

II.1102 – Projet : Robot Turtles Rapport technique



JULIEN COLOMBAIN - DAVID LAMY-VERDUN - DYLAN HU

ANNÉE 2019 - 2020

Table des matières

1	Introduction	2
2	Présentation du projet	2
2.1	Règles du jeu	2
2.2	Travail attendu	2
3	Architecture	3
3.1	Diagramme UML	3
3.2	Description des classes principales	3
3.2.1	Deck	3
3.2.2	Game	4
3.2.3	InterfaceLWJGL	4
3.2.4	Main	4
3.2.5	Occupier	4
4	Interface graphique	5
4.1	Réalisation des fenêtres	5
4.2	Réalisation des boutons	8
5	Implémentation du jeu	10
5.1	Modélisation du plateau de jeu	10
5.2	Orientation et déplacement des tortues	10
5.3	Obstacles et lasers	10
5.4	Initialisation du jeu	11
5.5	Déroulement d'un tour	11
5.5.1	Compléter le programme	12
5.5.2	Construire un mur	13
5.5.3	Exécuter le programme	15
5.5.4	Défausser sa main	16
5.6	Identification de fin d'une partie et scores	17
6	Conclusion	17

1 Introduction

Ce document technique décrit l'implémentation du jeu vidéo Robot Turtles en Java, dans le cadre du projet du module Algorithmique et Programmation (II.1102). Nous allons tout d'abord situer le contexte en rappelant les règles de ce jeu ainsi que les exigences attendues puis nous présenterons l'architecture de notre projet, la réalisation de notre interface graphique et notre implémentation du jeu. Les structures de données utilisées ont été nommées en Anglais en respectant les conventions Java. Le jeu ainsi que les commentaires sont en français. Le code source est accessible [ici](#).

2 Présentation du projet

2.1 Règles du jeu

”Robot Turtles est un jeu de plateau conçu pour introduire des notions élémentaires de l’algorithmique à des jeunes enfants. Robot Turtles se joue de 2 à 4 joueurs, et chaque joueur incarne une tortue se déplaçant sur un plateau de taille 8 x 8. L’objectif de chaque joueur est d’amener sa tortue sur un joyau placé sur le plateau en construisant un petit algorithme. Cet algorithme est construit à l’aide de cartes qui permettent de faire avancer la tortue ou de la faire tourner d’un quart de tour vers la gauche ou la droite. Des cartes supplémentaires permettent de créer des obstacles ou de les détruire.” Texte extrait de l’énoncé de Patrick Wang. Pour plus de détails, s’y référer [ici](#).

2.2 Travail attendu

Notre travail consiste à développer le jeu Robot Turtles en Java en respectant les règles citées avant. De plus, le jeu doit être jouable pour 2 à 4 joueurs sur une interface graphique conçue et implémenter les fonctionnalités suivantes :

- initialisation du jeu
- tour de jeu
- identification de la fin d’une partie
- calcul des scores

3 Architecture

3.1 Diagramme UML

Notre projet est constitué de 4 packages principaux. Voici une représentation des classes développées et de leurs relations. Les attributs ainsi que les méthodes de classe n'ont pas été représentés afin de préserver la vue du lecteur.

