2023-2024

Une image contenant Police, Graphique, logo, graphisme

Description générée automatiquement

Soltan Lenny

Rapport de projet : HELBTower

Table des matières

[Introduction 2](#_Toc155538232)

[Description des fonctionnalités de base 4](#_Toc155538233)

[Les *Character*: 4](#_Toc155538234)

[Le *MainChar* : 4](#_Toc155538235)

[Les *Guard*: 5](#_Toc155538236)

[L’OrangeGuard : 5](#_Toc155538237)

[Le BlueGuard : 6](#_Toc155538238)

[Le RedGuard : 6](#_Toc155538239)

[Le PurpleGuard : 6](#_Toc155538240)

[Les *GameElement*: 7](#_Toc155538241)

[Le *Coin*: 7](#_Toc155538242)

[Les *Potion*: 8](#_Toc155538243)

[La *Cloak*: 8](#_Toc155538244)

[Les *Teleporter :* 8](#_Toc155538245)

[Le *Wall*: 9](#_Toc155538246)

[Le *Controller :* 10](#_Toc155538247)

[La *View*: 13](#_Toc155538248)

[Fonctionnalités supplémentaires 18](#_Toc155538249)

[Le *Chronometer*: 18](#_Toc155538250)

[La direction des *Character & Sprites 2D*: 19](#_Toc155538251)

[Analyse 20](#_Toc155538252)

[Limitation technique 22](#_Toc155538253)

[*Chronometer* fuite de mémoire : 22](#_Toc155538254)

[*RedGuard* reste bloquer : 22](#_Toc155538255)

[*BlueGuard* peut rester bloqué : 22](#_Toc155538256)

[L’architecture logicielle peut être améliorée : 22](#_Toc155538257)

[Avec plus de temps ? : 23](#_Toc155538258)

[Conclusion 23](#_Toc155538259)

# **Introduction**

Dans le cadre du cours de programmation Java III, il nous a été donné comme projet de produire une application avec une interface graphique à l’aide du *framework Java FX*, en respectant une architecture *Model*, *View*, *Controller* (MVC). Celle-ci se nomme HELBTower et tel que demandé, voici ce qui a été réalisé.

Un *Controller*, une *View* et plusieurs *Models* afin de gérer les différentes fonctionnalités et de séparer les responsabilités de chaque élément.

L’espace de jeu (*GameBoard*) ses délimitations sont des obstacles (*Wall)* aucun personnage ne peut les traverser.

Le héros (*MainChar*), il apparait au centre en haut de l’espace de jeu c’est lui que l’on contrôle.

Les pièces (*Coin*) sont des éléments du jeu qui sont collectable par le joueur ce qui a pour effet d’augmenter son score de 1 et d’augmenter la vitesse du garde mauve (*PurpleGuard)*. Lorsqu’elles sont toutes ramassées un nouveau niveau est généré et la vitesse des gardes est augmentée.

Les murs (*Wall*) sont des éléments du jeu qui bloque tous les personnages, au centre il forme une croix, s’il le joueur possède une cape il peut traverser une seule fois un mur. Ils forment aussi des tours dans chaque bord de l’espace de jeu. En haut à droite et gauche et en bas à droite et gauche, celles-ci abritent les gardes ce sont leur point d’apparition au début de la partie.

Les gardiens (*Guard)* sont des personnages du jeu qui tente d’éliminer le joueur, il y en a 4 :

* L’orange (*OrangeGuard*), il se déplace aléatoire vers le haut, le bas, la droite ou la gauche.
* Le bleu (*BlueGuard*), il se déplace vers une case qu’il n’a pas encore exploré choisi aléatoirement, s’il a exploré toutes les cases alors il recommence.
* Le rouge (*RedGuard*), il se déplace constamment en direction du joueur.
* Le mauve (*PurpleGuard*), il se déplace vers une tour choisi aléatoirement. Sa vitesse augment à chaque fois que le joueur ramasse une pièce.

Tous les personnages (*Character*) ont la même vitesse de base, sauf exception, et leurs déplacements sont les mêmes, haut/bas/gauche/droite.

Les potions (*Potion*) sont des éléments du jeu qui servent à aider le joueur. Elles servent à augmenter la vitesse du joueur, il y en a 3 différentes leurs seules différences sont la durée de leur effet par ordre décroissant : la rouge, la orange et la jaune.

