



Besson Pierre-Victor
Desmazières Julien
Surget Kevin
Thaveau Joris

RAPPORT TECHNIQUE DU PROJET DÉVELOPPEMENT V2

7 juin 2017

Projet 70 : Développement d'un prototype de jeu vidéo en Java

A l'intention de l'équipe pédagogique du projet S2



IMT Atlantique
Bretagne-Pays de la Loire
École Mines-Télécom

Résumé du rapport

Ce rapport présente la programmation d'un prototype de jeu vidéo en Java*, qui permet de créer ses propres parties d'un jeu décrit dans ce rapport. Il s'agit d'un jeu de stratégie au tour par tour, similaire dans le principe à un jeu d'échec . 2 joueurs s'affrontent en contrôlant tour à tour des personnages se déplaçant sur une carte en 2 dimension composée de case carrées, dans le but de mettre hors jeu l'équipe adverse.

Nous avons d'abord convenu précisément des spécifications du jeu.

Puis, en plus d'une interface graphique programmée avec la bibliothèque Java *Swing**, des outils ont été créés afin de faciliter la tâche du développeur lors de la création d'instances de jeu. Enfin, un menu a été mis en place pour pouvoir naviguer entre les différentes fonctionnalités du prototype, comme sélectionner une partie ou quitter le logiciel. ¹

¹ A l'intention du lecteur.

Lors de la lecture de ce rapport, il y aura différents types d'écritures afin de différencier les termes, les voici :

- l'écriture du corps de texte ;
- *cette écriture qui référencera des classes, librairies Java ou tout autre terme qui fait référence à des variables utilisées dans le code.*

De plus, les * font référence au glossaire présent en fin de rapport, les [X] référencent la bibliographie présente en fin de rapport.

SOMMAIRE

Résumé du rapport.....	2
I.Introduction.....	4
II.Cahier des charges.....	5
III.Fonctionnement du jeu.....	6
III.1.Spécification de la carte.....	6
III.2.Spécification d'un personnage.....	7
III.3.Spécification d'une partie.....	8
IV.Système de jeu.....	8
IV.1.Gestion des Sauvegardes.....	8
IV.2. Gestion des déplacements.....	9
IV.3.Gestion de la ligne de vue.....	10
V.Développement du GUI.....	10
V.1.Fonctionnalités d'un GUI.....	10
V.2.Choix de la bibliothèque pour l'interface graphique.....	11
V.3.Utilisation de Swing.....	11
V.3.a.JFrame.....	11
V.3.b.JPanel.....	12
V.3.c.JButton.....	12
V.4.Structure du GUI.....	12
V.4.a.Case.....	12
V.4.b.Controller.....	13
V.4.c.Board.....	14
VI.Menu.....	15
VI.1.Principe du menu.....	15
VI.2.Les différentes classes du menu.....	16
VI.2.a.ComingSoon.....	16
VI.2.b.EditMenu.....	17
VI.2.c.LoadLevel.....	17
VI.2.d.MenuBackground.....	18
VI.2.e.MenuButton.....	18
VI.2.f.NewGame.....	18
VI.2.g.ScreenMenu.....	19
VI.2.h.Window.....	20
VI.2.i.Menu.....	20
VII.Logger.....	20
VIII.Tests et résultats.....	21
IX.Conclusion.....	21
X.Bibliographie.....	22
XI.Glossaire.....	23
XII.Annexes.....	24
XII.1.Planning initial.....	24
XII.2.Analyse du planning et comparaison du prévisionnel avec le réel.....	25

Besson Pierre-Victor : Rédacteur partie 1, 5, 8, 11 et résumé, relecteur parties 3, 4, 6 et 7

Desmazières Julien : Rédacteur parties 1, 3, 4, 8 et résumé, relecteur parties 2, 5 et 9

Surget Kevin : Relecteur parties 6 et 7.

Thaveau Joris : Rédacteur parties 6, 7, 9, relecteur de toutes les parties.

I. Introduction

Le marché du jeu vidéo est maintenant d'une taille comparable à d'autres marchés culturels, comme la musique ou la lecture, et peut déboucher sur de nombreuses carrières, que ce soit dans une entreprise déjà établie ou en indépendant. C'est dans ce cadre que nous avons décidé de développer pour ce projet un prototype de jeu vidéo. Contrairement à la majorité des projets de ce semestre, ce projet ne répond pas à un besoin particulier, mais est une initiation dans le monde de l'industrie vidéoludique, qui est pour nous un potentiel choix de carrière.