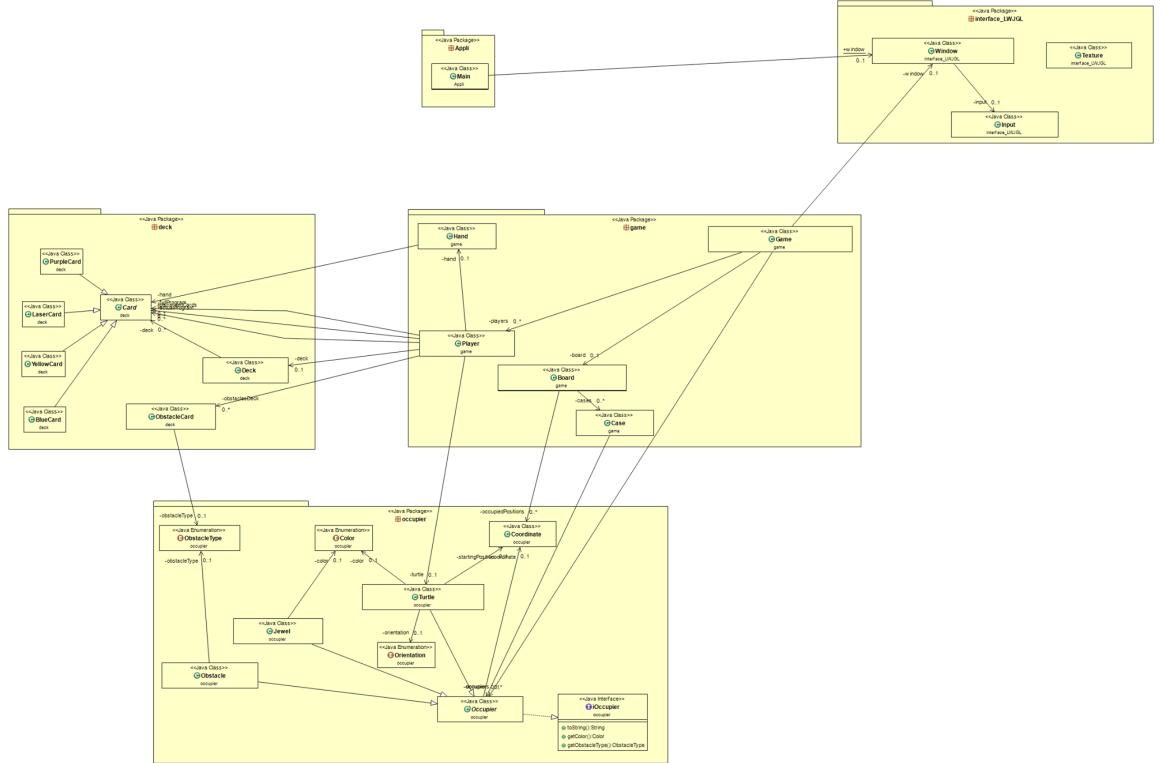


Figure 1: Diagramme UML

3.2 Description des classes principales

3.2.1 Deck

La classe `Deck` permet de modéliser le contenu du deck d'un joueur. Ce deck contient 37 cartes :

- 18 cartes bleues qui font avancer la tortue d'une case
- 8 cartes jaunes qui font tourner la tortue de 90° dans le sens anti-horaire

- 8 cartes violettes qui font tourner la tortue de 90°dans le sens horaire
- 3 cartes lasers qui permettent aux tortues d'utiliser leur laser pour attaquer l'obstacle devant elle

Cette classe contient un attribut *deck* qui est une *ArrayList* de d'instances de la classe *Card*. Le constructeur de cette classe crée un deck en lui ajoutant les cartes nécessaires dans les bonnes quantité. Pour modéliser toutes les cartes, nous avons défini une classe abstraite *Card* ainsi que 5 sous-classes : *BlueCard*, *PurpleCard*, *YellowCard*, *LaserCard* et *ObstacleCard*, qui représentent respectivement les cartes bleues, violettes, jaunes, les lasers et les obstacles.

3.2.2 Game

La classe *Game* permet le déroulement du jeu. Elle fait appel aux classes suivantes :

- *Board* pour représenter le plateau de jeu
- *Case* pour représenter une case sur le plateau
- *Player* pour représenter un joueur
- *Hand* pour représenter les cartes en main d'un joueur

3.2.3 InterfaceLWJGL

La classe *InterfaceLWJGL* constitue notre interface graphique. Elle permet la création de fenêtre est de boutons. Son fonctionnement est décrit en [partie 4](#).

3.2.4 Main

La classe *Main* permet l'exécution du programme. Son fonctionnement est décrit en [partie 5](#).