Les capes (*Cloak*) sont également des éléments du jeu qui servent à aider le joueur. Elles permettent au joueur de traverser un seul mur, une seule fois.

La période de jeu, il y a le jour, la partie se déroule normalement ; Il y a le matin, les gardes sont ralentis et le terrain jauni comme à l’aube et il y a la nuit, les gardes accélère et le terrain s’assombrit.

Les téléporteurs (*Teleporter*) permettent au joueur de traverser l’espace de jeu de bord en bord instantanément. Il y en a 2 paires, en haut et en bas de couleur bleu et à droite et à gauche de couleur rouge.

Les codes de triche sont là pour manipuler la partie :

* 0 : Reset la partie en cours.
* 1 : Défini la période du jeu sur matin.
* 2 : Défini la période du jeu sur jour.
* 3 : Défini la période du jeu sur nuit.
* 4 : Fait apparaitre sur une position aléatoire une potion.
* 5 : Fait apparaitre sur une position aléatoire une cape.
* 6 : Fait apparaitre un garde de type aléatoire sur une position aléatoire.
* R : Stop le gardien rouge.
* B : Stop le gardien bleu.
* M : Stop le gardien mauve.
* O : Stop le gardien orange.
* S : Active ou désactive les chronomètres et réinitialise la partie.

Les chronomètres (*Chronometer*) est la fonctionnalité supplémentaire qui est possible d’activer ou désactiver. Il s’agit d’un élément du jeu qui à pour but d’aider le joueur en le faisant remonter dans le temps, c’est-à-dire le faire reculer d’un certain nombre de case et de faire ralentir les gardes pendant une certaine durée.

# **Description des fonctionnalités de base**

## **Les *Character*:**

*Character* est une classe abstraite du *model* qui représente tous les personnages du jeu.

Une image contenant texte, capture d’écran, logiciel, affichage

Description générée automatiquement

Figure classe abstraite Character.java

### Le *MainChar* :

La classe *MainChar* est une classe du *model* qui hérite de la classe *Character*,elle représente le héros dans le jeu, c’est lui que le joueur contrôlera. Il possède différentes méthodes de vérifications et différente apparence en fonction de sa direction.

Une image contenant dessin humoristique, capture d’écran, Animation, art

Description générée automatiquement

Figure images du Héro en fonction des différentes directions dans l'ordre : vers le bas, le haut, la gauche et la droite

### Les *Guard*:

Guard est une classe abstraite du *model* qui représente tous les gardes du jeu.

Une image contenant texte, capture d’écran, logiciel

Description générée automatiquement

Figure classe abstraite Guard.java

#### L’*OrangeGuard* :

Une image contenant dessin humoristique, capture d’écran

Description générée automatiquementLa classe *OrangeGuard* est une classe du *model* qui hérite de la classe *Guard*, elle représente le garde orange dans le jeu, comme cité précédemment il se déplace de manière aléatoire dans une des directions disponibles entre les quatre disponibles, grâce à un nombre généré aléatoirement entre 0 et 3 compris.

Figure images du garde orange en fonction de la direction dans l'ordre : vers le bas, le haut, la droite et la gauche

#### Le BlueGuard :

La classe *BlueGuard* est une classe du *model* qui hérite de la classe *Guard*, elle représente le garde bleu dans le jeu, comme cité précédemment il se déplace en direction d’une case choisi aléatoirement parmi celle qu’il n’a pas déjà exploré. Pour cela, il a besoin de connaitre la taille de l’espace de jeu, qui lui sera communiqué par le *Controller*. Ensuite, l’algorithme fonctionne de la façon suivante, il génère une direction vers laquelle il va se déplacer, si celui-ci n’arrive pas à l’atteindre, il retient la position sur laquelle il se situe et génère une nouvelle direction.



Figure images du garde bleu en fonction de la direction dans l'ordre : vers le bas, le haut, la droite et la gauche

#### Le RedGuard :

La classe *RedGuard* est une classe du *model* qui hérite de la classe *Guard*, elle représente le garde rouge dans le jeu, comme cité précédemment il se déplace constamment en direction du joueur. Pour cela, il a juste besoin d’avoir une cible qui est le *MainChar*, ainsi il peut obtenir sa position et se diriger vers celle-ci.



