générer une nouvelle pièce. La figure 3 représente les différentes pièces pouvant être obtenues par rotation des sept pièces de base.

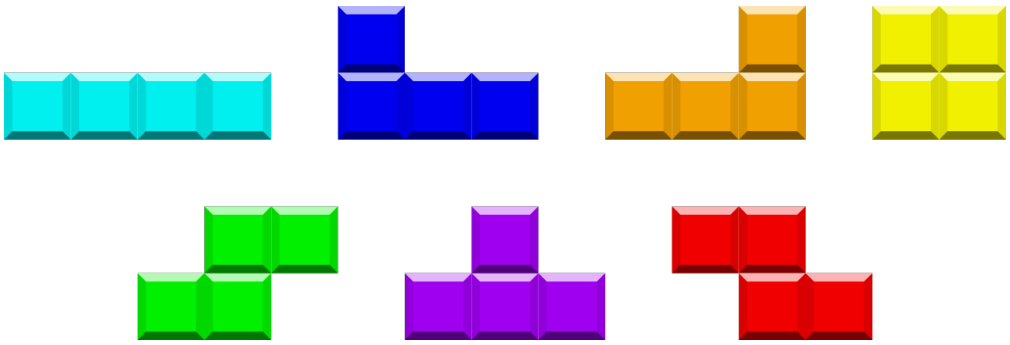


FIGURE 2 – Les différentes pièces de tetris : le bâton *(stick)*, le L et son miroir L2, le carré, le S, la pyramide et le S2.

1.1 Attributs

La classe `Piece` nous permettra de représenter une pièce de Tetris dans une rotation donnée. Une pièce sera définie par les coordonnées des blocs qui la compose, en supposant que chaque pièce dispose de son propre système de coordonnées dont l'origine est dans le coin gauche bas et que la pièce est positionnée aussi bas et à gauche que possible. Par exemple, les coordonnées des blocs composant un carré sont $\{(0,0),(0,1),(1,0),(1,1)\}$ et celles de la première rotation du S (pièce verte) : $\{(0,1),(0,2),(1,0),(1,1)\}$

Il est possible, à partir des coordonnées des blocs composant une pièce, de définir plusieurs propriétés intéressantes d'une pièce :

- sa largeur *(width)* : l'abscisse du bloc le plus à gauche de la pièce ;
- sa hauteur *(height)* : l'ordonnée du bloc le plus élevé de la pièce ;
- son contour *(skirt)* : un tableau d'entiers dont la taille est égale à la largeur de la pièce et qui stocke, pour chaque abscisse possible l'ordonnée du bloc le plus bas de cette colonne. Par exemple, dans le cas de la pièce S, le contour aura pour valeur $\{0,0,1\}$.

① Complétez le constructeur de la classe `Piece` se trouvant sur le site du cours permettant d'initialiser une pièce à partir d'un tableau de coordonnées. Il faut pour cela donner le code toutes les méthodes appelées dans le constructeur.

② Complétez le constructeur de la classe `Piece` permettant d'initialiser une pièce à partir d'une chaîne de caractères contenant les différentes coordonnées séparées par des espaces. Par exemple la chaîne `"0 0 0 1 0 2 1 0"` représentera la pièce de coordonnées $\{(0,0),(0,1),(0,2),(1,0)\}$

La classe `Piece` définit également 7 constantes, permettant de créer facilement une des pièces de base de Tetris :

```
Piece pyr1 = new Piece(Piece.PYRAMID_STR);
```