Cependant, plutôt qu'un véritable jeu c'est un moteur de jeu que nous avons construit : un code qui permet de créer une partie facilement personnalisable avec des règles simples prédéfinies, et qui pourra être peaufiné par la suite. Ainsi les compétences de « Game Design » (tout ce qui touche à l'expérience du joueur plutôt qu'à la machine derrière) requises pour la création d'un jeu vidéo sont mises de côté pour se concentrer véritablement sur la mise en place d'un code fonctionnel.

Ce travail a été effectué en Java* , afin d'éviter d'avoir à apprendre un tout nouveau langage tout en approfondissant nos connaissances dans ce langage. De plus, Java* est un des langages de programmation les plus utilisés, et ses limites techniques (en terme de vitesse par exemple) ne nous préoccupent pas ici étant donné la relative simplicité de notre projet.

Dans un premier temps, les besoins de l'utilisateur ont été spécifiés. Puis, toutes les modalités et contraintes du jeu ont dû être décrites de manière exhaustive. Une fois ces étapes accomplies, la programmation s'est faite selon deux axes principaux. D'une part toutes les classes et leurs interactions décrites précédemment ont été réalisées. D'autre part une interface Homme-Machine a été développée afin de détecter les actions du joueur. Enfin la liaison de ces deux parties a été effectuée, donnant suite à la réalisation de différents tests vérifiant l'intégrité du code. Ces tests nous ont permis d'améliorer et de corriger certaines parties du code que nous n'avions pas remarqué en amont. Mais ils nous permettent aussi d'assurer la durabilité du logiciel.

Tout d'abord, nous décriront toutes les contraintes liées au cahier des charges de ce jeu. Puis nous détaillerons le fonctionnement du jeu, la manière dont les classes interagissent entre-elles, ainsi que toutes les modalités et spécifications que nous avons choisies d'implémenter. Ensuite nous expliquerons le fonctionnement de certains principes spécifiques à notre jeu, donnant alors une description exhaustive du logiciel. Dans un second temps, nous entrerons dans les choix de programmation de l'interface graphique. Pour cela, nous allons tout d'abord expliciter les fonctionnalités de cette interface. Puis nous expliquerons le choix de la bibliothèque *Swing** pour son développement. Nous détaillerons ensuite la structure de cette interface. Nous finirons alors par la description du fonctionnement du menu, ainsi que du *Logger* utilisé. Ainsi, l'ensemble des éléments constituant notre prototype sera abordé.

II. Cahier des charges

Menu principal

Le logiciel devra comporter un menu principal, affiché à son lancement. Ce menu devra permettre de naviguer entre :

- une sélection de niveaux parmi ceux disponibles ;
- un accès à une sauvegarde ;
- un menu d'options. (optionnel)

La navigation s'effectuera à l'aide de commandes à la souris (clavier optionnel).

Sauvegarde

Le logiciel devra comporter une fonction de sauvegarde, accessible à chaque tour du joueur. Elle permettra d'enregistrer l'état d'une partie en cours sous la forme de fichiers texte et sera conservée même après fermeture du logiciel.

Ces sauvegardes pourront également être chargées afin de reprendre la partie en cours.

Chargement

Le logiciel devra comporter différents niveaux de jeu, qui différeront de par les personnages et ennemis présents, ainsi que par la carte. L'utilisateur pourra choisir de jouer à l'un ou l'autre de ces niveaux via le menu. Ces cartes seront composées d'un quadrillage de cases carrées (cf Cases).

Le logiciel pourra aussi charger les précédentes sauvegardes.

PvP* et IA*

Les personnages non joueurs se devront d'être contrôlés par une intelligence artificielle basique, capable de choisir aléatoirement une action parmi une liste d'actions prédéfinies.

Il y a aussi possibilité de contrôler les autres personnages par un autre joueur.

Système de combat

Cette catégorie regroupe toutes les fonctions nécessaires au bon déroulement d'une partie du jeu. Elle comporte :

- Personnages :

Les personnages se diviseront en 2 catégories : des personnages contrôlés par le joueur, et des personnages ennemis. Les 2 catégories seront, en pratique, codées par la même classe. Les personnages disposent de statistiques et compétences propres, ainsi que d'une apparence unique. Ils seront définis au moyen d'un fichier texte.

- Compétences :

Elles seront appelées par les personnages lors de leurs tours de jeu. Chaque compétence sera définie par un fichier texte et disposera de ses caractéristiques propres, comme sa portée ou ses dégâts.

