SameGame

DUT INFORMATIQUE 2021 IUT FONTAINEBLEAU (77)

**Arthur DECORBEZ & Shana LEFEVRE**

**Groupe 4 et 3**

**Table des matières**

[​ I] Introduction 3](#__RefHeading___Toc210_2238476167)

[​ II] Description et fonctionnalités 3](#__RefHeading___Toc212_2238476167)

[​ III] Présentation de la structure du programme 5](#__RefHeading___Toc214_2238476167)

[​ IV] Les algorithmes 6](#__RefHeading___Toc634_2238476167)

[​ 1er algorithme : Détection des groupes au survole de la souris : 6](#__RefHeading___Toc88_503968669)

[​ 2e algorithme : Détection des lignes vides 7](#__RefHeading___Toc100_503968669)

[​ 3e algorithme : Détection des colonnes vides 7](#__RefHeading___Toc153_587020016)

[​ 4e algorithme : Détection de fin de jeu 8](#__RefHeading___Toc155_587020016)

[​ V] Conclusions personnelles 9](#__RefHeading___Toc636_2238476167)

# I] Introduction

Pour ce deuxième semestre, nous avons comme projet tutoré le «Samegame» à coder en java. Le but de ce jeu est de vider la grilles des ses blocs. Il y a trois blocs possibles: bleu, rouge et vert. Les blocs s’éliminent par groupe. Un groupe est composé d’au minimum de deux blocs identiques adjacents. Le système de points dépend du nombre de blocs par groupe. On utilisera la formule suivante:

**( n - 2 ) ²**

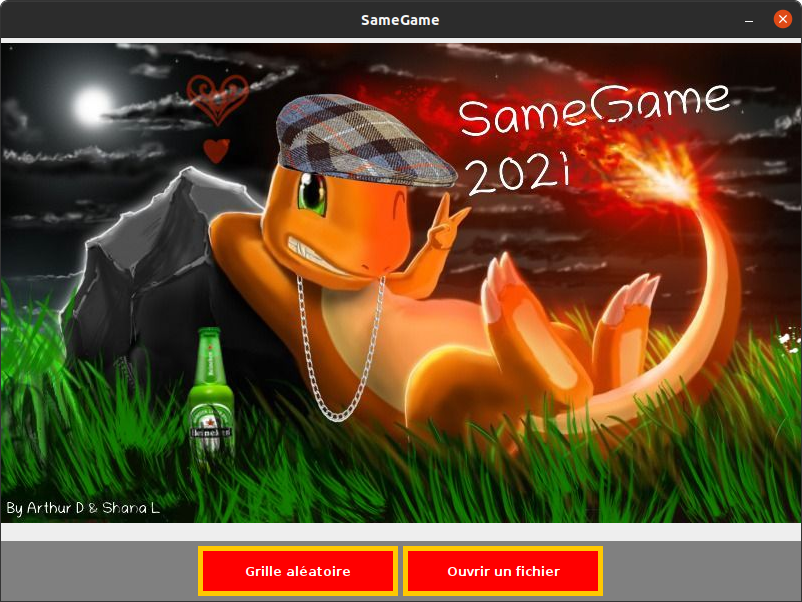
*Exemple: Groupe de 5 → ( 5 – 2 ) ² = 9 points*

Dès qu’un groupe est éliminé, les autres blocs se déplacent vers la gauche ou bien vers la droite afin de supprimer les «trous».

# II] Description et fonctionnalités

Durant toute l’exécution de notre programme, nous y avons ajouté des mentions dans le terminal qui nous servirons comme points de repères et qui nous assurerons le bon fonctionnement de notre jeu.