3.2.5 Occupier

La classe *Occupier* permet de modéliser un occupant sur une case, pouvant être une tortue ou un obstacle. Elle fait appel aux structures suivantes :

- *Color* : énumération pour représenter la couleur d'une tortue ou d'un joyau
- *Coordinate* : classe pour représenter les coordonnées d'un occupant sur le plateau
- *Jewel* : classe pour représenter les joyaux
- *Obstacle* : classe pour représenter les obstacles

- ObstacleType : énumération pour représenter le type d'obstacle (mur de glace, mur de pierre ou caisse en bois)
- Orientation : énumération pour représenter l'orientation d'une tortue (nord, sud, est ou ouest)
- Turtle : classe pour représenter une tortue

4 Interface graphique

Afin de réaliser l'interface graphique de notre projet, nous avons décidé d'utiliser LWJGL (LightWeightJavaGameLibrary), une librairie Java utilisée habituellement pour la réalisation de jeux. Cette librairie est extrêmement vaste et comprend plusieurs bibliothèques dont GLFW qui nous a permis de réaliser notre fenêtre.

```
package interface_LWJGL;
import org.lwjgl.glfw.Glfw;
import org.lwjgl.glfw.GlfwVidMode;
import org.lwjgl.glfw.GlfwWindowSizeCallback;
import org.lwjgl.opengl.GL;
import org.lwjgl.opengl.GL11;
```

4.1 Réalisation des fenêtres

Pour la réalisation de notre fenêtre, nous avons utilisé ces modules ci-dessus provenant tous de la librairie LWJGL. La première permet elle de faire appel au package dans lequel se situe la librairie étant donné que nous avions décider de créer un package spécifique pour cette dernière.

```
public class Window {
    private int width;
    private int height;
    private String title;
    private long window;
    private Input input;
    private float backgroundR, backgroundG, backgroundB;
    private GlfwWindowSizeCallback sizeCallback;
    private boolean isResized;
    private boolean isFullScreen;
    private int[] windowPosX;
    private int[] windowPosY;
    private String whichWindow;
}
```

Dans notre interface, notre fenêtre est une classe *Window* comportant les attributs ci-dessus. Dans ces attributs figurent des caractéristiques que toute fenêtre possède, à savoir, une hauteur, une largeur, un titre, une couleur de fond et des booléens, un premier afin de savoir si la fenêtre est en mode fenêtré ou bien plein écran et un second afin de savoir si la fenêtre a été redimensionnée.

On peut aussi noter parmi ces attributs l'utilisation de deux objets de LWJGL, Input qui permet de créer des entrées pour interagir avec notre fenêtre mais aussi GLFWWindowSizeCallback qui permet de contrôler les modifications sur le redimensionnement de la fenêtre.

Enfin, un attribut propre à notre projet et la chaîne de caractères *whichWindow* qui nous permet de savoir sur quel écran de jeu nous sommes placés, à savoir, le menu principal, le menu de sélection du nombre de joueurs ou bien le plateau de jeu.

```
/**
 * Crée une fenêtre graphique selon une taille et un
 * titre donné
 * @param width , la largeur de la fenetre en pixel
 * @param height , la hauteur de la fenetre en pixel
 * @param title , le titre de la fenetre
 */
public Window(int width , int height , String title) {
    this.width = width;
    this.height = height;
    this.title = title;
    windowPosX = new int [1];
    windowPosY = new int [1];
    whichWindow = "home";
}
```

Lors de l'instanciation de notre fenêtre, nous avons choisi d'utiliser en argument, deux entiers représentant respectivement largeur et hauteur et enfin une chaîne de caractères spécifiant le nom. Nous avons aussi instancié notre fenêtre sur le menu principal grâce à *whichWindow*.

Afin d'afficher du contenu sur notre fenêtre nous nous sommes servi d'une classe nommée *Texture*. Cette classe possède une méthode *draw* qui nous permet de dessiner au sein de notre fenêtre les images des instances de *Texture*.

```
private void home() {
    new Texture("game_title").draw(-0.5f , 1f , 0.5f ,
        -0.28f);
    new Texture("turtle_purple").draw(-1f , 1f , -0.7f ,
        0.33f);
```

```

new Texture("turtle_red").draw(0.7f, 1f, 1f, 0.33
f);
new Texture("turtle_green").draw(-1f, -0.33f,
-0.7f, -1f);
new Texture("turtle_blue").draw(0.7f, -0.33f, 1f,
-1f);
new Texture("btn_play").draw(-0.2f, -0.46f, 0.2f,
-0.61f);
new Texture("btn_exit").draw(-0.2f, -0.7f, 0.2f,
-0.85f);
}

```

Voici par exemple le code que nous avons utilisé pour créer l'affichage de notre menu principal. Chaque tortue est en réalité une image qui a été dessiné, de même pour le titre et les boutons "Jouer" et "Quitter". Ce ne sont que des images dessinées les unes auprès des autres et correctement agencées.



