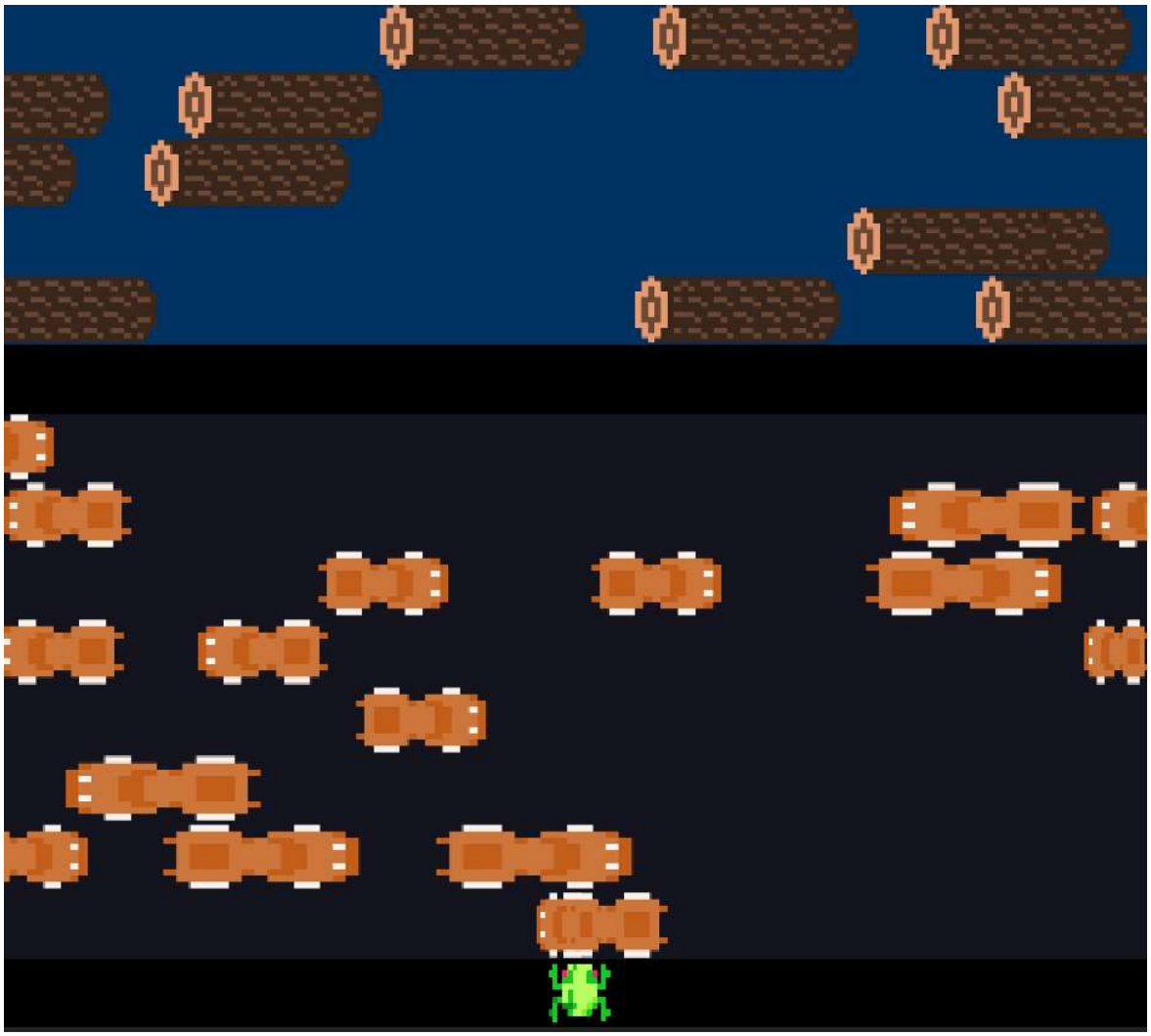

Rapport du projet de POO Java : Frogger

ENSTA Bretagne

2 rue F. Verny

29806 Brest Cedex 9, France

FOUQUET, Delphine,
delphine.fouquet@ensta-bretagne.org

KHA, Alexandre,
alexandre.kha@ensta-bretagne.org

NZONGANI, Sam,
sam.nzongani@ensta-bretagne.org

Remerciements

Nous tenons à remercier M. MAZO PENA ainsi que Mme HNAINI pour leurs conseils, enseignements et encadrement au cours de notre projet.

Table des matières

1. Introduction

Frogger est à la base un jeu d'arcade développé et sorti par Konami en 1981. Il a connu un large succès à sa sortie, ce qui l'a rapidement propulsé au rang de jeu iconique.

Le jeu consiste à contrôler, dans un format 2D, une grenouille dans le but de traverser une autoroute sans se faire écraser par les voitures, puis de sauter sur des rondins de bois et des tortues qui circulent le long d'une rivière sans tomber à l'eau. Ceci pour au final atteindre une ligne d'arrivée, ce qui marque la victoire au jeu.