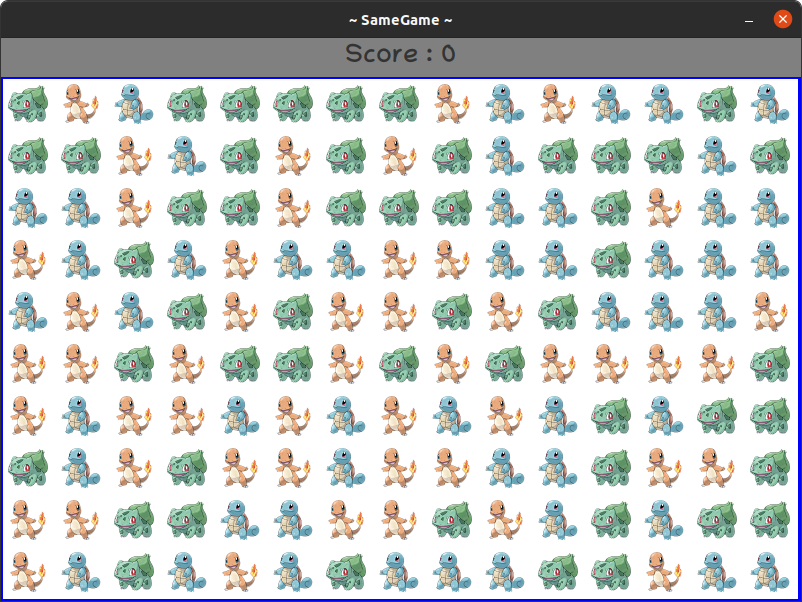
Notre jeu accueille le joueur sur notre page menu. Delà, l’utilisateur peut choisir comment générer sa grille : à partir d’un fichier ou bien aléatoirement. Les deux modes ont la même finalité sauf que c’est la génération de la grille qui changera.



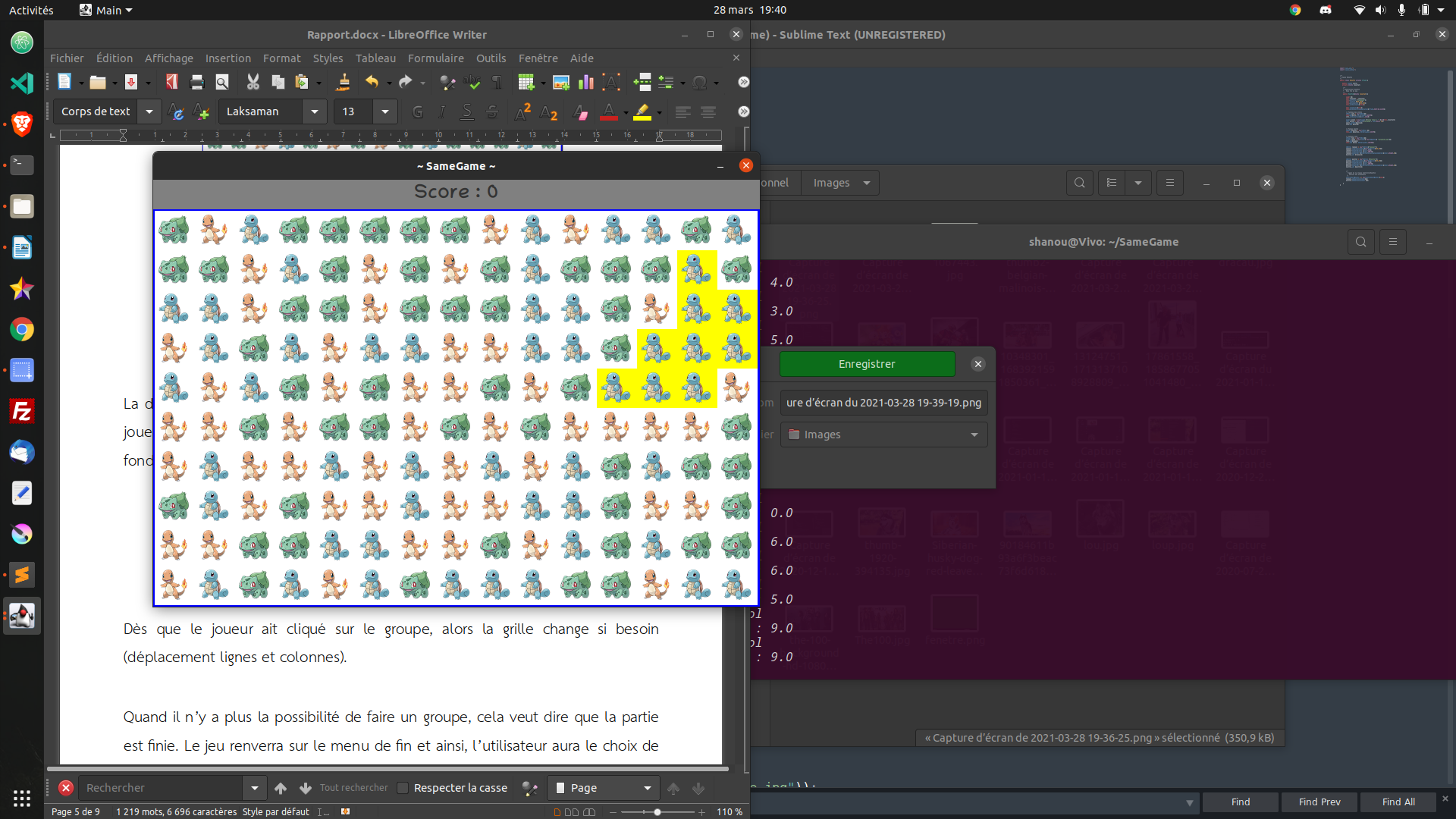
Si l’utilisateur choisit d’ouvrir un fichier, cette fenêtre ci-dessous s’ouvrira. Le joueur ne pourra ouvrir qu’un fichier ayant comme extension «**.gri**».