- Tours de jeu :

Les combats s'effectueront au tour par tour, chaque personnage pouvant effectuer des actions selon un ordre prédéfini. Durant un tour de jeu, un personnage pourra effectuer des compétences et/ou se déplacer.

Ces tours seront interrompus à la demande du joueur dans le cas de ses personnages sauf exception, et à la fin des actions de l'IA* dans le cas des personnages non joueur.

- Cases :

Les personnages évolueront sur des cases, définies par la carte. Ces cases pourront être occupées par un personnage, un obstacle, ou des pièges, qui auront des effets sur les personnages se situant dessus.

Ces cases seront également l'unité de base déterminant la portée des attaques et des déplacements des personnages.

- Ligne de vue :

Certaines compétences nécessiteront d'avoir une ligne de vue entre le personnage souhaitant effectuer cette compétence et sa cible. Cette ligne de vue sera déterminée par la présence ou non d'obstacles entre les 2.

- Déplacement.

Les personnages seront capables de se déplacer d'une case à une autre, sur demande du joueur ou choix de l'IA*. Ces déplacements seront soit effectués case par case, soit calculés par algorithmes de plus court chemin dans le cas d'un trajet plus long.

GUI*

Le logiciel disposera d'une interface utilisateur graphique, permettant l'affichage d'une fenêtre où s'afficheront des informations concernant la partie. Cette interface sera également capable de recevoir des commandes du joueur via le clavier ou la souris et de les transmettre au logiciel.

III. Fonctionnement du jeu

La création d'une partie se fait en trois étapes. Il y a d'un cotés la création de personnage jouable, d'un autre la création d'une carte de jeu, et enfin la création d'une partie qui combine une carte, plusieurs personnages disposés sur cette carte et un lien vers le contrôleur.

III.1. Spécification de la carte

La carte est un quadrillage rectangulaire de largeur et de hauteur compris entre 2 et 20 cases. Chaque case (*Square*) peut être composée d'un arrière plan (*Background*), d'un élément de décor (*Element*), d'un piège (*Trap*), d'une hauteur (*Height*), et d'un personnage (*Character*) (cf *Illustration 1*).

L'arrière plan correspond au terrain de base, les différentes valeurs qu'il peut prendre sont « eau », « terre », « sable » ou « vide ». Elles définissent principalement la couleur de l'arrière plan.

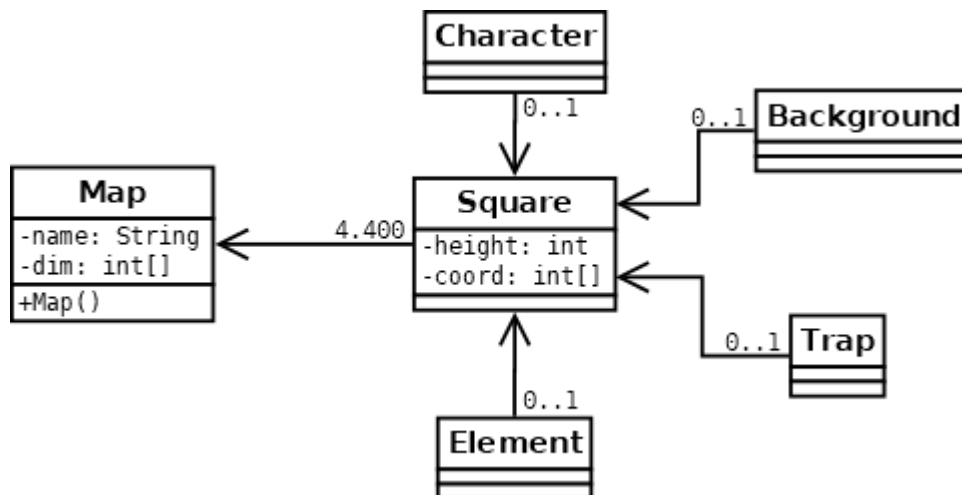


Illustration 1: Diagramme de classe simplifié des composants d'une carte (*Map*)

Une case peut aussi avoir un unique élément de décor. Chaque élément est défini grâce à une classe dédiée qui hérite de la classe *Element*. Ces éléments peuvent ainsi empêcher le déplacement des personnages, ou la ligne de vue.

Un personnage peut aussi à travers une attaque spéciale, poser un piège sur la carte de jeu. Ce piège n'affecte en rien la ligne de vue ou le déplacement des personnages. Mais lorsqu'un personnage passe sur cette case, le piège déclenche son effet.