La programmation de ce jeu s'inscrit dans le cadre du cours de Programmation Orientée Objet (POO) en JAVA de M. MAZO PENA.

Ainsi nous avons formé un trinôme, l'équipe 7, afin de travailler sur ce projet. Cette équipe est composée de :

- FOUQUET Delphine (spécialité Systèmes Numériques et Sécurité)
- KHA Alexandre (spécialité Systèmes d'Observation et Intelligence Artificielle)
- NZONGANI Sam (spécialité Systèmes Numériques et Sécurité)

Nous utiliserons les librairies provenant du logiciel Processing pour produire l'interface graphique. Cette idée est inspirée par la chaîne Youtube TheCodingTrain qui illustre l'utilisation de Processing en codant des jeux, dont une version simple de Frogger.

Dans un premier temps le projet a consisté à coder une version du Frogger sans ligne d'eau mais avec seulement une autoroute et des voitures.

Plus tard une version avec ligne d'eau puis un mode infini ont été implémentés.

La version finale présente un menu qui permet de choisir le nombre de joueurs (1 ou 2 qui jouent sur les touches de direction et Z/Q/S/D), puis la difficulté de jeu :

- Autoroute seulement
- Autoroute et ligne d'eau
- Mode infini (seulement en 1 joueur).

Il y aura également la possibilité dans ce menu de consulter un tableau de scores.

Dans ce rapport nous détaillons le cahier des charges initial du projet, les améliorations implémentées, la structure du code, les garanties de qualité du code ainsi que nos méthodes de travail.

2. Cahier des charges fonctionnel

2.1. Cahier des charges de base

Nous présentons le cahier des charges fonctionnel de base du programme sous la forme du tableau ci-dessous :

Fonctions	Enoncé des fonctions	Critères d'appréciation	Niveaux d'exigence
FP1	Permettre à 1 joueur d'avoir le visuel du jeu sur un écran d'ordinateur	- Visibilité de la fenêtre de jeu - Objets visuellement reconnaissables - Fluidité du défilement - Condition de fin de partie	- Fenêtre de jeu de 16 cases de largeur par 19 cases de hauteur, avec des cases de 45 pixels - Grenouille représentée par un élément vert, autoroute sombre, voitures de la même couleur autre que vert - Mouvements continus des obstacles et absence de "lags" - Objectif du jeu très explicite
FP2	Permettre au joueur de jouer en utilisant le clavier	- Touches utilisées	- Mouvement de la grenouille lors du pressage des touches de direction avec possibilité de rester maintenu pour continuer le mouvement
FC1	Doit être fait en langage JAVA	- Librairies JAVA utilisables	- Toutes les librairies sont utilisables en dehors de celles "deprecated" (obsolètes)
FC2	Doit correspondre au paradigme de POO	- Structure du code - Niveau d'abstraction propre à la POO	- Structure proposée avec 4 packages : frog, gameCommons, graphicalElements, util - Utilisation de façon optimale des héritages, du polymorphisme et des interfaces
FC3	Doit éviter d'être redondant et ennuyant	- Taille, vitesse et sens de déplacement des voitures	- Tailles, vitesse et sens de déplacement des voitures initialisés de façon aléatoire
FC4	Doit fournir des preuves de qualité	- Présence de tests unitaires - Gestion des exceptions	- Chaque méthode doit avoir au moins un test unitaire lorsque cela est

			possible (i.e. en dehors de l'IHM) - Utilisation des outils de gestion d'exception (try/catch/finally) dans les parties du code qui sont sujettes à d'éventuels problèmes et affichage de messages clairs lors des levées d'exception

2.2. Divergences et nouvelles fonctionnalités

On présente ci-dessous un tableau qui récapitule les fonctionnalités modifiées et ajoutées par rapport au cahier des charges précédent :

Nouvelles fonctions	**Enoncé des nouvelles fonctions**	**Critères d'appréciation**	**Niveaux d'exigence**
FP3	Permettre à un 2e joueur de jouer sur le même ordinateur	- Personnage jouable - Nombre de fenêtres de jeu - Touches utilisées - Condition de fin de partie	- Deuxième grenouille jouable - Jeu simultané sur la même fenêtre de jeu - Z/Q/S/D pour le deuxième joueur - La première grenouille qui atteint l'arrivée marque la fin du jeu (et gagne)
FC2	Doit correspondre au paradigme de POO	- Structure du code - Niveau d'abstraction propre à la POO	- Création d'objets pour les voitures et les troncs en plus dans la structure proposée. Création d'une classe Button pour le menu de jeu - Utilisation de façon optimale des héritages, du polymorphisme et des interfaces
FC3	Doit éviter d'être redondant et ennuyant	- Taille, vitesse et sens de déplacement des voitures	- Tailles, vitesse et sens de déplacement des voitures initialisés de façon aléatoire