Figure 2: Notre menu principal

La méthode *draw* prend en argument les abscisses et ordonnées à partir desquelles les images devraient être dessinées et jusqu'où elles devraient l'être. Nous avons décidé d'utiliser des flottants car cela nous permet de garder des images de ratio fixes et de tailles redimensionnables à souhait.

Afin de placer nos textures où on le souhaite avec ces flottants, il est nécessaire d'effectuer des conversions.

Nous avons défini pour notre fenêtre une hauteur de 1280 et une largeur de 760, ce qui nous fait une demi-hauteur de 640 et une demi-largeur de 380. Au sein de LWJGL, la fenêtre est définie en flottant de -1 à 1 à la fois en largeur et en hauteur. -1 correspondant en largeur au bord latéral gauche et en hauteur au bord latéral bas.

Voici un exemple pour le bouton "Jouer" :

```
new Texture("btn_play").draw(-0.2f, -0.46f, 0.2f, -0.61f);
```

On sait qu'en largeur, -1 représente 0 pixel et 1 représente 1280 pixels ; si l'on souhaite que notre bouton soit centré par rapport à l'axe horizontal, il est nécessaire qu'il soit dessiné autour du flottant 0. Ce qui est bien le cas : $\frac{-0.2+0.2}{2} = 0$. Pour les dimensions, nous procédons de la même manière, il est possible de passer de flottants à pixels : $|0.2 - (-0.2)| * 640 = 256$ et $| -0.61 - (-0.46)| * 380 = 57$. En notant respectivement X, Y, A et B, les bords gauche, droite, haut et bas de nos textures, et P la taille en pixel on peut généraliser : $(Y - X) * 640 = P$ et $(B - A) * P$.

4.2 Réalisation des boutons

Contrairement à de nombreuses interfaces graphiques, LWJGL ne permet pas de créer des boutons. Ainsi, nos boutons se résument en réalité à de simples images auxquelles on associe une "zone active", qui lorsque sollicitée, générera une entrée.

Tout comme pour le menu principal, nous utilisons la méthode *draw()* de la classe *Texture* afin de générer le visuel de nos boutons. Suite à cela, nous utilisons des conditions sur les entrées de la souris. En effet, avec LWJGL, nous pouvons récupérer la position X en largeur et Y en hauteur du pixel sur lequel on clique. Ainsi il suffit d'utiliser des conditions afin de savoir si le clic est dans une zone de pixels correspondante à un bouton.

Si l'on reprend l'exemple du bouton "Jouer" utilisé précédemment, nous pouvons lui ajouter une zone active afin de le transformer en bouton effectif et qu'il ne soit non plus qu'une simple image :

```
if ( Input .isButtonDown (GLFW.GLFW_MOUSE_BUTTON_LEFT) && (
    Input .getMouseX () >= (8*window .getWidth () /20) && Input
    .getMouseX () <= (12*window .getWidth () /20) )) {
    if ( Input .getMouseY () >= (14.6*window .getHeight ()
        /20) && Input .getMouseY () <= (16.1*window .
        getHeight () /20)) {
        window .setWhichWindow ("menu");
        window .updateScreen ();
    }
}
```

Avec la méthode *isButtonDown()* de la classe *Input*, il est possible de récupérer quel bouton a été utilisé.

Ensuite nous utilisons les méthodes *getMouseX()* et *getMouseY()* de cette même classe pour où à eu lieu le clic. C'est ici que nous nous servons des conditions citées plus haut.

À l'aide des méthodes *getWidth()* et *getHeight()* de la classe *Window*, nous pouvons récupérer la taille de notre fenêtre afin d'effectuer des conversions similaires aux conversions des flottants pour délimiter une "zone active".