Figure images du garde rouge en fonction de la direction dans l'ordre : vers le bas, le haut, la droite et la gauche

#### Le PurpleGuard :

La classe *PurpleGuard* est une classe du *model* qui hérite la classe *Guard*, elle représente le garde mauve dans le jeu, celui-ci se déplace de tour en tour choisi aléatoirement et a une vitesse variable en fonction du nombre de pièces ramassées. Pour permettre cela, il a besoin de connaitre toutes les positions des tours, qu’il reçoit du *Controller* et il générera aléatoirement un nombre entre 0 et 3 compris pour choisir une des quatre tours.



Figure images du garde mauve en fonction de la direction dans l'ordre : vers le bas, le haut, la droite et la gauche

## **Les *GameElement*:**

*GameElement* est une classe abstraite du *model* qui nous a été fourni de base dans le *snakeFX, celle-ci représente tous les éléments du jeu*

Une image contenant texte, capture d’écran

Description générée automatiquement

Figure classe abstraite GameElement

### Le *Coin*:

Une image contenant cercle, jaune

Description générée automatiquement*Coin* est une classe du *model* qui hérite de *GameElement*, elle représente une pièce à ramasser dans le jeu. Elle contient uniquement la logique appliquée lorsqu’elle est récupérée, elle vérifie s’il reste des pièces à ramassé si ce n’est pas le cas elle déclare dans le *GameBoard*, le niveau comme étant terminé. Ensuite, elle déplace la pièce dans le vide pour la ‘supprimer’ et elle augmente le score de 1. Elles sont générées par le *GameBoard* sur toutes les cases qui ne contiennent pas déjà un élément du jeu.

Figure image de la pièce en jeu

### Les *Potion*:

*Potion* est une classe du *model* qui hérite de *GameElement*, elle représente une potion à utiliser dans l’espace du jeu. Elle contient une durée d’effet, un booléen afin de savoir si elle a été prise ou non et une couleur qui sera défini par l’index qu’on lui transmet lors de son instanciation. Sa méthode *triggerAction* est redéfinie pour la définir comme étant prise lorsque c’est le cas et mettre sa position dans le vide afin de la ‘supprimer’. La logique de son effet est gérée dans le *GameBoard* car lui seul gère quand le MainChar la récupère. Elles sont également générées par ce dernier aléatoirement sur une case qui n’est pas déjà occupé par un autre *GameElement*.

Figure images des trois potions par ordre croissant de puissance

### La *Cloak*:

*Cloak* est une classe du *model* qui hérite du GameElement, elle représente une cape que le joueur peut récupérer dans l’espace du jeu. Celle-ci est similaire à la potion à la seule différence qu’elle n’a pas de couleur différente. Elle permet au joueur de traverser un seul mur, une seule fois. Cette logique est définie dans le *GameBoard* car c’est lui qui détient la logique de savoir quand est-ce que le *MainChar* va la récupérer. Elles sont générées aléatoirement là ou ne se trouve pas déjà un *GameElement*.

Figure Image d'une cape en jeu

### Les *Teleporter :*

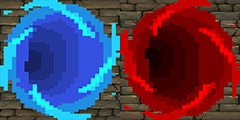
*Teleporter* est une classe du model qui hérite du GameElement, elle représente une paire de portail qui téléporte de l’un à l’autre. Il y en à quatre ou deux paires et ils se situent chacun dans les quatre bords de l’espace de jeu. Les bleus en haut et en bas, les rouges à droite et à gauche. Cette classe contient, de la même manière que les potions, un index pour la couleur et la position X et Y du second portail auquel il y est relié. La logique de téléportation se trouve dans le *GameBoard* puisque celui-ci peut manipuler le *MainChar*. Celle-ci est très simple, lorsque le *MainChar* rencontre la position d’un portail sa position est défini devant celle du portail homonyme.

Figure image des deux couleurs de portail

### Le *Wall*:

*Wall* est une classe du *model* qui hérite du *GameElement*, elle représente un obstacle pour le tous les personnages du jeu. Celle-ci est très simple, elle contient uniquement une image et une position X et Y.

Une image contenant texte, capture d’écran, logiciel, Logiciel multimédia

Description générée automatiquement

Figure classe Wall.java

Grâce à cette classe nous allons pouvoir générer les différentes structures du jeu, celles-ci seront générées à partir du *Controller* à l’aide du *GameBoard*. Il génèrera les bordures ensuite les murs centraux et pour finir les tours.