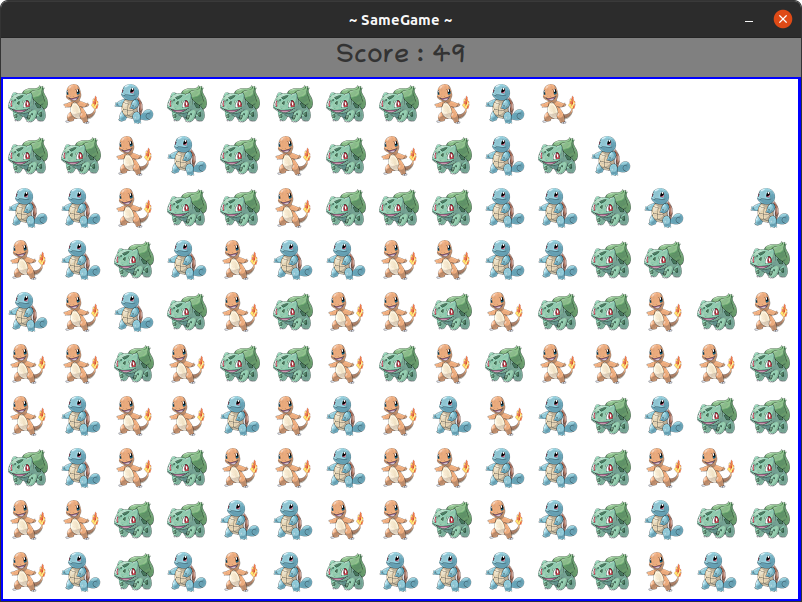
Après que le joueur ait généré soit sa grille aléatoirement ou soit via l’ouverture d’un fichier, la grille apparaît sur la fenêtre de jeu.



La détection d’un groupe de même bloc se fait automatiquement. Dès que le joueur mettra sa souris sur un des blocs de la grille, son groupe apparaîtra en fond jaune.



Dès que le joueur ait cliqué sur le groupe, alors la grille change si besoin (déplacement lignes et colonnes). Le score se met ensuite à jour,



Quand il n’y a plus la possibilité de faire un groupe, cela veut dire que la partie est finie. Le jeu renverra sur le menu de fin et ainsi, l’utilisateur aura le choix de rejouer ou bien de quitter le programme.

# III] Présentation de la structure du programme

Notre programme est composé de plusieurs fichiers répartis en quatre grandes parties.

- La première partie : l’accueil du joueur avec le fichier Menu.java ;

- La seconde partie : la création de la grille (aléatoire ou fichier) ;

- La troisième partie : le jeu ;

- La quatrième partie : la fin du jeu avec le fichier MenuFin.java et la possibilité de relancer le jeu.