		- Modes de jeu - Système de compétition	- Au moins deux modes de jeu (autoroute et autoroute + ligne d'eau) - Mise en place d'un classement de scores
FC5	Doit être esthétiquement plaisant	- Représentation d'objets - Signalétique d'évènements	- Utilisation d'images pour représenter les éléments du jeu - Utilisation de sons et visibilité à tout moment d'un score à l'écran de jeu - Fin de partie explicitement annoncée
FC6	Doit être facilement utilisable	- Présence d'instructions - Visuellement parlant	- Notice d'instructions en README pour le descriptif du jeu ainsi que pour l'installation préalable de librairies - Menu de jeu avec possibilités de click sur des boutons

3. Architecture du logiciel

3.1. Description succincte des classes et de leurs constructeurs

3.1.1. Package MovingElements

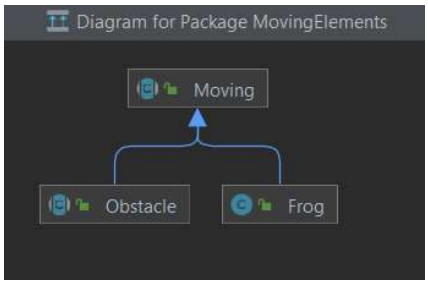

Figure 1 : Package MovingElements

Dans ce package nous implémentons la classe abstraite Moving qui servira de classe mère pour tous les objets mobiles du jeu, à savoir la grenouille (Frog) et les obstacles (Obstacle) dont descendent les voitures (Car) et les troncs d'arbre (Trunk).

Un objet mobile sera modélisé par un rectangle. Ainsi la classe Moving a pour attributs l'abscisse du côté gauche (*left*) du rectangle, l'abscisse du côté droit (*right*), l'ordonnée du côté supérieur (*top*) et l'ordonnée du côté inférieur (*bottom*).

De plus elle possède les attributs *range* pour le numéro de rangée du rectangle, *width* pour sa largeur, *height* pour sa hauteur et finalement *speed* pour sa vitesse (bien que cette donnée soit superflue pour la grenouille). La seule méthode de cette classe est *move(int xdir, int ydir),* qui sert à faire déplacer les objets mobiles et qui sera surchargée par chaque sous-classe (Frog et Obstacle) afin de faire bouger les éléments mobiles.

Frog hérite de tous les attributs de Moving, initialisés toutefois pour en faire un carré (largeur et hauteur égale) à la rangée 0 et à la vitesse 0. De plus Frog possède trois autres attributs booléens, *car_intesection, trunk_intersection* et *GAMEOVER* indiquant respectivement si Frog a intersecté une voiture, s'il est sur un tronc ou si Frog a déjà perdu. m*ove* est surchargée de façon à changer les coordonnées de Frog en y additionnant les paramètres xdir et ydir de la méthode. Frog dispose finalement d'une méthode *intersect(Obstacle other)* qui renvoie un booléen signalant si Frog intersecte un Car ou un Trunk. On notera cependant que l'intersection n'est pas perçue de la même manière si l'obstacle rencontré est un Car ou s'il est un Trunk. En effet, il suffit que Frog touche un Car pour que la méthode renvoit *true* (cas où Frog est "écrasé") tandis qu'il faut que Frog se situe à l'intérieur d'un Trunk pour qu'elle renvoie *true* (cas où Frog "vit").

La classe Obstacle hérite de tous les attributs de Moving et possède en plus *abs_limit* et *ord_lim* qui renseignent sur l'abscisse et l'ordonnée maximales que l'objet peut atteindre (concrètement il s'agit de la largeur et de la hauteur de la fenêtre de jeu). m*ove* est surchargée comme chez Frog, à la différence que si l'obstacle dépasse la fenêtre de jeu (ce que l'on sait notamment grâce à *abs_limit* et *ord_lim*), il est réinitialisé à l'extrémité opposée de l'écran. Les sous-classes Car et Trunk de Obstacle héritent de tous ses attributs et de *move*.

3.1.2. Package GameCommons

Figure 2 : Package GameCommons

Dans ce package, dédié à la gestion d'une partie de jeu, nous créons la classe Game qui centralise toutes les actions qui servent au bon déroulement du jeu (initialisation des objets, vérification des intersections de la grenouille, reconnaissance de la fin de partie).

La classe Game a pour attributs :

- *grid* (le nombre de pixels d'une case)
- *ranges* (le nombre de rangées)
- *columns* (le nombre de colonne)
- *game_width* (la largeur de la fenêtre de jeu, soit columns*grid)
- *game_height* (la hauteur de la fenêtre, soit ranges*grid)
- Le booléen gameState qui devient *true* si Frog est arrivé à la fin du parcours.
- Les String *PlayerMode, Diff* et *leaderboard,* qui indiquent respectivement le nombre de joueurs par "1 PLAYER" ou "2 PLAYERS", la difficulté de jeu "EASY", "HARD" ou "INFINITE" et enfin la difficulté du Leaderboard. Ils sont initialisés à *null.*

Nous détaillons les méthodes implémentées par la classe Game dans la sous-partie suivante dédiée aux interfaces.