Une image contenant capture d’écran, Caractère coloré

Description générée automatiquement

Figure Capture d'écran du plateau de jeu après la génération de tous les éléments du jeu

## **Le *Controller :***

Une image contenant texte, capture d’écran

Description générée automatiquementLe *Controller*, dans le principe d’architecture logicielle du *MVC*, il sert à faire le lien entre tous les différents *Models* vue précédemmentet la *View,* il est en quelque sorte le point central du programme. C’est également là qu’on va initialiser toutes les variables pour la configuration de l’application.

Figure ensemble des constantes de configuration du Controller

Nous retrouvons la taille en hauteur et largeur, le nombre de lignes et de colonnes, l’ouverture entre les murs de la cour du château, le facteur X et Y de la position des tours, le nombre d’éléments du jeu générés et le délai des gardes durant le matin, le jour et la nuit. Dans son constructeur nous allons instancier les *Models* déclarer précédemment puis appelé la méthode de démarrage de l’application.\*

\*Lors de l’écriture de ce rapport la variable *model* a été renommé *gameBoard* pour plus de compréhension.

Une image contenant texte, capture d’écran, logiciel

Description générée automatiquementDans cette méthode de démarrage premièrement, nous avons la configuration de Java FX. Ensuite, nous avons la gestion d’interaction lorsque le joueur appuie sur une touche de son clavier. Il y a les déplacements du joueur mais également les codes de triches cités en introduction.

Figure condition qui gère les déplacements dans le Controller.java

Une image contenant texte, capture d’écran, logiciel

Description générée automatiquementPour finir, la méthode pour initialiser la partie est appelé à la fin de cette méthode de démarrage. Celle-ci va permettre tout simplement de démarrer une nouvelle partie lors du démarrage mais également de réinitialiser une partie en cours grâce à un booléen.

Une image contenant texte, capture d’écran, logiciel

Description générée automatiquement

Une image contenant texte, capture d’écran, logiciel

Description générée automatiquement

Figure méthode initGame du Controller partie 2

Figure méthode initGame du Controller.java partie 1

Figure méthode initGame du Controller.java partie 2

Figure méthode initGame du Controller.java partie 1

Nous pouvons voir qu’à la fin de la méthode, la *Timeline* est lancé avec un cycle d’une durée de 20 millisecondes ce qui permet de garder une application assez fluide. Cependant, nous verrons plus loin qu’il y a une gestion de cycle plus approfondie. Cette *Timeline* exécute indéfiniment la méthode *run.*

Dans celle-ci nous retrouverons toutes les actions qui se déroule au cours de la *Timeline*.

Figure méthode run du Controller.java

Une image contenant texte, capture d’écran, logiciel

Description générée automatiquement

Lors de ce cycle, nous vérifions d’abord si la partie est terminé et si c’est le cas le cycle prend fin en affichant l’écran de *Game Over*.

Ensuite, ce sont les méthodes de la View qui sont appelé afin d’afficher les différents éléments du jeu.

Après, c’est au tour des méthodes du *GameBoard*, ce sont elles qui vont gérer la logique pour voir si le niveau est terminé ou non, pour passer au suivant et aussi lorsqu’un élément du jeu est traversé afin d’interagir avec.

Par la suite, ce sont les gardes qui sont gérés. Dans un premier temps, l’algorithme vérifie si 25% des pièces sont ramassé pour donner vie aux gardes.

Secondement, comme cité plus haut, le programme vérifie si la durée du cycle du garde est dépassée afin qu’il puisse faire un mouvement. Pour cela nous utilisons ‘*System.currentTimeMillis()’* afin de récupérer le temps actuel en milliseconde, si le temps actuel est supérieur au moment enregistré la dernière fois qu’il a fait un mouvement PLUS le délai qu’on a défini alors il peut se déplacer.

Pour finir, le programme vérifie s’il le joueur est tué ou non et mets fin à la partie si c’est le cas.

## **La *View*:**

La *View* est la classe qui va gérer l’affichage de l’application, c’est elle qui va recevoir toutes les données pour afficher les images et les textes dans l’interface utilisateur.

Une image contenant texte, capture d’écran, Police

Description générée automatiquement

Figure Variables et constructeur de la View.java

Nous pouvons voir que les chemins vers les images du background sont hard codés. Il y en a deux par période pour créer une sorte damier pour améliorer la lisibilité du jeu.

