Mini-Projet : jeu 2048 (à faire en binôme)

Le jeu 2048 est une variante du *jeu de taquin*, inventée par le web designer Gabriele Cirulli en 2014. Le jeu original est une grille 4×4 où chaque case est soit vide, soit contient une tuile étiquetée par une puissance de 2. Le but du jeu est de faire glisser (verticalement ou horizontalement) toutes les tuiles (en même temps) sur la grille afin de combiner les tuiles de mêmes valeurs. Si deux tuiles de mêmes valeurs (2^k) sont adjacentes pendant le glissement, elles se combinent en une unique tuile étiquetée par la somme des valeurs (2^{k+1}), somme qu'on ajoute au "score" de la grille. Après chaque déplacement, une nouvelle tuile apparaît aléatoirement sur un des emplacements vides. Cette nouvelle tuile a pour valeur soit 2, soit 4, avec probabilité respective $\frac{9}{10}$ et $\frac{1}{10}$ (dans la version initiale). Le jeu commence avec deux tuiles placées aléatoirement sur la grille ; il se termine soit lorsque le total est atteint (2048 dans la version initiale), soit lorsque toutes les cases sont occupées et qu'aucun mouvement ne permet de combiner de tuiles.

Objectif : Rendre un programme mettant en oeuvre les bases du jeu 2048, en introduisant quelques variantes par rapport à la version originale [1]

Bases du jeu

L'action de base du jeu consiste donc à provoquer des glissements des tuiles actuelles de façon à fusionner deux tuiles "adjacentes" (elles peuvent être séparées par des cases vides) de mêmes valeurs. A chaque action on provoque le glissement en indiquant par une touche la direction de ce glissement. Qu'on ait pu procéder ou non à des fusions, l'étape se termine par un compactage de la grille dans la direction indiquée : les tuiles sont déplacés de façon à ne pas laisser de case vide entre deux tuiles dans cette direction.

Le principe du jeu est clair et le jeu est accessible sur Internet. Nous apportons quelques précisions pour lever certaines ambiguïtés. Nous illustrons les situations avec un déplacement vers la droite sur une seule ligne. Les autres cas sont similaires :

— A chaque étape, une tuile ne peut faire l'objet que d'une fusion : une tuile qui résulte de la fusion de deux tuiles ne peut pas, dans la même étape, fusionner avec une autre tuile ;

— A chaque étape il est possible de faire plusieurs fusions sur la même ligne ou colonne tant que ces fusions s'appliquent à des tuiles différentes
L'exemple ci-dessous illustre les deux règle précédentes, avec deux fusions indépendantes :

```
*************************                    *************************
*    4 *   4 *   4 *   4 *     donne         *     *     *   8 *   8 *
*************************                    *************************
```

— Quand on a le choix de fusionner une tuile avec l'une ou l'autre de ses deux tuiles adjacentes, on commence les fusions par les tuiles à l'extrémité de la ligne/colonne dans la direction du mouvement. Dans l'exemple ci-après, puisqu'il s'agit d'un déplacement vers la droite, la tuile du milieu fusionne avec celle à sa droite :

```
************************                     ************************
*   4 *   4 *   4 *     *      donne         *     *     *   4 *   8 *
************************                     ************************
```

De même dans le premier exemple, ce ne sont pas les deux tuiles du milieu qui ont fusionné, sinon la ligne contiendrait deux tuiles 4 qui encadreraient une tuile 8.

1. L'aspect plaisant du jeu est lié à la taille de la grille, la valeur à atteindre et les proportions respectives des tuiles 2 et 4, de façon à rendre le jeu ni trop simple, ni trop long.

Réalisation

Le découpage du programme en fonctions est imposé mais vous pouvez ajouter des fonctions auxiliaires. Votre réalisation doit respecter les en-têtes des fonctions et les comportements indiqués. Votre réalisation sera soumise à un programme de test basé sur ces fonctions. Si vous en changez le comportement, votre programme échouera aux tests.