Il est possible d'attribuer une hauteur à une case, ce qui peut modifier les déplacements et la ligne de vue avec les autres cases.

Enfin sur chaque case, il peut y avoir un personnage. Ces personnages peuvent interagir avec la carte en se déplaçant dessus, en interagissant avec les éléments du décor, ou en posant ou déclenchant des pièges.

III.2. Spécification d'un personnage

Le personnage est l'unité contrôlable du jeu. Chaque personnage possède un nom, un identifiant utile lors de la sauvegarde ou du chargement d'une partie (cf 2.1), un état, des statistiques qui lui sont propres, ainsi que des attaques.

Les statistiques sont au nombre de 5, les points de vie, les points de mouvement qui permettent au personnage de se déplacer, les points d'action qui permettent au personnage d'utiliser ses attaques, la défense et la défense magique qui permettent de diminuer les dégâts infligés par une attaque subie, et la vitesse qui dicte la fréquence à laquelle le personnage pourra jouer dans une partie.

Il existe quatre types d'attaques : les attaques mono-cibles qui infligent des dégâts, les attaques mono-cibles de statut, les pièges et les attaques qui affectent tous les personnages. Les attaques mono-cibles permettent d'infliger des dégâts directement à une cible unique si celui-ci se trouve dans la ligne de vue de l'attaquant (cf IV.4).

Les pièges sont eux placés sur une case ne possédant pas déjà un autre piège ou un personnage. Le déclenchement du piège se fait à distance par la personne qui l'a posé ou par un personnage qui passe sur la case en question.

Enfin l'état du personnage modifie ses caractéristiques. Il peut être "mort" dans ce cas, il sera retiré du terrain et ne pourra plus effectuer aucune action ; « inerte » la récupération de points de mouvement est alors amoindrie ; « paralysé » dans ce cas la récupération de points d'action est amoindrie ; « brûlé » le personnage perdra des points de vie à la fin de chacun de ses tours.

III.3. Spécification d'une partie

Une partie est composée d'une carte et d'un nombre de personnages compris entre 2 et 20 (si la taille de la carte le permet) répartis dans 2 équipes de manière à ce qu'il y ait au moins un personnage dans chaque équipe, avec une liste donnant l'ordre d'apparition des personnages.

La partie se déroule au tour par tour, et consiste en la confrontation de deux équipes contrôlées par des joueurs humains ou par des Intelligences Artificielles. A chaque tour, une interface est recalculée. Elle permet d'envoyer au contrôleur (cf?) les données concernant le personnage jouable si ce personnage appartient à une équipe contrôlée par un joueur humain. Elle exécute sinon l'intelligence artificielle associée à l'équipe du personnage.

Un personnage finit son tour en utilisant l'action *EndTurn()*. Cela déclenche le calcul de l'interface du personnage suivant. Mais aussi, cela repositionne le personnage dans la liste qui donne l'ordre des personnages, en fonction de sa vitesse et de celle des autres personnages.

La fin d'une partie est déclarée à partir du moment où tous les personnages d'une des deux équipes sont morts.

De plus une partie peut être sauvegardée dans le dossier prévu à cette effet « *gameSave* », en suivant les restrictions de la partie.

IV. Système de jeu

IV.1. Gestion des Sauvegardes

Lors d'une partie, un joueur peut sauvegarder de manière persistante l'état actuel de sa partie à n'importe quel moment. Pour cela, des fichiers texte sont utilisés dans lesquels toutes les informations nécessaires sont enregistrées sous forme de chaîne de caractères respectant des contraintes spécifiques. [6][7][8][12]

En effet lorsqu'un joueur décide de sauvegarder, un dossier est créé portant le nom de la partie en cours, dans lequel il y a un fichier texte qui contient les données liées à l'affichage de la partie dans les menus (cf VI.2.c), un fichier texte qui contient les spécifications de la carte, ainsi que des fichiers texte contenant chacun les spécifications d'un des personnages de la partie. Le fichier décrivant la carte est construit de la manière suivante :

- nom de la carte sous la forme d'une chaîne de caractères
- taille de la carte sous la forme largeur x longueur

La matrice représentant les cases de la carte dont chaque colonne est délimitée par le caractère « underscore », et dont chaque case est définie par une chaîne de caractères qui est de la forme suivante : entier b ; entier h ; entier e ; entier t ; entier p.