Une image contenant motif, capture d’écran, Rectangle, carré

Description générée automatiquement

Figure images du background du jeu en fonction de la période, dans l'ordre : jour, matin et nuit

Également, nous pouvons remarquer qu’il y a trois *HashMap* différentes, une pour les personnages, une pour les éléments du jeu et une pour le cache. En effet, cette *View* est munis d’un cache d’image, son fonctionnement est simple. Pour commencer, les chemins vers les images de l’application sont chargés dans la *View* grâce à la méthode *loadPaths*.

Une image contenant texte, capture d’écran, Police

Description générée automatiquement

Figure méthode loadPaths de la View.java

Une image contenant texte, capture d’écran, logiciel

Description générée automatiquement

Figure méthodes loadGameElementPaths et loadCharacterPaths de la View.java

Les chemins sont récupérés grâce à la méthode *getPathToImage* qui est une méthode de base de l’objet *GameElement* qui nous a été fourni avec *SnakeFX,* celui-ci est placé dans le *gameElemPathMap* avec pour clé le nom de la classe. Cependant, lorsqu’il s’agit d’une potion ou d’un téléporteur on rajoute à ceux-ci la couleur afin d’afficher l’image adéquate. En ce qui concerne les personnages, leurs chemins vers leurs images sont récupérés et placer dans le *charPathMap* en fonction de leur direction car comme nous le verrons plus tard ceux-ci ont une direction.

Pour finir, les éléments sont affichés grâce à leur méthode respective les éléments du jeu dans *drawGameElements*, les personnages dans *drawChar* et le background dans *drawBackground*.

Une image contenant texte, capture d’écran, affichage, logiciel

Description générée automatiquement

Figure méthode drawGameElements de la View.java

Ici, l’entièreté des éléments du jeu sont parcouru et affiché en fonction de leur type. Lorsqu’il s’agit d’un téléporteur sa paire est dessiné en même temps, lorsqu’il s’agit d’une potion nous récupérons sa couleur pour afficher l’image adéquate.

Une image contenant texte, capture d’écran, Police

Description générée automatiquement

Figure méthode drawChar de la View.java

Une image contenant texte, capture d’écran, Police, affichage

Description générée automatiquement

Figure méthode getCurrentSkinPath de la View.java

Ici, les personnages sont dessinés en fonction de la direction dans laquelle ils se dirigent.

Une image contenant texte, capture d’écran, logiciel

Description générée automatiquement

Figure méthodes drawBackground et getBackgroundPath de la View.java

Là, le background est dessiné en damier comme expliqué plus haut. Nous pouvons remarquer que nous utilisons une condition ternaire pour faciliter l’envoie du chemin adéquat selon la période actuelle du jeu.

Une image contenant texte, capture d’écran, Police, affichage

Description générée automatiquement

Figure méthode drawImage de la View.java

C’est ici que la magie du cache prend forme, en effet lorsqu’on fait appel à cette méthode l’algorithme vérifie si l’image n’est pas déjà présente dans le *imageCache* si c’est le cas il la rajoute et l’affiche par la suite. Ce système permet de ne pas inonder la pile avec une nouvelle instanciation d’une nouvelle image à chaque affichage. Grâce à cela la cache contient toutes les images dont l’application à besoin ni plus ni moins, il n’y a aucune duplication.

# **Fonctionnalités supplémentaires**

## **Le *Chronometer*:**

*Chronometer* est une classe du *model* qui hérite du *GameElement*, elle représente un sablier qui peut être récupéré par le joueur. Il aura pour effet de ralentir la vitesse des gardes pendant un temps défini mais transporter le joueur dans le passé. Cette fonctionnalité est inspirée du principe de la franchise : *Prince Of Persia (développé et édité par Ubisoft)*, où le personnage principal détient un sablier du temps qui lui permet de remonter dans le temps pour éviter un drame.

Une image contenant texte, livre, fiction, affiche

Description générée automatiquementUne image contenant texte, capture d’écran, logiciel, Système d’exploitation

Description générée automatiquement

Figure l'un des premiers jeux de la franchise Prince Of Persia franchise appartenant à Ubisoft

Figure classe Chronometer.java

Des sabliers sont générés lorsque l’on active la fonctionnalité supplémentaire en appuyant sur ‘S’, la partie est réinitialisé et des sabliers sont générés grâce au *GameBoard* c’est également lui qui va gérer la logique lorsque l’on en récupère un car c’est lui qui gère la méthode pour récupérer un *GameElement*.