3.1.3. Package GraphicalElements

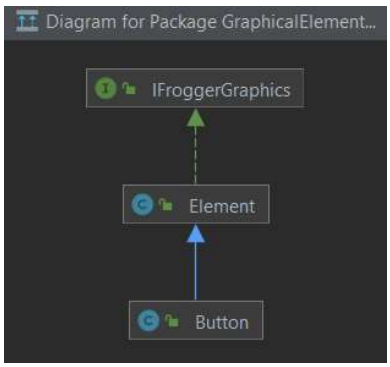

Figure 3 : Package GraphicalElements

Dans ce package dédié à la création d'éléments graphiques, tels que les cases ou l'affichage d'images, nous créons la classe Element.

Celle-ci a pour seul attribut une instance de la classe PApplet, laquelle est utilisée pour lancer un *main* et pour produire une interface graphique. La classe PApplet est disponible dans la librairie processing.core qui sera expliquée plus en détail par la suite. Toutes les méthodes d'Element réemploient les méthodes de PApplet sans toutefois les surcharger. On les détaille dans la sous-partie suivante traitant des interfaces.

Plus tard nous créons également la classe Button, qui descend d'Element et qui permet de spécifier un type de case particulier avec lequel on peut interagir (détection lorsque l'on clique dessus, changement de couleur lorsque la souris est dessus). La classe possède en attributs une instance de PApplet, les coordonnées du bouton ainsi que le texte qui s'affichera sur le bouton.

3.1.4. Package Util

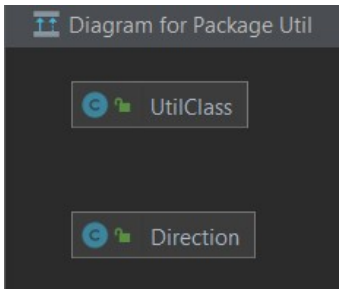

Figure 4 : Package Util

Ce package comporte deux classes sans constructeurs : Direction et UtilClass. La première référence les valeurs de keyCode pour pouvoir ensuite jouer en cliquant les touches du clavier (par exemple : si la touche correspondant à Direction.UP est cliquée, alors telle action). La deuxième regroupe plusieurs méthodes utiles à travers le projet, notamment pour la génération d'une valeur aléatoire entre deux bornes ou pour lire un fichier .txt, ce qui est utile lorsqu'on veut montrer un tableau de scores.

3.2. Interfaces et méthodes principales

L'interface est une manière de "faire passer un contrat" à une classe, car en implémentant une interface, une classe doit redéfinir toutes les méthodes abstraites de cette interface. En outre une interface renforce la clarté du code en triant les méthodes d'une classe sous une certaine fonctionnalité, ce qui nous intéresse particulièrement ici. Dans cette sous-partie nous voyons comment les interfaces sont utilisées et quelles sont les méthodes qu'elles contiennent.

3.2.1. Interfaces implémentées par Game : IFrog et IEnvironment

Les interfaces IFrog et IEnvironment décrivent les actions disponibles pour la grenouille et l'environnement respectivement. Toutes les méthodes de Game proviennent exclusivement de ces interfaces.

Trois méthodes permettent de gérer la grenouille :

- *set_Frog(int num)* créé une instance de Frog qui sera contrôlée par un joueur. Cette instance est initialisée au milieu de l'extrémité inférieure de la fenêtre en "1 PLAYER". En "2 PLAYERS", si *num* vaut 1, la grenouille est initialisée à ¼ de l'extrémité inférieure de la fenêtre, autrement elle est initialisée à ¾.
- *resetFrog(Frog frog, int num)* permet de réinitialiser quasiment tous les attributs d'un Frog comme en début de partie (*set_Frog(int num)*) tout en prenant en compte le PlayerMode ainsi que le numéro de joueur (paramètre *num)*
- *stateFrog(Frog frog, int num)* est la méthode phare de cette interface, conçue pour vérifier si frog meurt (intersection avec un Car ou chute dans la ligne d'eau), auquel cas la méthode réinitialise frog grâce à *resetFrog()*, mais elle peut également modifier gameState en *true* lorsque frog atteint la ligne d'arrivée, ce qui pourra enclencher la procédure de fin de jeu.

L'interface IEnvironment définit 4 principales méthodes afin d'initialiser les obstacles dans la fenêtre de jeu :

- *car_range(int i)* et *trunk_range(int i)* créent respectivement une ArrayList de Car et de Trunk en rangée i, le nombre d'obstacles étant choisi aléatoirement entre 2 et 3 pour *car_range()* et 3 pour *trunk_range()*. Chaque obstacle de ces listes est initialisé avec une taille et une vitesse aléatoire, et on notera qu'une rangée paire aura un sens de circulation différent qu'une rangée impaire.
- *allCars(int e)* et *allTrunks(int b,int e)* produisent respectivement des ArrayList d'ArrayList de Car et de Trunk, qui regroupent tous les obstacles sur toutes les rangées spécifiées en paramètres. Ces listes définissent entièrement les obstacles du jeu.

La dernière méthode, *move_allCars(ArrayList<ArrayList<Car>> cars, float ydir)* n'a d'utilité que dans le mode de difficulté "INFINITE", permettant de bouger verticalement d'un pas ydir toutes les instances de Car de la liste spécifiée en paramètre. En effet, en pressant la touche de direction haute, on ne fait plus bouger la grenouille, mais toutes les voitures vers le bas (ce qui donne l'effet visuel que c'est la grenouille qui bouge vers le haut).