Voici ci-dessous un diagramme de classes de notre projet :

# IV] Les algorithmes

Pour nos algorithmes, nous utilisons deux tableaux de type *char* pour référencer les coordonnées d’un bloc :

- **tabTerm [ ][ ]**: correspond à la valeur d’un bloc. Il peut prendre comme valeur soit : ‘**R’** (rouge), ‘**V’** (vert), ‘**B’** (bleu) et ‘**X’** (case n’ayant pas de bloc)

- **bool [ ][ ]** : correspond au statut d’un bloc. Il peut prendre comme valeur soit : ‘**T’** (true), ‘**F’** (false) et ‘**.’** (pas de bloc donc pas de statut particulier).

## 1er algorithme : Détection des groupes au survole de la souris :

Nous utilisons la méthode **Radar** qui prend en paramètre un x et un y de type int qui correspondent respectivement à une ligne et à une colonne que le joueur aura survolés avec sa souris.

La méthode va commencer par vérifier si tabTerm [x][y] a la même valeur que les tabTerm [ ][ ] du haut, du bas, de la droite et de la gauche soit R, V ou B.

Si le tabTerm [ ][ ] découvert est de la même valeur, alors à ses coordonnées , on y place un ‘T’ au tableau bool [ ][ ]. Le tableau bool [ ][ ] permet de dire que tel bloc appartient ou non au groupe donc, si bool [ ][ ] == ’F’ alors le tabTerm [ ][ ] correspondant n’appartient pas au groupe contrairement à bool [ ][ ] = ’T’. Donc, pour lancer une vérification dans les 4 sens, il faut que bool [ ][ ] == ’F’.

Après chaque appel de la méthode Radar, on réinitialise le tableau bool [ ][ ] = ’F’. Cependant si bool [ ][ ] = ’.’, alors cela signifie que tabTerm [ ][ ] n’a plus de bloc correspondant (R, V, B), il prend donc la valeur ‘X’.

## 2e algorithme : Détection des lignes vides

Afin de faire déplacer le ou les bloc(s) dès qu’une « case » est libre, il faut détecter une case libre. On commence d’abord par parcourir le tableau ligne par colonne avec deux boucles imbriquées. Si on trouve tabTerm [ ][ ] == ’X’, alors on appelle la méthode **OrganiserLignes** qui récupère en argument la ou les colonnes où il y a une case libre.

Dans notre méthode **OrganiserLignes**, on regarde pour chaque ligne que la ou les case(s) soient libres. On « copie » en mettant toutes les valeurs qui nous intéresse dans la variable *color* de type char*.* Enfin, nous modifions la valeur de tabTerm [ ][ ] en case libre en ligne - 1 pour après rajouter notre variable *color* en argument en appelant notre méthode **ChangerBloc**.

## 3e algorithme : Détection des colonnes vides

Puis, après d’avoir détecté des lignes vides, il faut détecter s’il y a des colonnes vides. Pour cela, on commence par parcourir toutes les colonnes une par une. Pour chaque colonne, on vérifie si notre méthode **VerificationCol** est vraie. Si cette méthode retourne *true*, alors on fait appelle à la méthode **CopyColonne**.

La méthode **VerificationCol** prend en paramètre la n colonne et elle permet de voir si toute la colonne est vide (tabTerm [ ][ ] == ’X’). Si oui, alors elle retourne *true*, sinon *false*.

La méthode **CopyColonne** permet de copier les colonnes concernées et de les décaler vers la gauche. Pour se faire, nous créons trois boucles imbriquées. Les deux premières concernent le nombre total de colonnes que l’on peut décaler. Dans ce cas, 14/15 colonnes peuvent être décalées. Puis on vérifie que la colonne est vide avec la méthode **VerificationCol**. Enfin, pour chaque ligne, on décale de colonne +1.

## 4e algorithme : Détection de fin de jeu

A chaque clique, on vérifie s’il y a encore la possibilité de faire des groupes avec les blocs restants.

On appelle la méthode **End**. Si la méthode **End** est *false*, alors le jeu est finit.

La méthode **End** parcours la totalité du tableau. Si les cases ne sont pas vides (tabTerm [ ][ ] != ‘X’) et si elles n’appartiennent pas à un groupe (bool [ ][ ] = ’F’), alors, on appelle la méthode **Radar** (*décrite précédemment)*. Si le groupe est supérieur ou égal à 1, alors la méthode nous retourne *true*. Dans le cas contraire, *false*, il n’y a plus de groupe possible, c’est la fin du jeu.

# V] Conclusions personnelles

Arthur :

Shana :