Les spécifications de ces entiers sont :

- l'entier devant b correspond à l'*id** de l'objet *Background* associé à la case ;
- l'entier devant h correspond à la hauteur de la case ;
- l'entier devant e correspond à l'*id** de l'élément associé à la case ;
- l'entier devant t correspond à l'*id** du piège associé ;
- l'entier devant p correspond à l'*id** du personnage associé à la case.

Les pièges et les éléments sont sauvegardés de la même manière mais dans deux fichiers textes différents. Chaque ligne du fichier piège (respectivement élément) correspond à une instance d'un piège (respectivement élément) et contient l'*id* ainsi que le nom de la classe. Ainsi en utilisant la réflexivité de Java* [9], on pourra directement charger ces objets, en utilisant la chaîne de caractères sauvegardée.

Le fichier décrivant un personnage est construit de la manière suivante :

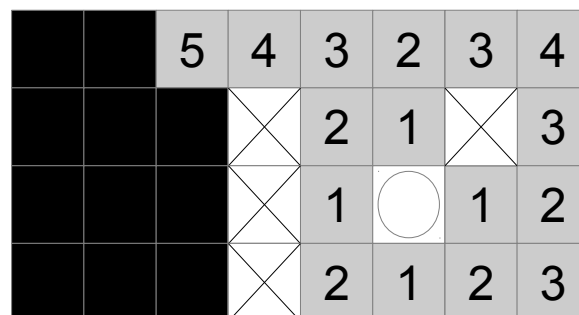
- le nom du personnage
- son identifiant ;
- chacune des statistiques (une par ligne) ;
- l'équipe du personnage ;
- son nombre de tours ;
- la liste des attaques du personnage.

Le chargement d'une partie se fait grâce à la lecture de tous les fichiers du dossier correspondant. Cependant pour les attaques des personnages on utilise la réflexivité. En effet un personnage possède une liste d'objets héritant de la classe *Ability*. De ce fait, on utilise la réflexivité pour obtenir une instance d'une classe à partir d'une chaîne de caractères correspondant au nom de cette classe.

IV.2. Gestion des déplacements

Lorsqu'on sélectionne un personnage, celui-ci a la possibilité de se déplacer sur la carte. Cependant, ce déplacement est soumis à plusieurs contraintes. Le personnage ne se déplace qu'à travers des cases adjacentes horizontalement ou verticalement[11]. Il ne peut se déplacer sur une case où se situe déjà un autre personnage, ou un élément de décor ne permettant pas qu'on se positionne dessus.

De plus le déplacement est limité par le nombre de points de mouvement que le joueur possède. La distance maximale de déplacement est affectée par l'état du personnage et par la case sur laquelle il se trouve (*cf Illustration 2*).



Légende :





- obstacles ; 
- case accessible en 1 mouvement ; 
- case inaccessible ; 
- personnage. 

Illustration 2: Schéma illustrant les déplacements d'un personnage possédant 5 points de mouvement

IV.3. Gestion de la ligne de vue

Lorsqu'un personnage souhaite attaquer, il est contraint par la ligne de vue. Certains éléments de décor peuvent bloquer la ligne de vue du personnage, ce qui l'empêche d'atteindre une cible située derrière. La ligne de vue est construite de la manière suivante : (cf *Illustration 3*)

On échantillonne le segment passant par les coordonnées des deux cases. Si aucun carré, situé entre les deux précédentes cases, n'a de points qui vérifient l'équation de la droite, alors la ligne de vue est ouverte. Dans le cas contraire, la ligne de vue est considérée comme bloquée. Comme les déplacements, les éléments du décor peuvent gêner la ligne de vue uniquement suivant une certaine direction horizontale, verticale ou diagonale.