3.2.2. Interface implémentée par Element : IFroggerGraphics

IFroggerGraphics définit les méthodes liées à l'interface graphique qu'Element devra implémenter. Ces méthodes utilisent massivement les méthodes de la classe PApplet. On retiendra notamment :

- *create_case(float left, …, float r, float g, float b),* qui comme son nom l'indique créé une case (un rectangle) aux coordonnées spécifiées et à la couleur définie par les float r,g et b. Cette méthode est notamment utile pour créer un décor, pour représenter par exemple l'autoroute, une ligne d'eau et la ligne d'arrivée. Dans les premières versions de notre Frogger, cette méthode servait également à représenter de façon simpliste la grenouille, les voitures et les troncs par des rectangles colorés.
- *show_image(PImage icon, int left, int bottom, int width, int height)*, qui affiche une PImage aux coordonnées et dimensions spécifiées.
- *show_frog(PImage icon, Frog frog), show_car(PImage icon, Car car)* et *show_trunk(PImage icon, Trunk trunk)* permettent de lier un objet mobile à une image prise en paramètre : ces méthodes affichent alors une image aux coordonnées de l'objet mobile en appelant *show_image()*.
- *create_text(String txt, …,int r, int g, int b)* créé un texte à la taille et aux coordonnées spécifiées, et avec une couleur définie par le triplet (r,g,b).

D'autres méthodes existent également notamment pour initialiser la fenêtre de jeu, la colorer et afficher une image d'arrière-plan.

3.3. Description des librairies processing.core et minim, et du Main

La librairie processing.core est inspirée du logiciel Processing fonctionnant sur la base du JAVA.

On remarquera le caractère fortement intuitif de la librairie.

Dans cette librairie on utilisera surtout la classe PApplet, dont on définira une instance en tant que Main. C'est elle qui permet de lancer une IHM.

On retiendra 4 méthodes principales de cette classe que l'on surcharge dans Main:

- *settings()* dans laquelle un utilisateur peut entrer des paramètres liés à l'IHM, comme la taille de la fenêtre par exemple. Elle n'est appelée qu'une fois lors du lancement du programme
- *setup()* dans laquelle sont initialisés les objets qui seront mobilisés dans l'IHM : par exemple on y déclare des instances de Game, Element, Frog, Car, Trunk, ou encore des paths et des images. Certaines de ces déclarations peuvent se faire dans *settings()* et vice-versa (mais pas toutes, par exemple la taille de la fenêtre se fait uniquement dans *settings*). Comme *settings()*, *setup()* n'est appelé qu'en début du lancement
- *keyPressed()* est une méthode qui va permettre d'associer des clicks sur les touches du clavier à des actions particulières. Dans la version standard du Frogger, cela servira à modifier les coordonnées de la grenouille lors des clicks sur les touches de direction. La méthode exploite le fait que chaque touche du clavier possède une valeur entière caractéristique appelée *keycode*
- *draw(),* qui comme son nom l'indique, permet de dessiner les objets et évènements que l'on veut représenter. Cette méthode est appelée en boucle sans arrêt. C'est dans celle-ci que réside réellement la structure du *main* ainsi le déroulement du jeu, ce que l'on détaille par la suite.

Dans le *draw()*, une première partie est dédiée à l'affichage du menu ainsi que des boutons "1 PLAYER", "2 PLAYERS" et "LEADERBOARD".

Un clic sur un bouton permet de passer à la page suivante avec d'autres choix, la difficulté du jeu ou alors la difficulté du leaderboard que l'on veut consulter. Concrètement, le clic permet de changer l'attribut *PlayerMode* du jeu (instance de Game), ce qui indique à l'application de redessiner un autre (même) arrière-plan ainsi que d'autres boutons.

L'accès à un Leaderboard est possible grâce à la lecture d'un fichier .txt recensant les scores et en les affichant.

Lorsque l'on clique sur la difficulté du jeu, l'attribut *Diff* du jeu est changé en conséquence et le jeu est réellement lancé. Voici les étapes du jeu :

- La ou les grenouilles sont initialisées via *game.set_Frog()* et peuvent être contrôlées via les touches de direction ou Z/Q/S/D. Un timer est aussi lancé et affiché dans le coin supérieur gauche dans tous les modes hors "INFINITE"
- De plus les obstacles sont initialisés selon la difficulté du jeu via *game.allCars()* et *game.allTrunks()*. Chaque obstacle entre en mouvement via la méthode *obstacle.move()*
- Chaque instance de Moving est affichée par une image grâce aux méthodes d'Element dérivées de *show_image()*
- *game.stateFrog()* vérifie à tout moment l'intersection entre une grenouille et les obstacles en surveillant le booléen renvoyé par *frog.intersect(obstacle).* Selon la valeur de ce booléen, la grenouille est réinitialisée ou non. Si une grenouille atteint la

dernière rangée du jeu, la méthode change *game.GameState* en true et le jeu s'arrête en affichant le temps du joueur

Le mode "INFINITE" est légèrement différent car on associe les clics sur la touche UP (ou Z) au mouvement des voitures vers le bas et non plus à la montée de la grenouille. De plus le score affiché est la rangée de la grenouille. Finalement la fin du jeu est déclenchée lorsque la grenouille intersecte une voiture.