En divisant la largeur de notre fenêtre par 20, nous obtenons des portions de fenêtre dont les dimensions sont relatives à la taille de la fenêtre, comme pour les flottants. Contrairement à LWJGL qui utilise des flottants compris entre -1 à 1, nous nous servons pas de valeurs négatives pour définir les "zones actives". Il suffit donc de convertir ces flottants en suffisant d'ajouter 1. Par ailleurs afin d'obtenir un comportement similaire à LWJGL qui découpe l'écran en 2 à partir du centre, nous avons divisé par 2 les valeurs obtenus. Ainsi $-0.2f$ correspondrait à $0.8 * window.getWidth() / 2$ dans notre cas, ce que nous préférons écrire $8 * window.getWidth() / 20$ pour plus de lisibilité et commodité.

5 Implémentation du jeu

5.1 Modélisation du plateau de jeu

Pour modéliser le plateau de jeu, nous avons développé une classe *Board* contenant un tableau de 2 dimensions appelé *cases* de taille 8 x 8 et une *ArrayList* de coordonnées appelée *occupiedPositions*. On peut ainsi stocker en mémoire la position des différents occupants sur le plateau, qu'il s'agisse d'une tortue, d'un joyau, ou d'un obstacle.

5.2 Orientation et déplacement des tortues

Pour stocker en mémoire l'orientation des tortues, nous utilisons l'énumération *Orientation*. Pour modifier l'orientation d'une tortue, on utilise la méthode *setOrientation()* de la classe *Turtle*.

Pour réaliser les déplacements, on se sert du système de coordonnées afin de pouvoir modifier la case qu'occupe une tortue se déplaçant. Les méthodes *getCoordX()* et *getCoordY()* de la classe *Turtle* nous fournissent les coordonnées d'une tortue. On les modifie donc avec les méthodes *setCoordX()* et *setCoordY()*. On peut ainsi déplacer une tortue sur le plateau de jeu.

5.3 Obstacles et lasers

Pour modéliser les obstacles, on utilise la classe *Obstacle*, sous classe de *Ocupier*. On stocke en mémoire la position des obstacles de la même manière que la position des tortues. Pour identifier le type de chaque obstacle, on utilise l'énumération *ObstacleType*. On peut ainsi savoir si l'obstacle est un mur de glace, de pierre ou une caisse en bois. Nous avons distingué les cartes obstacles qui restent dans la main des joueurs correspondant à la classe *ObstacleCard* et les obstacles posés correspondant à la classe *Obstacle*.

Pour le cas du laser, nous avons défini la classe *LaserCard* ainsi que la méthode *fireLaser()* qui permet de définir le comportement du laser face aux obstacles lorsqu'un joueur souhaite utiliser une carte laser.

- Si le laser rencontre un mur de glace, le mur est détruit
- Si le laser rencontre un joyau, le laser est réflechi et frappe donc la tortue qui a tiré le laser et le renvoie à sa position initiale
- Si le laser touche une tortue, il la renvoie à sa position initiale

5.4 Initialisation du jeu

Lors de l'initialisation du jeu, nous utilisons la méthode *update()* de la classe *Main* afin d'afficher le menu principal. Lors du clic sur le bouton "Jouer", cette même fonction sera appelée pour cette fois-ci afficher le menu de sélection du nombres de joueurs. En fonction du nombre de joueurs choisi on crée alors une instance de la classe *Game* qui prend en argument ce nombre. Dans le constructeur de la classe *Game* on appelle la méthode *initPlayers()* de cette même classe qui à nouveau prend en argument le nombre de joueurs choisi. Cette méthode a pour but d'instancier le début de partie en créant le nombre de joueurs nécessaires, plaçant les tortues et enfin les joyaux. Suite à l'instanciation de notre classe *Game*, on fait alors appel à la méthode *launchGame()* de cette classe. Cette méthode est répétée sans arrêt jusqu'à ce que l'on décide de fermer la fenêtre, elle a pour but d'afficher l'écran de déroulement du jeu et appeler les différentes méthodes correspondant à l'action sélectionnée en fonction du bouton sollicité, *completeProgram()*, *buildWall()* et *executeProgram()*.

5.5 Déroulement d'un tour

Au cours d'un tour, le joueur a le choix entre compléter son programme, construire un mur ou exécuter son programme. Il peut également à la fin de son tour faire le choix de défausser sa main pour obtenir de nouvelles cartes. Nous allons présenter les différentes fonctions permettant d'effectuer ces actions. Chacune de ces fonction prend en argument un entier faisant référence au joueur. Par exemple, 1 désigne le joueur 1.