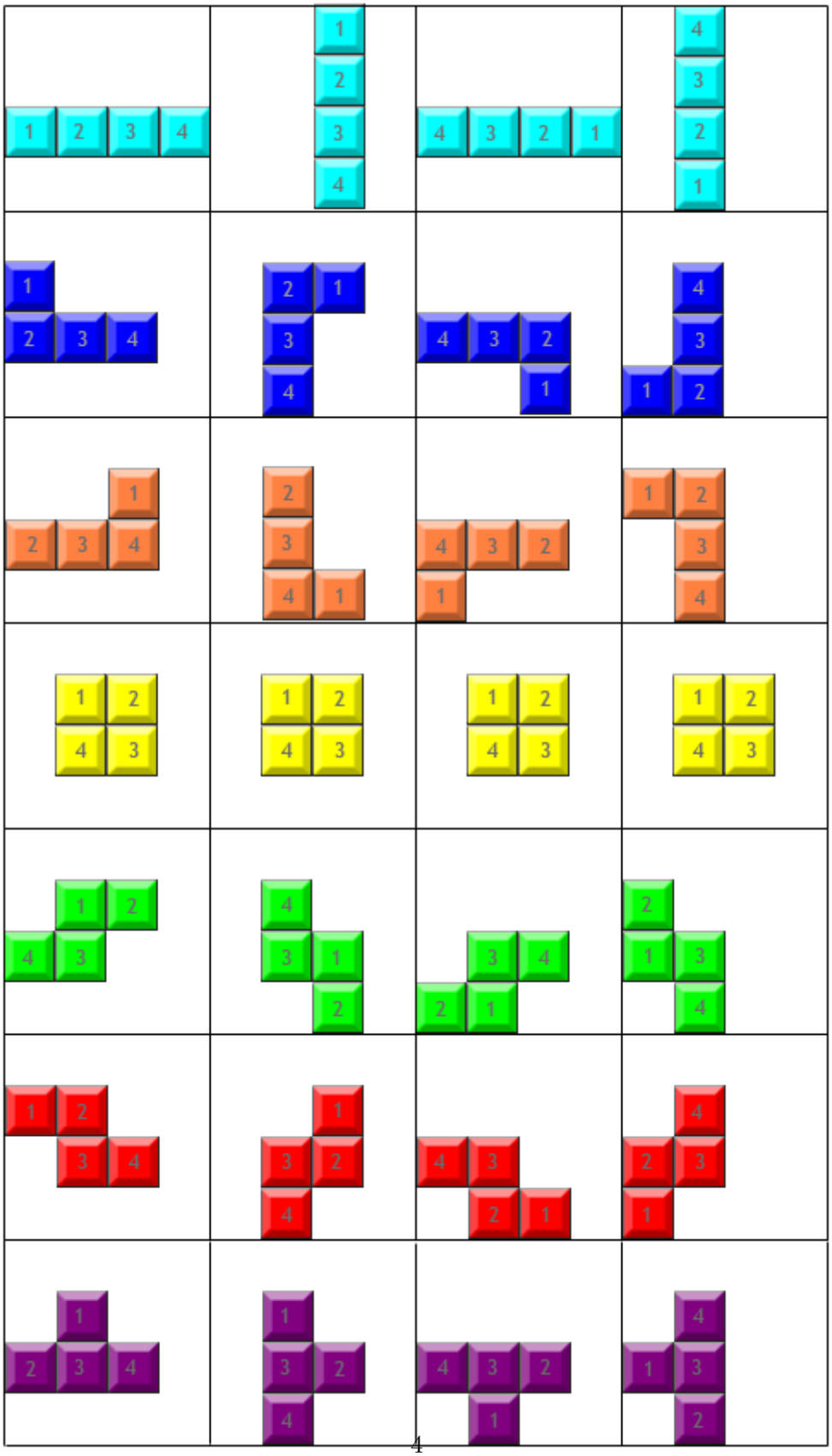


FIGURE 3 – Les différentes pièces obtenues par rotation des pièces de base.

1.2 Méthodes utilitaires

③ Donnez le code des méthodes `equals` et `toString` de la classe `Piece`. Ces méthodes seront utilisées dans les autres classes du projet et, notamment, dans les tests unitaires.

1.3 Rotation d'une pièce

La méthode `computeNextRotation()` de la classe `Piece` permet de déterminer la prochaine rotation d'une pièce. Il est important de noter que dans Tetris la rotation des pièces se fait en sens anti-horaire. La classe `Piece` étant *immutable*, cette méthode renvoie une nouvelle instance de `Piece`, l'instance courante n'étant pas modifiée.

④ Écrivez le code de la méthode `computeNextRotation()`.

Indication : on pourra considérer trois opérations : i) l'inversion des abscisses et des ordonnées de chaque point, ii) la symétrie des coordonnées selon un axe horizontal et iii) la symétrie des coordonnées selon un axe vertical.

2 Le Board

La classe `Board` permet de représenter la grille sur laquelle évolueront les différentes pièces. Elle maintient également différentes structures de données facilitant le placement des pièces et la détection des lignes qui sont pleines.