Une instance du jeu est décrite par le type `Grille`. Les fonctions demandées prennent en paramètre une référence sur une grille et permettent d'accéder à sa configuration courante ou de la manipuler. On peut tester l'application en appliquant les fonctions à des configurations prédéfinies qu'il serait problématique d'obtenir en se basant uniquement sur les actions de jeu. Les interactions avec utilisateur et l'aspect aléatoire permettront dans un second temps d'enrichir votre application avec des situations non anticipées et de réaliser un vrai jeu.

Fonctions à implémenter

- `bool init(Grille &g, int dimension, int cible, int proportion)`
  initialise `g` avec les paramètres indiqués ; En sortie, `g` doit comporter deux tuiles dont les valeurs et positions sont obtenues par des appels aux fonctions `nouvelle` et `place` décrites ci-après ; la fonction renvoie `true` en cas de succès et `false` sinon.
- `bool charge(Grille &g, const vector<vector<int>> &v, int cible, int proportion)`
  même principe que `init` en initialisant la grille à partir du vecteur `v` : chacune de ses lignes donne les valeurs des tuiles, de gauche à droite (la valeur 0 désigne une case vide). Le score de la grille est initialisé à 0 quel que soit le contenu de `v`.
  Cette fonction permet de placer la grille dans une configuration définie à l'avance.

- Les fonctions `int gauche(Grille &g)`, `int droite(Grille &g)`, `int haut(Grille &g)`, `int bas(Grille &g)` correspondent aux actions de glissement. Chaque fonction procède aux fusions de tuiles dans sa direction, puis au compactage, et renvoie le nombre de cases libres de la grille. Si l'action demandée est impossible, la fonction renvoie -1 et laisse la grille inchangée. Dans ce cas, au retour de la fonction, on n'ajoutera pas de nouvelle tuile.

Les fonctions ci-dessous permettent de connaître la configuration courante du jeu :

- `bool succes(const Grille &g)` : renvoie *true* ssi la valeur cible a été atteinte ;
- `int vides(const Grille &g)` : renvoie le nombre de case vides ;
- `int cible(const Grille &g)` : renvoie la valeur à atteindre ;
- `int score(const Grille &g)` : renvoie le score courant ;
- `int proportion(const Grille &g)` : renvoie la proportion de tuiles 2 au tirage, exprimée par un entier entre 0 et 10 (/ 10) ;
- `int dimension(const Grille &g)` : renvoie la dimension de la grille.

Fonctions mises à disposition

- `void affiche(const Grille &g)` : affiche la grille ainsi que le score courant et le nombre de cases libres. Pour obtenir un affichage propre, on suppose que la valeur d'une tuile occupe au plus 4 positions.
- `int nouvelle(const Grille &g)` : renvoie la valeur de la prochaine tuile en respectant la proportion demandée de tuiles de valeur 2 ;
- `int place(const Grille &g)` : décrit implicitement la case qui recevra la prochaine tuile (`g` doit avoir au moins une case libre). La fonction renvoie un entier [2] compris entre 1 et le nombre de cases libres de `g`, qu'on interprète comme le rang de la case libre lors d'un parcours de la grille de haut en bas et de gauche à droite. Par exemple, les coordonnées des cases libres dans la grille ci-après sont `(1, 3)`, `(1, 4)`, `(3, 1)`, `(4, 2)`. L'appel `place(g)` renverra un entier entre 1 et 4 : si c'est 1, la prochaine tuile sera à placer sur la case de coordonnée `(1, 3)` ; si c'est 2, elle sera à placer sur la case `(1, 4)`, etc.

2. Cet entier est obtenu via un "générateur de nombres pseudo-aléatoire" (voir une explication sur Wikipédia) en respectant une distribution uniforme sur l'ensemble des cases vides.

```
************************
* 128 *  4  *     *     *
************************
*  2  * 16  *  4  *  2  *
************************
*     *  8  *  8  *  4  *
************************
*  2  *     *  2  *  4  *
************************
```

— `void resetRand(bool mode)` : Si `mode` vaut *true* le générateur de nombres aléatoires est initialisé avec une valeur aléatoire et fournira à chaque exécution une séquence différente de valeurs. Sinon, il fournira toujours la même séquence de valeurs si on procède aux mêmes appels à `nouvelle` et `place`. Utiliser la même séquence de nombres aléatoires facilite la mise au point de votre programme en assurant la reproductibilité du comportement d'une exécution à l'autre. Pour ne pas répéter la même partie à chaque fois, on appelle `resetRand` avec `true` au lancement du programme.