5.5.1 Compléter le programme

Lorsque le joueur complète son programme, il peut cliquer sur les cartes qui sont actuellement dans sa main et visible en bas à droite de l'écran. Les cartes sélectionnées sont retirées de la main, ajoutées à l'attribut liste *currentProgram* de la classe *Player* et alors affichées en haut à droite de l'écran. Si le joueur veut revisiter le programme de ce tour-ci, il peut alors cliquer sur les cartes de *currentProgram* pour effectuer l'action inverse, retirer les cartes de *currentProgram* et les renvoyer dans sa main.

Si le joueur sélectionne le bouton "Phase suivante", les cartes de *currentProgram* sont ajoutées dans le bon ordre à l'attribut liste *fullProgram* de la classe *Player* correspondant au programme complet du joueur qui reste à être exécuté. Une fois l'ajout des cartes effectuées, *currentProgram* est alors vidée et le tour se termine.



Figure 3: Menu du joueur pour compléter son programme

5.5.2 Construire un mur

Pour construire un mur, nous avons développé la méthode `buildWall()`. Cette méthode est composée de deux parties.

Tout d'abord, le joueur voit les murs qu'il lui reste à placer et se doit de sélectionner lequel il veut placer. Une fois un mur sélectionné, il peut alors cliquer sur une des cases du plateau afin de placer ce dernier. Si le mur sélectionné est un mur de glace, alors ce dernier est placé si la case sélectionnée est libre sans aucune autre contrainte. En revanche, si le mur sélectionné est fait de pierre, on vérifie que ce mur n'obstrue pas le chemin grâce à la méthode `isItBuildable()` ; `isItBuildable()` permet de vérifier qu'il existe toujours un chemin entre :

- Une tortue et un joyau
- Une des positions initiales des tortues au joyau

Pour ce faire, `isItBuildable()` appelle la méthode `findPath()` pour toutes les coordonnées correspondant aux positions actuelle des tortues et leurs positions initiales. `findPath()` est une méthode prenant en paramètre une position de départ et déterminant un chemin jusqu'au joyau. Elle se base sur un algorithme de découverte de chemin nommé Astar (ou A*) :

- On crée une liste de coordonnées à visiter
- On y met les coordonnées de la position de départ renseignées en paramètre
- On retire de la liste des coordonnées à visiter, les coordonnées qu'on visite actuellement
- On regarde les coordonnées adjacentes, si aucun mur de pierre ne s'y trouve, on ajoute ces coordonnées à la liste des coordonnées à visiter
- On assigne aux prochaines coordonnées à visiter une des coordonnées de la liste des coordonnées à visiter
- On vérifie si les coordonnées d'un joyau appartiennent à la liste des coordonnées à visiter. Le cas échéant on retourne `true`
- On réitère jusqu'à ce que la liste des coordonnées à visiter est vide. Si la liste est vide, on a visité toutes les coordonnées voisines disponibles sans trouver de joyau et retourne donc `false`

Durant notre implémentation, nous avons dû apporter certaines modifications dues aux contraintes imposées par les choix que nous avons pris dans le développement de notre jeu. En lieu et place d'une liste des coordonnées à visiter, nous avions deux listes `toVisitCoordinatesX` et `toVisitCoordinatesY` correspondant respectivement aux coordonnées en X et aux coordonnées en Y à visiter et enfin un tableau de booléens de deux dimensions de taille 8*8 `toVisitCoordinates` correspondant aux coordonnées à visiter. Ce tableau de booléens est initialisé rempli de `false` et les booléens deviennent `true` lorsque les index de la case correspondante du tableau appartiennent aux listes `toVisitCoordinatesX` et `toVisitCoordinatesY`.

Afin de pouvoir utiliser cet algorithme A* et donc cette méthode `findPath()` par l'intermédiaire de `isItBuildable()`, il est nécessaire de simuler un plateau à l'état N+1 afin de déterminer si **une fois le mur placé** le joyau serait ou non toujours accessible. Pour cela, nous plaçons donc mur de pierre au clic de souris mais n'actualisons pas l'affichage. Nous nous contentons d'actualiser la liste des positions occupées et les occupants figurant dans chacune des cases à l'aide des méthodes `takenCases()` et `refreshCases()` de la classe `Board`.