Son principal attribut, `grid` représente une grille (un tableau de booléens de deux dimensions) permettant d'indiquer quelles sont les positions occupées par des pièces (ce sont, par convention, les positions dont la valeur dans le tableau est `true`). L'origine de la grille (le point de coordonnées $(0, 0)$) est choisi comme le coin inférieur gauche. Les deux principales méthodes permettant de manipuler la grille sont `clearRows` qui permet de supprimer une ligne pleine (et de faire « tomber » les lignes se situant au-dessus de celle-ci) et `place` qui permet d'ajouter une pièce à la grille.

La classe `Piece` définit également deux attributs, `heights` et `widths` facilitant le développement des autres méthodes de la classe. Ces deux attributs permettent de représenter, respectivement, le nombre de cases remplies sur chaque colonne et sur chaque ligne. La Figure 4 donnent un exemple du calcul de ces valeurs.

⑤ Complétez le constructeur de la classe `Board`.

⑥ Définissez un constructeur de copie qui permet d'initialiser une instance de `Board` en *copiant* les attributs d'une instance passée en paramètre.

2.1 Ajout d'une pièce

La méthode `place` permet d'ajouter une pièce à la grille, c'est-à-dire de faire passer à `true` toutes les cases occupées par la pièce. Elle prend en paramèter, une `Piece` ainsi que les coordonnées (x, y) auxquelles la pièce doit être placée.

La méthode renvoie :

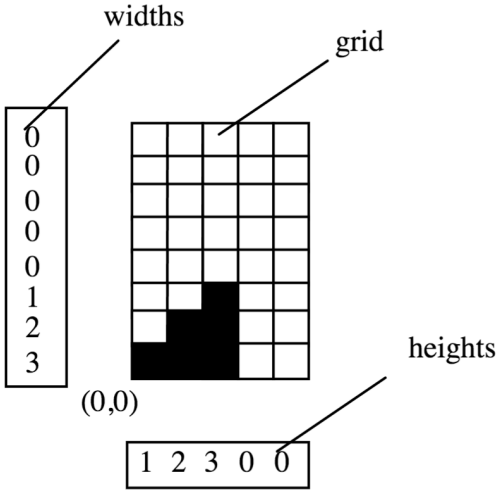


FIGURE 4 – Exemple des valeurs des attributs `heights` et `widths`.

— `PLACE_BAD` si la pièce entre en collision avec une pièce déjà placée sur la grille ;

— `PLACE_OUT_BOUNDS` si la pièce sort de la grille ;

— `PLACE_ROW_FILLED` si la pièce peut être placée et qu'elle complète la ligne ;

— `PLACE_OK` dans tous les autres cas.

Cette méthode doit également mettre à jour les attributs `heights` et `widths`.

⑦ Implémentez la méthode `place`.

⑧ Implémentez lat méthode `getMaxHeight` qui retourne la taille de la colonne la plus haute.

2.2 Suppression des lignes pleines

La méthode `clearRows` supprime toutes les lignes de la grille qui sont pleines, entraînant la chute des éléments situées au-dessus de celle-ci. Elle renvoie le nombre de lignes supprimées. Des lignes vides sont ajoutées en haut de la grille. Il peut y avoir plusieurs lignes à supprimer et les lignes à supprimer ne sont pas nécessairement adjacentes. Note : dans la version originale de Tetris, il n'y a pas de « gravité » et un bloc ne tombe que d'une case même s'il « atterrit » sur une case vide.

⑨ Implémentez la méthode `clearRows`.

2.3 La méthode `dropHeight`

La méthode `dropHeight` calcule l'ordonnée y à laquelle l'origine d'une pièce se trouvera si celle-ci « tombait » directement de sa position actuelle. Cette valeur peut être

déterminée efficacement (c.-à-d. en une seule itération sur la largeur de la pièce) en utilisant l'attribut `heights` et la *skirt* de la pièce.

On supposera qu'il n'est pas possible de déplacer la pièce au cours de sa chute.

⑩ Implémentez la méthode `dropHeight`.

2.4 Les méthodes `undo`/`commit`

La classe `Board` implémente également un mécanisme permettant d'annuler le placement d'une pièce. Ce mécanisme permet de gérer facilement la chute et le déplacement d'une pièce en « effaçant » l'objet qui vient d'être affiché et en le re-dessinant à une autre position. Son implémentation nécessite la définition de deux méthodes : la méthode `undo` qui permet d'annuler le dernier placement et la méthode `commit` qui permet de « valider » le dernier placement et de « sauvegarder » l'état de la grille. Par soucis de simplicité, il ne sera possible d'annuler qu'une seule opération `place`. Pour garantir cela, un attribut supplémentaire, `commited` sera défini : il permettra d'assurer qu'un appel à `place` est systématiquement suivi soit d'un appel à `commit`, soit d'un appel à `undo`.