Interaction avec l'utilisateur

On indique chaque déplacement (*haut, bas, gauche* et *droite*) par l'intermédiaire de caractères (par exemple `h`, `b`, `g` et `d` ou encore `i`, `k`, `j` et `l`) suivi de `Entrée`[3]. On affiche la grille à chaque étape, une fois la nouvelle tuile insérée.

Voici un exemple possible d'exécution :

```
************************
* 128 *  4  *     *     *
************************
*  2  * 16  *  4  *  2  *
************************
*     *     *     *     *
************************
*  2  *     *     *     *
************************

Entrer commande : h
************************
* 128 *  4  *  4  *  2  *
************************
*  4  * 16  *     *     *     fusion des cases (2, 1) et (4, 1)
************************
*     *     *     *     *
************************
*     *  2  *     *     *     apparition d'une tuile 2 en position (4, 2)
************************

Entrer commande : g
************************
* 128 *  8  *  2  *     *     fusion des cases (1, 2) et (1, 3) et
************************
*  4  * 16  *     *     *     glissement de la tuile (1, 4)
************************
*     *     *  4  *     *     apparition d'une tuile 4 en position (3, 3)
************************
*  2  *     *     *     *     glissement vers la gauche de la tuile (4, 2)
************************
```

3. Ne pas se servir des touches de type “flèche” du clavier qui sont interprétées de façon spéciale.

Extensions possibles

Pour les groupes qui le souhaiteraient, une fois le coeur de l'application réalisé, il est possible d'ajouter quelques fonctionnalités :

- réaliser un menu interactif permettant de changer interactivement les valeurs des paramètres de départ, de lancer un test spécifique, de redémarrer une partie, etc. Un exemple de menu interactif est présent dans le premier polycopié comme illustration de l'instruction `switch` ;
- Sauvegarder la configuration courante dans un fichier textuel dont on passe le nom en paramètre et restaurer une configuration à partir d'un fichier : le jeu reprend dans l'état dans lequel il était au moment de la sauvegarde.
- Conserver l'historique des commandes de l'utilisateur pour pouvoir rejouer (partiellement) une partie.

Les squelettes des fonctions pour lire et écrire dans un fichier seront fournis ultérieurement.

Organisation des fichiers

Pour faciliter les tests, votre programme sera constitué des fichiers suivants :

- `jeu.h` contient votre définition du type `Grille` et les en-têtes des principales fonctions ;
- `jeu.cpp` contient les fonctions demandées ainsi que vos éventuelles fonctions auxiliaires ;
- `dispo.cpp` contient des fonctions fournies par l'équipe enseignante (vous ne devez pas modifier ce fichier car il pourra évoluer si nous mettons de nouvelles fonctions à disposition) ;
- `test.cpp` contient vos fonctions de test ainsi que la fonction `main` pour lancer l'exécution.

Une archive sera fournie avec les squelettes des fichiers ainsi qu'un moyen de lancer la compilation et l'exécution, comme en TP.

Modalités d'évaluation

La notation tiendra compte de la qualité de votre code et de la pertinence des tests que vous aurez pu ajouter à ceux fournis. La réalisation des extensions proposées ne comptera que marginalement. Vous devrez aussi fournir un document décrivant brièvement l'état d'achèvement du projet et la participation de chaque membre du groupe.

Le projet est à faire en binôme mais la note pourra être individualisée au vu de la participation de chacun à la réalisation (notamment lors des séances encadrées).

L'équipe pédagogique pourra utiliser des logiciels de calcul de similarités entre programmes pour mettre en évidence d'éventuels plagiats.