1. Rappel du sujet

Il est question ici des réaliser un jeu dans le langage java. Le principe de base est simple, un plateau est créés, avec un certain nombre de cases qui ont une couleur parmi les 6 couleurs de bases choisie aléatoirement. Les joueurs commencent sur un coin du plateau avec une couleur. Le but est de choisir une couleur à chaque tour pour ainsi conquérir les cases adjacentes de cette même couleur et de réaliser le meilleur score.

1. Modélisation

Mon code est divisé en 5 packages :

* Main
* Map
* Player
* Gui
* data

1. Main
2. Main

La classe Main est la classe principale du projet Filler. C’est donc dans cette classe qu’est présent la méthode main, exécutée au lancement de l’application.

Cette classe étends de la classe PApplet, qui est la librairie de du logiciel Processing. (<https://processing.org/>) C’est une librairie graphique qui a deux fonctions principales :

* settings () : Initialisation les paramètres du jeu
* draw () : Appelée 60 fois par secondes, c’est là qu’on va mettre à jour le jeu.

On a ensuite la fonction mousePressed () : Cette fonction sera appelée quand un bouton de la souris est pressée. Elle donne accès à la position de la souris sur les axes X et Y de la fenêtre d’affichage. La fonction keyPressed () est sensiblement pareille mai intervient lors de la saisie au clavier.

Voilà pour la classe principale qui est enfaite juste un intermédiaire entre la librairie de Processing et la classe Filler.

1. GameState

GameState est une énumération des états de jeu. J’ai décidé de mettre en place ce système d’état afin de pouvoir séparer facilement les différentes phases dans le jeu : *MENU*, *INGAME* et *SETTINGS*. On va pouvoir visualiser son utilité réelle dans le paragraphe suivant.

1. Filler

Attributs principaux :

* static PApplet p : référence à classe Main, ça me permet de pouvoir accéder aux fonctions de la librairie de Processing statiquement en appelant « Filler.p »
* static Game game : instance de la partie en cours.
* static GameState gameState : définit l’état actuel de l’application.
* static Player currentPlayer : joueur qui est en train de jouer.

Constructeur :

* Définition de la taille de la fenêtre, ici toujours 800x600 pixels.
* Chargement des images utilisées pour les éléments graphiques de l’application et aussi du fichier d’options prédéfinies au format .json.
* Initialisation des paramètres de jeu, tels que : l’état premier de l’application la taille et le type de la grille de base, chargement des 6 couleurs.

L’objet Filler n’est créé qu’une seule fois au démarrage de l’application dans la classe Main. Ensuite il est actualisé par la fonction update (), cette méthode est appelée dans la méthode draw () de la classe Main. Ainsi après son initialisation, l’objet Filler est actualisé à chaque instant.

Méthode update () :

C’est ici qu’on va voir l’utilité de l’énumération GameState. En effet la fonction update () actualise différentes choses en fonction du gameState actuel, si l’application est dans l’état MENU, alors on va actualiser la classe StartMenuGui, qui est la classe qui définit l’interface utilisateur du menu principale. Alors que si l’état actuel est INGAME, l’application actualise la classe InGameGui, et l’instance du jeu en cours : game.

Ce principe est répété dans la méthode mousePressed (). Si l’état est MENU alors on appelle la fonction mousePressed () de la classe StartMenuGui.

1. Game

Attributs principaux :

* static Map<MapType> map : instance du plateau de jeu.
* static ArrayList<Player> players : tableau constitué des joueurs présents dans la partie actuelle.

Constructeur :

* Initialisation du plateau en fonction du type choisit dans les options dans le menu du début.
* Création des joueurs humains et artificiels.

Méthode begin () :

Cette méthode est appelée quand le client appui sur le bouton « Play » dans le menu. Elle créée une nouvelle instance de la classe Game.

Méthode display () :

Cette méthode est appelée à chaque instant et ne fait qu’appeler la méthode display () du plateau de jeu.

1. Map
2. Map

La classe Map est abstraite et générique. C’est-à-dire qu’on peut créer des plateaux de jeu de différents types de cellules, dans ce cas-ci : SQUARE.

Elle contient une méthode abstraite display () qui sera différente selon le type du plateau de jeu.

1. MapType

Cette classe abstraite me sert à définir la forme des cellules du plateau de jeu, c’est-à-dire si elles sont de type Square, ou pourquoi pas hexagonale. Grace à ça mon code est extensible car pas limité à un seul type de cellule et y incorporé un nouveau type est simple.

Elle contient une méthode abstraite display () qui sera différente selon le type du plateau de jeu.

Attributs :

* int x, y (éventuellement z) : coordonnées de la cellule.
* Color color : couleur de la cellule, de base la couleur est choisie aléatoirement.
* int domination : c’est un entier qui définit l’index (dans le tableau players de la classe Game) du joueur qui contrôle la cellule.
* boolean visited : booléen qui sert dans l’intelligence artificielle.

Constructeurs :

Il y a deux constructeurs pour cette classe abstraite. Le premier décrit un agencement des cellules sur le plateau de jeu à 2 dimensions et le second à 3 dimensions. (Par exemple des cellules hexagonales)

1. EMapType

Ici il est encore question d’une énumération, celle-ci définit les types possibles de cellule pour le plateau de jeu.