Tout au long du *draw()* nous utilisons aussi la librairie minim provenant de Processing pour produire des sons ou des musiques lors des clics sur les boutons ou alors pendant une partie de jeu.

Il aurait été plus propre de les redéfinir dans une interface IFroggerSound implémentée par Element pour au moins 3 raisons :

- Inciter l'utilisateur à passer par Element lors de leur exploitation, par soucis de cohérence avec la structure du code
- Simplifier les appels de plusieurs méthodes successives en une seule (par exemple pour régler le gain et lancer le son)
- Gérer dans ces méthodes les exceptions éventuelles, et ne pas le faire dans le *Main*

Par faute de temps, nous n'avons pas pu réaliser cette tâche.

4. Garanties de qualité

4.1. Gestion des exceptions

Les principales exceptions pouvant être retournées lors de la compilation du code concernent la gestion de fichiers à lire, comme des images ou des tableaux de scores.

Les plus relevées dans ce code sont alors les *NullPointerException* retournées par les méthodes graphiques de la classe Element, ainsi que les *NoSuchFileException* pour les méthodes de traitement de fichier d'UtilClass.

Les premières exceptions sont résolues en permettant de continuer le jeu en affichant cependant des cases blanches ou un fond noir lorsque les images correspondantes sont mal déclarées. Si un texte est *null,* il ne sera tout simplement pas affiché.

De plus lors de ces levées d'exception, un message d'erreur est affiché en rouge au coin de l'écran, du type "Warning : Paths specified for images not found", comme nous pouvons le voir ici :

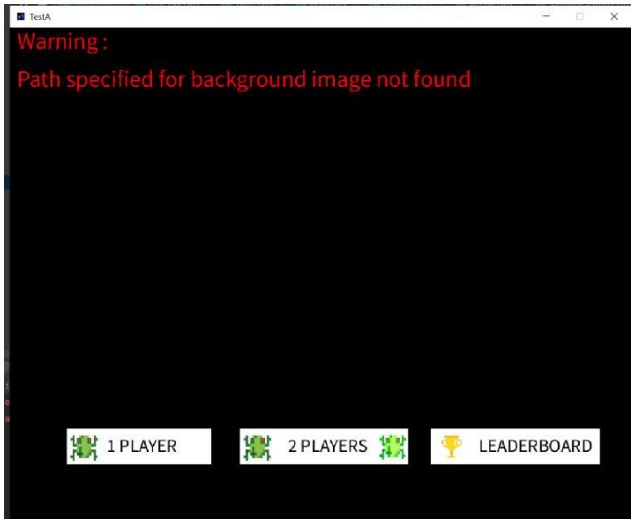

Figure 5 : Exemple de levée d'une NullPointer Exception

Les levées des IOException des méthodes d'UtilClass n'induisent pas de procédure particulière et laissent tourner le jeu. Mais éventuellement, elles produiront une *NullPointerException* dans les méthodes graphiques (par exemple lors de la lecture d'un texte *null* à afficher), qui sera alors gérée comme vu précédemment.

On notera cependant que toutes les exceptions liées à un path incorrect ne sont pas gérées dans le code, notamment ceux des fichiers audios lus par les méthodes du module *minim*.

La levée de ces exceptions ne porte pas de problème particulier mais peut rendre le *main* lourd vu le nombre d'appels qui y sont faits. Pour pallier ce problème, on aurait pu, comme vu dans la partie précédente, redéfinir les méthodes de ce module tout en les encapsulant à l'intérieur, et gérer les exceptions dans ces méthodes (ce qui est le cas pour les méthodes d'Element).

4.2. Tests unitaires

Nous avons testé les méthodes qui nous semblaient les plus utiles et testables. Ainsi, nous n'avons pas testé les méthodes d'IHM (c'est-à-dire tout le package GraphicalElements), les méthodes des interfaces et des classes abstraites, ainsi que les méthodes de notre Main.java.

Pour réaliser nos tests, nous avons utilisé JUnit avec les librairies Maven. Nous avons notamment utilisé la libraire Assertions qui a permis par exemple de vérifier qu'une valeur renvoyée par la méthode testée était bien égale à la valeur attendue, en utilisant assertEquals(valeur_attendue, valeur_testée).

Faire ces tests nous a été bien utile. En effet, en testant la méthode move par exemple, nous nous sommes rendu compte que nous avions justement une erreur dans cette méthode. Nous avons réalisé nos tests au fur et à mesure de l'écriture du code, afin de ne pas être bloqué par la suite si une de nos méthodes ne marchait pas.