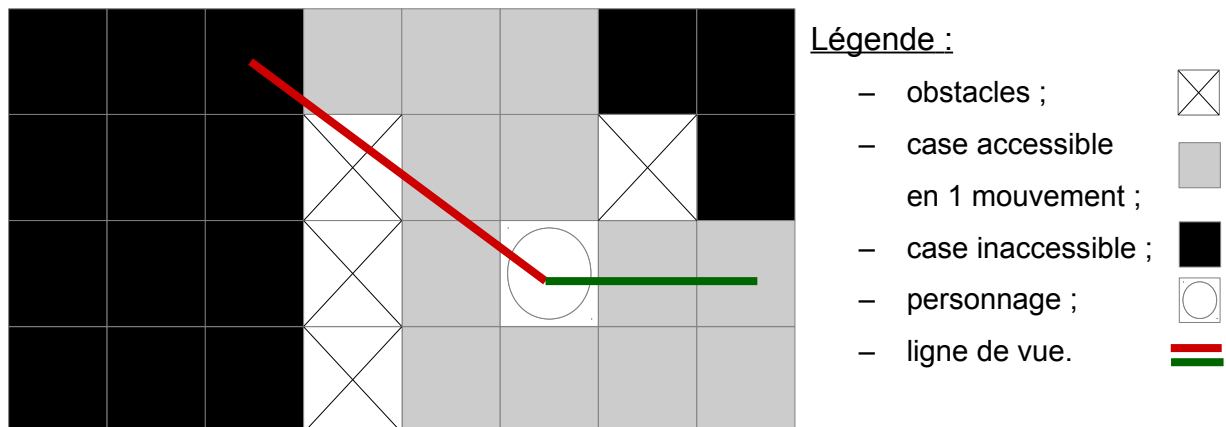


Illustration 3: Schéma illustrant la ligne de vue d'une attaque

Ainsi tous ces éléments constituent les fonctionnalités du jeu et aussi ses limites et contraintes. Cependant un utilisateur ne peut pas interagir directement avec ces classes, il a besoin d'une interface. C'est au GUI* (Game User Interface) que revient cette tâche comme développé dans la partie ci-après.

V. Développement du GUI

V.1. Fonctionnalités d'un GUI

Le GUI* permet à l'utilisateur de l'application d'interagir avec le jeu via une interface graphique. Conformément au cahier des charges, le GUI* affiche un plateau de jeu composé de cases cliquables.

Ces cases ont des apparences modifiables selon les besoins du jeu.

Il assure également la cohérence entre cette interface graphique et un modèle du jeu non accessible à l'utilisateur.

Ce GUI est construit selon l'architecture MVC [10] (Model View Controller) (cf *Illustration 4*). Dans cette architecture, une vue graphique (*View*) reçoit les commandes de l'utilisateur et les transmet au contrôleur (*Controller*), qui applique ensuite leur effet sur le modèle (*Model*). Les modifications sont alors retransmises en amont à la vue.

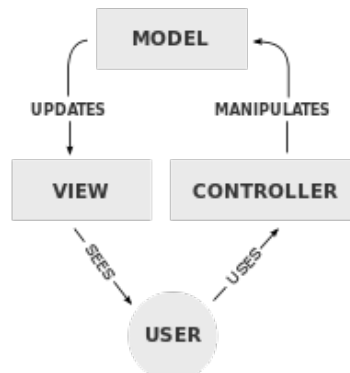


Illustration 4: Représentation schématique du modèle MVC et des interactions entre ses composants.

Le modèle est assuré par les classes vues précédemment, en particulier *partyInterface* (cf). La vue et le contrôleur seront traités dans cette partie. La vue requiert cependant l'utilisation d'une bibliothèque graphique.

V.2. Choix de la bibliothèque pour l'interface graphique

Deux bibliothèques sont proposés par Java* afin d'afficher une interface graphique :

- AWT*, la plus ancienne, mais aussi la plus lourde, celle-ci sera néanmoins utilisée pour certains éléments non présents dans l'autre bibliothèque.
- Swing*, cette bibliothèque est plus légère que son homologue, et permet d'effectuer la plupart des actions que nous souhaiterons faire.

Afin d'éviter tout conflit entre les deux bibliothèques, puisqu'elles agissent sur les mêmes composants, nous privilégions l'utilisation de Swing pour l'interface graphique.

V.3. Utilisation de Swing

Elle est importée par la ligne de code `import javax.swing.*;`

Elle est choisie ici pour réaliser le GUI* de ce projet, et aura nécessité beaucoup de recherches et de pratique avant de la maîtriser suffisamment.

Les 3 classes de Swing* les plus utilisées dans ce projet sont *JPanel**, *JButton** et *JFrame**. Elles sont détaillées dans leur partie correspondante. De plus, les *Layout** seront aussi détaillés.

V.3.a. JFrame

*JFrame** [3] permet de créer une fenêtre graphique. Cette fenêtre peut ensuite être remplie par d'autres composants *Swing**.

Après la création d'un objet *JFrame**, il est possible d'interagir avec lui de plusieurs manières. Il est par exemple possible de :

- choisir le titre de la fenêtre ;
- choisir la taille de la fenêtre ;
- choisir l'opération à exécuter en cas de fermeture de la fenêtre ;
- afficher ou non la fenêtre ;
- interdire ou autoriser le redimensionnement de la fenêtre par l'utilisateur ;
- ajouter des éléments *Swing** à la fenêtre.