1. SquareMap

SquareMap est un package qui va contenir la classe SquareMap et la classe Square, c’est un des type de cellule possible.

1. SquareMap

Cette classe étend de Map<MapType> c’est-à-dire que c’est un plateau de jeu particulier.

Constructeur :

* C’est ici que le plateau de jeu est créé, et dans ce constructeur, on le remplie avec des objets de type Square.

Méthode display () :

Elle appelle la fonction display () de chacun des objets Square sur le plateau.

1. Square

Constructeur :

Dans le constructeur il y a l’initialisation des voisins de la cellule.

Méthode display () :

Fonction ultrasimple pour le type de cellule Square, on dessine un carré aux coordonnées x et y.

1. TriangleMap
2. TriangleMap

Le principe est exactement le même que pour la classe SquareMap

1. Triangle

Méthode display () :

Un petit peu plus complexe que pour dessiner un carré, j’utilise ici la fonction beginShape () qui me permet ensuite de spécifier des points qui seront les sommets des triangles.

1. Player
2. Player

Encore une classe abstraite qui va être la classe parente de HumanPlayer et de ComputerPlayer.

Attributs :

* Color color : couleur actuelle du joueur.
* String name : le nom du joueur.
* int nb : son numéro, utilisé pour la domination des cellules.
* int score : le nombre de cellules qu’il contrôle

Constructeur :

Dans ce constructeur on initialise :

* le nom
* le numéro
* la couleur
* la position de départ. (on initialise la domination de la cellule de départ par le numéro du joueur)

Méthode fill () :

Cette méthode est appelée quand un joueur capture une couleur. Elle créée une variable group dans laquelle il y aura toute les cellules de la couleur choisie qui touchent directement les cellules du joueur.

A la fin de la méthode on appelle la méthode récursive fill (group) expliquée ci-dessous.

Méthode fill (ArrayList<MapType> group) :

La variable group est un ensemble de cellules.

La méthode change la domination et la couleur de toutes les cellules à capturer dans la variable group. Ensuite elle cherche les cellules voisines des cellules précédemment capturées et les capture à leur tours, ainsi de suite jusqu’à ce qu’il n’y ait plus de cellule voisine qui soit de la couleur choisie.

Méthode pass () :

Cette méthode passe le tour du joueur qui vient de finir de jouer. Elle change tout simplement le currentPlayer de la classe Filler.

Méthode checkColor (Color color)

Cette méthode renvoie true si la couleur demandée est disponible à la capture.

Méthode calculateScore ()

Cette méthode calcule le score du joueur, il est actualisé à chaque fois que le joueur joue.

Méthode resetVisited ()

Utilisée par les intelligences artificielles pour remettre à false la variable visited de toutes les cellules du plateau.

1. HumanPlayer

C’est la classe qui décrit un joueur humain.

Méthode update ()

Alors ici, la méthode update () ne fera rien tant que la variable choosedColor ne sera pas initialisée avec une couleur.

Par contre, dès que le joueur choisit une couleur (on verra le principe de sélection de la couleur un peu plus tard) alors on change sa couleur avec la couleur choisie, ensuite le processus de capture est lancé avec la méthode fill () ensuite on réinitialise la variable choosedColor à null et enfin on fait passer le tour du joueur avec la méthode pass ().

1. ComputerPlayer

C’est la classe qui décrit un joueur ordinateur, qui va jouer avec une intelligence artificielle.

Méthode update ()

Un peu différente de celle du joueur humain, dans cette classe la méthode update () exécute une intelligence artificielle qui va choisir une couleur à capturer. Ensuite le déroulement est le même que pour la méthode du joueur humain.

Méthode easyAI ()

C’est une intelligence artificielle assez simple. La fonction met dans un tableau le nombre de fois que chaque couleur apparait dans les voisins directe du terrain que le joueur a déjà capturé. Ensuite elle choisit la couleur qui apparait le plus de fois.

1. Gui
2. Gui

C’est la classe parente à toutes les interfaces utilisateurs. J’y ai mis une méthode qui est utilisée dans tous les GUI.

1. StartMenuGui

Rien de spécial pour cette classe, elle permet de lancer le jeu, en lançant la méthode Game.begin (), d’accéder au menu d’options ou encore de quitter l’application.

1. SettingsGui

Ce GUI permet de changer quelques options pour la partie à venir, telles que la taille du plateau ou encore les noms des joueurs.

Dans cette classe on gère beaucoup d’action à la souris avec les coordonnées des boutons à l’écran.

1. VictoryGui

Ecore rien d’extraordinaire pour cette classe, c’est juste pour gérer la victoire d’un joueur, et rediriger vers le menu ou proposer de rejouer.

1. Data

Le package Data est utilisé par la librairie Processing, c’est ici qu’on doit mettre les fichiers de ressources dont on a besoin dans l’application, j’y aie donc mis les images de fonds, le fichier d’options et même une police d’écriture.

1. Conclusion

Pour finir je pense que mon architecture est plutôt bien construite et me permet de moduler assez facilement l’application. Le fait d’avoir divisé le code en beau coup de classes rend le code épuré et on peut s’y retrouver facilement. Et sinon la partie graphique a été plutôt sympathique à programmer avec la librairie Processing. J’ai essayé de soigner le visuel de l’application.