5. Méthode de travail

5.1. Outils collaboratifs

En dehors des réunions, le principal outil utilisé pour échanger a été l'application Messenger,dans laquelle nous avions créé un groupe. Si au début du projet nous l'utilisions assez peu (1 à 2 fois par semaine), vers les dernières semaines nous l'utilisions quasiment quotidiennement en raison du temps libre des vacances et pour être dans les temps de rendu du projet.

De plus nous avons utilisé GitHub afin de partager nos codes, ce qui a pu s'avérer particulièrement pratique pour être rapidement à jour avec les versions de chacun des membres.

Finalement nous avons utilisé Teams pour rédiger ce rapport ainsi que pour préparer la soutenance du projet.

5.2. Planning de travail

Le planning de travail n'a pas été conçu à l'avance mais s'est fait progressivement. Chaque semaine, nous proposions des tâches pour la semaine d'après, et chacun choisissait d'en faire une selon ses préférences. Le planning se récapitule comme suit :

- 1ere semaine (début Octobre) : Premiers objets : Frog, Car. Initiation à la librairie Processing
- 2e semaine : Mise en place de la structure de code préconisée
- 3e semaine : Version fonctionnelle avec des cases pour les objets mobiles. Mode de jeu à 2, utilisation d'images et tableau de scores
- 4e semaine : Version infinie, utilisation de sons et implémentation d'un menu de jeu à partir des travaux précédents
- 5e semaine (semaine du 01/11) : Factorisation, gestion d'exceptions, tests unitaires et préparation de la soutenance

Conclusion

Durant ce projet, nous avons commencé par coder un modèle très simple du Frogger, avec une grenouille en guise de case verte et des voitures en guise de cases rouges, puis nous avons ensuite implémenté progressivement des améliorations, pour la ligne d'eau, le mode de jeu à deux, l'affichage d'un score. Finalement, on s'est penché sur l'aspect esthétique du jeu, en permettant l'affichage d'images, ainsi que d'un menu avec divers modes de jeu, ce qui nous a permis de répondre au cahier des charges que l'on s'était donné.

La partie interface graphique a pu être facilitée par le caractère très intuitif des librairies de Processing, ce qui nous a laissé le temps de nous concentrer sur l'implémentation des objets du jeu.

Cependant, le projet nous a posé des problèmes à plusieurs niveaux.

Tout d'abord au niveau de la coordination, car il fallait souvent analyser le code de chacun avant de pouvoir continuer, ce qui pouvait prendre un certain temps. Les commentaires ne suffisant pas à la compréhension du code, le plus souvent il était nécessaire de se retrouver, de s'appeler ou encore d'échanger par messages. Par contre il n'y a eu aucun souci par rapport au respect des tâches et délais que l'on s'était imposés.

Une deuxième difficulté a été la prise en main de GitHub, auquel nous n'étions pas familiers. Il a fallu un certain temps pour se documenter sur ses fonctionnalités avant que l'on commence à utiliser GitHub, du moins efficacement.

Une troisième et dernière difficulté a été d'associer la structure de code préconisée avec l'utilisation de la classe PApplet de processing.core. Par exemple, nous avons choisi que la classe Element devait prendre en paramètre une instance de PApplet, le Main, pour pouvoir définir des méthodes graphiques (car finalement ce sont des méthodes de PApplet que l'on utilise intrinsèquement). Mais cela induit un degré d'abstraction qui peut nous interroger sur ce que représente vraiment une instance d'Element. Dans notre cas, nous pouvons la voir comme "le contrôle de la fenêtre associée à l'application courante" (que l'on appelle d'ailleurs board dans le Main) et qui affiche ce que l'utilisateur spécifie. De plus il est aussi arrivé que l'on se demande quelle méthode irait le mieux dans quelle classe (*move* a été définie dans Frog mais peut l'être dans Game aussi), même si au final il semble que cela pose peu de problème, pourvu que l'on reste cohérent dans le Main.

On relèvera également qu'il existe plusieurs subtilités à l'utilisation de processing.core.

Déjà, une case peut continuer d'exister en dehors de la fenêtre de jeu, ce qui incite à ne garder qu'un nombre limité d'objets à gérer dans la fenêtre, par souci d'optimisation de la mémoire.

De plus il existe un bug graphique qui est pallié par l'utilisation d'images mais qui existe bel et bien : lorsqu'un obstacle passe de l'autre côté de l'écran et est réinitialisé à l'extrémité opposée, il accumule un retard ou une avance, ce qui fait qu'il peut intersecter un autre obstacle de la rangée. A priori les opérations mathématiques exécutées sur ces objets ne sont pas en cause, mais plutôt la gestion simultanée par l'application de plusieurs objets à la fois.

Au final le code semble être plutôt ergonomique dans le sens où l'implémentation de nouveaux objets (des obstacles par exemple) et leur déclaration se fait facilement.

Par exemple, on pourrait créer un oiseau ennemi qui survolerait les rangées du jeu de façon aléatoire et rapide en créant une classe Bird descendant de Car (en fait il serait plus approprié de redéfinir le nom de cette classe en Hostile par exemple). Alors en vertu de ce qui a été fait et sans beaucoup plus d'ajout, la grenouille perd le jeu en rencontrant un oiseau.