Voici comment il fonctionne :

Pour commencer le *MainChar* retient dans sa mémoire les positions sur lesquels il a marché grâce à un *ArrayList* ensuite lorsque le joueur ramasse un sablier celui-ci établi la position du joueur un certain nombre de case en arrière pour donner un effet de retour dans le temps et au même moment il augmente le délai des gardes afin de les ralentir. Dès, que le délai du sablier est dépassé celui des gardes est réinitialisé.

Figure image du sablier en jeu

## **La direction des *Character & Sprites 2D*:**

En effet, la direction des personnages est une fonctionnalité supplémentaire que j’ai décidé d’implémenter par défaut car je trouve que cela améliore grandement l’expérience utilisateur grâce au visuel que l’on peut leur donner. Comme nous avons pu le voir plus haut de nombreux *sprites* 2D ont été implémenté pour améliorer le design. Voici comment cela est géré dans le code :

Une image contenant texte, capture d’écran, logiciel

Description générée automatiquement

Figure méthode tryMove de la classe MainCharacter.java

# Une image contenant capture d’écran, texte, conception Description générée automatiquement**Analyse**

Nous pouvons remarquer que la structure de l’implémentation repose bel et bien sur le principe du *Model*, *View*, *Controller*. Nous avons le *Main.java* qui est le point d’entré du programme, dans celui-ci nous avons seulement une instanciation du *Controller.java*.

Dans celui-ci, nous avons toutes les instanciations des *Models* et de la *View*, en plus des variables et constantes de configurations vue plus haut. Dans le GameBoard, qui est un des *Model* du programme, se trouve toutes les générations d’éléments du jeu comme vue expliqué plus haut.

Tous ces éléments du jeu sont des classes qui ont héritées de la classe abstraite *GameElement*, elles possèdent donc par défaut une position X/Y un tableau de chemins vers une image et une méthode *triggerAction* qui sera déclencher lors de l’interaction avec cet élément.

Ensuite, le *Coin* est une classe simple avec un triggerAction() établie et une image déclaré de base. La *Potion* possède une durée, un index vers l’image de la potion, une couleur et un booléen qui défini si elle est prise ou non. La *Cloak* possède uniquement un booléen qui défini si elle est prise ou non. Le *Wall* est similaire au coin dans sa structure. Le *Chronometer* possède une puissance pour savoir à quel point le joueur remonte dans le temps, une durée pour définir combien de temps les gardes sont ralentis et un booléen pour savoir s’il est pris ou non. Dans le *Teleporter*, il y a seulement la position X/Y du second portail et l’index vers l’image du portail.

Pour finir, tous les personnages sont des classes qui ont héritées de la classe abstraite *Character*. Le *MainChar* possède uniquement un booléen qui permet de savoir s’il est en vie ou non et un tableau qui retient les positions sur lesquels il est passé. Pour les gardes, ils ont tous hérités de la classe abstraite *Guard* qui a elle-même héritée de la classe abstraite *Character*, elle possède un booléen pour savoir s’ils sont actifs ou non et une méthode abstraite *move(GameElement)*. Le *OrangeGuard* possède uniquement une redéfinition de la méthode *move()*. Le *BlueGuard* possède une direction vers laquelle se diriger, la taille de l’espace de jeu et un tableau pour retenir toutes les positions qu’il a exploré. Le *RedGuard* possède un *MainChar* afin de savoir constamment où il se situe. Le *PurpleGuard* détient un tableau de la position de toutes les tours et la direction vers laquelle il se dirige.

# **Limitation technique**

## ***Chronometer* fuite de mémoire :**

En effet, une fois tous les sabliers récupérés le *MainChar* continue d’enregistrer dans sa mémoire les positions sur lesquels il est allé. Ce qui entrainera une perte de mémoire. Il aurait fallu instaurer un système de réinitialisation mais par manque de temps cela n’a pas pu être terminé.

## ***RedGuard* reste bloquer :**

Effectivement, le garde rouge reste bloqué contre un mur quand le chemin le plus rapide pour aller vers le joueur se passe par un mur. Je n’ai pas trouvé de solution rapide et efficace de résoudre ce problème. Contrairement aux autres gardes qui s’il se bloque change cible, lui ne peut pas car sa seule cible est le joueur.

## ***BlueGuard* peut rester bloqué :**

Théoriquement ce serait possible mais cela n’a encore jamais été observé. En effet, si le garde bleu à exploré toute les case de l’espace de jeu mais qui lui en reste une seule et qu’il n’arrive pas à l’atteindre il ne pourra pas changer de direction et restera donc bloqué.