Si `isItBuildable()` retourne true, alors la construction du mur n'obstruerait aucun chemin et on retire alors simplement le mur de la main du joueur et passe au tour suivant ; autrement, la construction du mur obstruerait un chemin auquel cas nous retirons le mur du plateau et rafraîchissons à nouveau la liste des positions occupées et les occupants figurant sur chaque case à nouveau par le biais des méthodes `takenCases()` et `refreshCases()` de la classe `Board`.



Figure 4: Menu du joueur pour choisir un obstacle

5.5.3 Exécuter le programme

Pour exécuter le programme d'un joueur, nous avons développé la fonction `executeProgram()`. Cette fonction va regarder les différentes étapes du programme exécuté. S'il y a une carte jaune, on déplace la tortue du joueur de 90°dans le sens anti-horaire. S'il y a une carte violette, on déplace la tortue du joueur de °dans le sens horaire. S'il y a une carte bleue, on appelle la fonction `obstacleEncountered()` qui vérifie l'occupant qui pourrait se trouvait dans la case en face de la tortue. Si cette case est vide, la tortue avance et se déplace d'une case. Si cette case est occupé par un obstacle qui n'est pas un joyau, la tortue n'avancera pas adoptera un comportement différent en fonction de l'obstacle. Si l'obstacle n'est pas un joyau, deux choix s'offrent alors, soit l'obstacle est un mur, ou bien une tortue :

- Dans le cas d'un mur : la tortue effectue sur elle-même une rotation de 180°.
- Dans le cas d'une tortue : les deux tortues, la tortue courante et la tortue rencontrée, font tous deux un tour de 180° sur elles-même.



Figure 5: Menu du joueur lorsque son programme est exécuté

5.5.4 Défausser sa main

Pour défausser sa main, nous avons développé la fonction `discardCards()`. Cette fonction est appelée après chaque fin d'action, que ce soit après `executeProgram()`, `completeProgram()` ou `buildWall()`. On affiche ensuite deux boutons et on crée deux zones actives : "Défausser" ou sur "Ne pas défausser". Si le joueur appuie sur "Défausser", toutes les cartes de `Hand` sont ajouté à la défausse et sa main est vidée. Si le joueur clique sur "Ne pas défausser", il ne se passe rien. À chaque fin de tour, on exécute la fonction `endTurn()` qui appelle `drawHand()` qui va compléter la main du joueur. Ainsi, si le joueur a défaussé sa main et qu'elle est vide, il va piocher 5 cartes. S'il n'a pas défaussé, il pioche un nombre de cartes jusqu'à en avoir 5 dans sa main. Ces cartes là provenant du deck, dans le cas où il est vide, on place la défausse du joueur et la remet dans son deck.



Figure 6: Menu du joueur pour défausser sa main

5.6 Identification de fin d'une partie et scores

Nous avons fait le choix d'arrêter une partie dès qu'une des tortues atteint son joyau. Pour réaliser cela, nous déclarons un booléen *victory()*, initialisé à *false* ainsi qu'une méthode *isVictory()* qui renvoie la valeur de ce booléen. Chaque fois qu'un joueur exécute son programme, la fonction *executeProgram()* est appelé. Lorsque le programme contient une carte bleue qui fait avancer la tortue, la fonction *obstacleEncountered()* a pour rôle de regarder l'occupant qui se trouve dans la case devant la tortue. Si c'est le joyau correspondant à la tortue, *victory()* devient *false*. Le jeu se trouvant dans une boucle *while(!isVictory)*, cela marque ainsi la condition d'arrêt et le jeu se termine. On obtient ainsi le message suivant en fonction du numéro du joueur qui a gagné.

VICTOIRE DU JOUEUR 1

Figure 7: Message de victoire

6 Conclusion

Robots Turtles est un projet relativement intéressant qui permet d'appréhender les notions de classes et ainsi la programmation orientée objet. Ce projet a aussi été l'occasion de se familiariser avec Git, un outil indispensable pour une bonne cohésion d'équipe, et l'opportunité de se pencher sur la création d'une interface graphique.