De manière plus formelle :

- la grille est créé avec `commited` mis à `true` ;
- il est possible de faire un unique appel à `place` qui fait passer l'attribut `commited` à `false` ; l'opération `place` n'est pas possible dans ce cas : si deux appels successifs à `place` ont lieux, le second lévera une exception `RuntimeException`.
- il est alors possible de faire une opération `undo` qui restaure la grille dans son état orignal (avant l'appel à `place`) et fait passer l'attribut `commited` à `true`.
- ou alors, la méthode `commit` est appelée qui fait une copie de l'état courant de la grille (et des autres attributs qui lui sont liées) et fait passer l'attribut `commited` à `true`.

Cette stratégie assure que soit la méthode `undo` soit la méthode `commit` ait été appelé entre deux ajouts de pièces.

Les méthodes `undo` et `commit` ne font rien lorsque `commit` est `true`.

L'implémentation de ces deux méthodes nécessitent de définir trois nouveau attributs `backupGrid`, `backupWidths` et `backupHeights` pour stocker les sauvegardes des trois attributs « principaux ».

⑪ Modifiez la méthode `place` pour qu'une `RuntimeException` soit levée en cas de deux appels successifs ;

⑫ Implémentez les méthodes `undo` et `commit`.

3 Intelligence artificielle

Nous allons maintenant ajouter la possibilité de faire jouer une intelligence artificielle à notre version de Tetris. Nous définissions pour cela l'interface `Brain` qui définit la méthode `bestMove` capable de déterminer la position optimale d'une pièce. Une première implémentation de cette interface vous ait donné (classe `DefaultBrain`).

L'objectif de cette partie du projet est développer une classe `JBrainTetris` est une sous-classe de `JTetris` qui utilise une intelligence artificielle pour jouer « automatiquement » à Tetris. Cette classe devra :

⑬ Définir une fonction `main` responsable de la création d'une *frame* contenant une instance de `JBrainTetris`.

⑭ Surcharger la méthode `createControlPanel()` pour ajouter une étiquette `"Brain"` à côté d'une case à cocher qui contrôle si le cerveau est « actif » ou non. Par défaut celui-ci n'est pas actif. Il suffit pour cela d'utiliser le code suivant :

```
panel.add(new JLabel("Brain:"));
brainMode = new JCheckBox("Brain active");
panel.add(brainMode);
```

⑮ Définir une instance de `DefaultBrain`

⑯ Surcharger la méthode `tick()` de manière à ce qu'à chaque appelle celle-ci avec le verbe `DOWN`, la pièce soit déplacée vers la place prédite par le cerveau. On autorisera un déplacement vers la gauche ou la droite et une rotation par appel à `thick`.

⑰ `JBrainTetris` doit détecter la mise en place d'une nouvelle pièce (par exemple grâce à l'attribut `count` de la classe `JTetris`) pour déterminer (une seule fois !) l'endroit où la pièce devra être placée.

4 Jeu avec un adversaire

⑱ Modifier la fonction `createControlPanel()` de la classe `JBrainTetris` pour ajouter une étiquette `"Adversaire"` et un curseur *(slider)* allant de 0 à 100.

⑲ Surcharger la méthode `pickNextPiece` pour que l'adversaire puisse choisir la pièce avec une probabilité correspondant à la valeur du curseur. Il suffit pour cela de tirer un nombre aléatoire entre 0 et 100. Si celui-ci est plus petit que la valeur du curseur, la pièce est choisi aléatoirement ; sinon, l'adversaire va choisir la pièce la plus désavantageuse pour le joueur.

On pourra (ré)utiliser la classe `DefaultBrain`.

5 Travail à effectuer

Le projet s'effectuera en binome. Vous devrez rendre à votre chargé de TP :

— le code de toutes les classes demandées ;

— un rapport de deux pages décrivant le travail effectué, les difficultés rencontrées et les solutions apportées ainsi que la contribution de chacun des membre du binôme. Ce rapport ne devra pas comporter de code.

Le projet devra être rendu avant le 22 décembre minuit.