V.3.b. JPanel

Un objet *JPanel**[4] est une sorte de toile de peinture qui peut être remplie puis ajoutée à une *JFrame**.

Tout comme *JFrame**, il est possible d'en modifier de nombreux paramètres (taille, visibilité, couleur de fond...). Il est également possible d'y ajouter d'autres éléments *Swing**, et surtout, de les ajouter selon un motif via des *Layout** [5], disponibles dans la bibliothèque *java.awt*.

Par exemple, la commande *setLayout(new GridLayout(length, height))* ; permet de subdiviser le *JPanel** en une grille de taille *length* x *height*.

V.3.c. JButton

Un objet *JButton** [2] est un bouton cliquable. Il est possible d'en définir la taille, l'icône, et surtout l'action à effectuer si il est cliqué.

Cependant, pour pouvoir détecter si un bouton est cliqué, ou de manière générale détecter une action de l'utilisateur sur un élément *Swing**, il est nécessaire d'y ajouter un *ActionListener**, à l'aide de l'interface *ActionListener** disponible dans la bibliothèque *java.awt.event*. Ceci permet de générer un objet *ActionEvent** à chaque action de l'utilisateur, qui pourra ensuite être utilisé pour réagir de manière appropriée à cette action.

Il est aussi possible d'associer directement une *ActionCommand** à un bouton via la commande *setActionCommand(String nomAction)*. Cela permet d'associer un nom d'action à effectuer à un bouton et de récupérer ce nom quand le bouton est appuyé pour choisir l'action à effectuer.

Ceci est utile dans le cas de nombreux boutons qui doivent effectuer des actions similaires. En effet au lieu d'écrire une action à effectuer pour chaque bouton, il suffira de définir pour chaque bouton une *ActionCommand**, que l'on pourra récupérer ensuite et donner en argument d'une fonction générale à appliquer lors de l'appui d'un bouton.

V.4. Structure du GUI

Le code du GUI* est décomposé en 3 classes, qui seront détaillées dans leur propre partie :

- La classe *Case*, qui représente une case de l'interface graphique. Elle est cliquable, et de ce fait hérite de la classe *JButton* de *Swing*.
- La classe *Controller*, qui assure les interactions entre l'utilisateur et les données du jeu.
- La classe *Board*, qui constitue l'interface graphique à proprement parler, et qui affiche le plateau de jeu. Elle communique également avec *Controller* afin de transmettre les commandes rentrées par l'utilisateur sur l'interface graphique.

V.4.a. Case

Case est une classe héritant de *JButton**. Elle dispose en attributs :

- de coordonnées (i,j), i et j étant des entiers *int* ;
- d'une icône *ImageIcon* ;

- d'un *Square*.

Son apparence sera constituée d'une couleur de fond et éventuellement d'une icône.

L'attribut *Square*, donné lors de la création de *Case* par le constructeur, permet de déterminer l'apparence de la case. En effet, la couleur de fond sera déterminée par l'attribut *Background* du *Square*, et si un personnage de la classe *Character* est présent dans *Square*, l'icône de *Case* sera l'icône de *Character*, récupérée dans les fichiers du jeu par *getSpriteLoc*.

À chaque *Case* est associée une *ActionCommand*, qui contient ses coordonnées.

V.4.b. Controller

Controller, conformément au modèle MVC, gère les interactions entre l'utilisateur et le modèle de jeu. Elle n'hérite d'aucune classe existante. Elle dispose en attribut :

- d'une *Case[][] ListCase* ;
- d'un *Character currentCharacter* ;
- d'une *PartyInterface partyInterface*.

Les fonctions de *Controller* seront appelées en temps voulu par l'interface graphique et affecteront *PartyInterface*, qui contient toutes les informations relatives au déroulement de la partie. Les 2 fonctions les plus importantes de cette classe sont :

- *public int askCommand()*. Cette fonction déclenche un affichage sur la console des actions disponibles à l'utilisateur à un moment du jeu donné et pour un personnage donné. A noter qu'en pratique *askCommand* ne sera appelée que si le personnage pour qui on demande les actions correspond à l'attribut *currentCharacter*. A chacune de ces commandes est associée un entier. (cf *Illustration 5*). L'utilisateur rentre l'entier correspondant à la commande, et la fonction renvoie cet entier.

```
name : pers1, id : 1, pv : 1000/1000, pm : 3/3, pa : 6/6, def : 20/20, defM : 30/30, speed : 30, status :HEALTHY
C'est à pers1 de commencer !
Commandes possibles :

0 - Fin de tour
1 - Déplacement
2 - Punch

3 - BasicArrow

Entrez la commande désirée
```

Illustration 5: Affichage de la console lors de l'appel de *askCommand()* sur le personnage *pers1*

- *public void applyCommand(Square squareFin, int actionID)*. Cette fonction est la suite logique de la fonction précédente, et prend en argument un entier, qui correspond à une action, ainsi qu'un *Square* qui correspond à la cible de cette commande. Cette fonction applique alors simplement la commande sur la cible.