## **L’architecture logicielle peut être améliorée :**

Étant le premier projet avec une architecture *MVC* sur lequel nous avons travaillé, je pense m’en être bien sorti mais je pense tout de même que celle-ci peut être mieux implémentée. Je sais que si le projet était à refaire je ne ferais pas de la même manière.

## **Avec plus de temps ? :**

Avec plus de temps j’aurais aimé améliorer l’algorithme de déplacement des gardes, l’architecture logicielle et améliorer ma fonctionnalité supplémentaire. J’aurai également voulu implémenter un système de menu pour pouvoir reset, sauvegarder et quitter.

# **Conclusion**

Pour conclure, j’ai trouvé ce projet assez riche en termes d’apprentissage. Le principe du *MVC* est quelque chose d’intéressant à appliquer lors de la création d’une application. De plus, c’est la première que nous développons une application java avec une interface graphique. J’ai trouvé ça très passionnant et j’ai hâte d’en apprendre plus dans cette direction.

Malheureusement, j’ai également trouvé ce projet très éprouvant avec le peu de connaissance que nous possédions sur le sujet (*MVC, Java FX*). Il est vrai qu’énormément de travail en autodidacte nous a été demandé et nous n’avons pas été habitué à cela. Aussi, à titre personnel j’aurais aimé avoir des itérations au même titre que le projet Web. Cela permet d’avoir plusieurs deadlines mais également un retour périodique si le travail que l’on a fourni.

Enfin, j’ai appris de nombreuse chose avec ce projet et je sais d’ores et déjà ce que j’ai à appliqué dans les futurs projets pour que ceux-ci se passe encore mieux pour que tout soit produit dans les meilleurs délais.

**Sources**

**Sprites 2D :**

Les sprites sont des images 2D souvent utilisé dans des jeux vidéo, il s’agit souvent d’une image sous forme de pixel.

Site web Pinterest. Sprites de potions. Consulté en décembre 2023 à l’adresse : <https://www.pinterest.com/pin/554716879105805278/>

Site web BulbaGarden. *‘Bag Covert Cloak SV Sprite.png’*. Consulté en décembre 2023 à l’adresse : <https://bulbapedia.bulbagarden.net/wiki/File:Bag_Covert_Cloak_SV_Sprite.png>

Site web Shutterstock. Sprite de sablier. Consulter en décembre 2023 à l’adresse : <https://www.shutterstock.com/fr/image-vector/hourglass-pixel-art-icon-design-logo-2145811153>

Site web Pinterest. *‘Knight character design for topdown game’*. Consulté en décembre 2023 à l’adresse : <https://www.pinterest.com/pin/22518066874714643/>

Site web OpenGameArt.org. ‘2D Character sprite’. Consulté en novembre 2023 à l’adresse : <https://opengameart.org/content/2d-character-sprite-redshrike-mod>

Site web freepik.com. ‘Vecteur gratuit motif d'herbe verte transparente’. Consulté en novembre 2023 à l’adresse : [https://fr.freepik.com/vecteurs-libre/motif-herbe-verte-transparente\_13187581.htm#query=texture%20gazon%203d&position=0&from\_view=keyword&track=ais&uuid=4eaacd0b-68ec-4f8a-95c2-c0ab8353b5d6](https://fr.freepik.com/vecteurs-libre/motif-herbe-verte-transparente_13187581.htm%23query=texture%20gazon%203d&position=0&from_view=keyword&track=ais&uuid=4eaacd0b-68ec-4f8a-95c2-c0ab8353b5d6)

Site web Google Play. ‘Coin Clicker’ de Pond Studios. Consulté en novembre 2023 à l’adresse : <https://play.google.com/store/apps/details?id=com.joshuapond.coinclicker&pli=1>

Site web de Pinterest. ‘Stone Wall Texture 02 | 2D Stone | Unity Asset Store’. Consulté en novembre 2023 à l’adresse : <https://www.pinterest.com/pin/762445411905610517/>

Site web de Pinterest. ‘BuzzFeed’. Consulté en novembre 2023 à l’adresse : <https://www.pinterest.com/pin/84583299239782792/>

**Source du rapport :**

Site web Jeuxvidéo.com. Prince of Persia : Les sables du temps. Consulté en janvier 2024 à l’adresse : <https://www.jeuxvideo.com/jeux/playstation-2-ps2/00011869-prince-of-persia-les-sables-du-temps.htm>