Mais même si le code du Main est assez linéaire, on pourra cependant regretter que l'on y spécifie trop d'actions de contrôle (par exemple : Pour chaque tronc, si frog intersecte le tronc alors son attribut *Trunk_intersection =true)*, alors que des méthodes de game pourraient s'en charger pour rendre le Main moins lourd et plus compréhensible, ce qui peut constituer un point d'amélioration du code.

Annexe

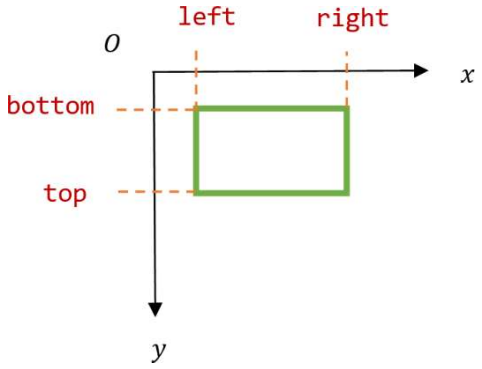

Figure 6 : Attributs en coordonnées d'un objet de type Moving

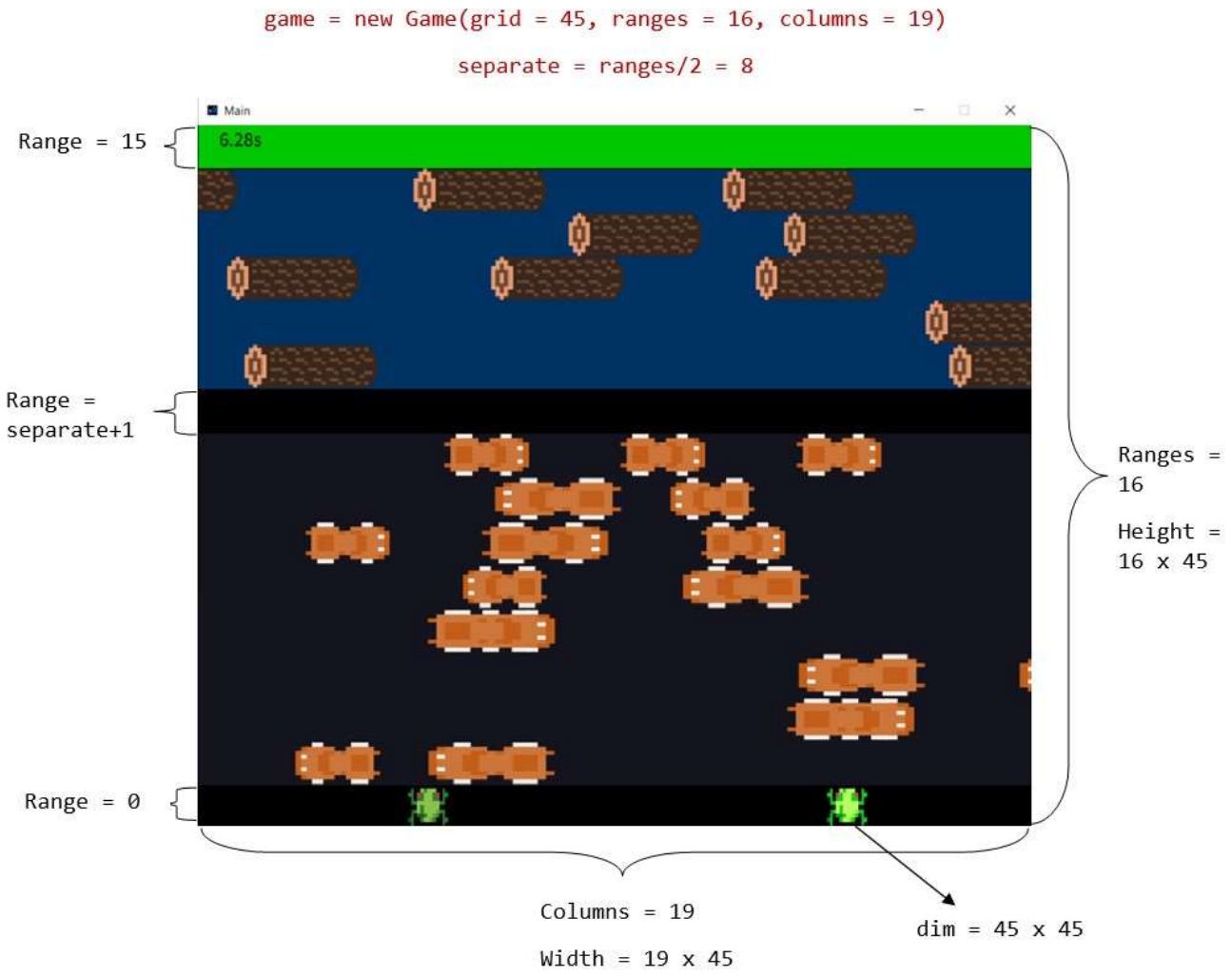

Figure 7 :
Représentation d'un jeu