Controller dispose également de fonctions visant à rendre l'affichage plus interactif, comme par exemple *showMovementRange()* qui permet de changer le contour des cases accessibles par un personnage lorsque celui ci souhaite se déplacer.

V.4.c. Board

Board représente l'interface graphique elle-même. Elle hérite de *JPanel** et implémente *ActionListener**. Elle dispose en attributs :

- d'une hauteur *height* et d'une longueur *length*, toutes 2 entières ;
- d'une *MatriceBoard matriceBoard* ;
- d'une *Case[] ListeCase* ;
- d'une instance *JFrame frame* ;
- d'un *Controller controller* ;
- d'une *partyInterface partyInterface* ;
- d'un *Character charRequesting* et d'un entier *actionRequested*, qui aideront à l'exécution des actions demandées par l'utilisateur.

La construction de la fenêtre se fait en plusieurs étapes.

L'objet *Board* est d'abord créé en prenant en argument un objet *partyInterface*, la fenêtre *Window* et un nom pour la sauvegarde du fichier.

L'objet *JFrame frame* est créé, sa taille son nom et sa position sont choisis et il y est ajouté un *ActionListener**.

Enfin l'objet *Board* se voit attribuer un *Layout** en forme de grille, et est ensuite rempli d'objets *Case*. Les objets *Case* ajoutés ont été construits en donnant en argument du constructeur *Case* les *Square* de *MatriceBoard.matrice*, qui est une *Square[][]*.

Les composants sont alors rendus visibles. (cf *Illustration 6*)

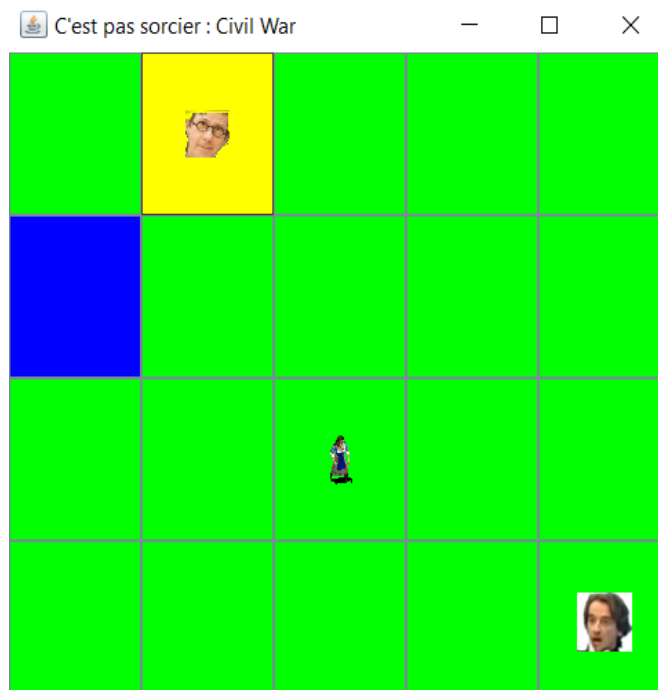


Illustration 6: Affichage du GUI. Le plateau de jeu est composé de cases d'herbes (vert) d'une case d'eau (bleu) et d'une case de sable (jaune), sur lesquelles vont évoluer 3 personnages. Le contour différent autour du personnage du haut signifie que c'est à son tour de jouer.

Board récupère les commandes de l'utilisateur via les boutons de *ListCase[][]*, et les transmet à *Controller*. Lors d'un clic de l'utilisateur sur une des cases, les coordonnées de la case sont récupérées grâce au *ActionCommand** de l'objet *Case* sur lequel il a cliqué. Si la case cliquée contient le personnage dont c'est le tour de jeu (vérifié à l'aide de *partyInterface*) , la fonction *askCommand()* de *Controller* est alors appelée. Cette fonction implémente des *hashTable* pour associer un numéro à une commande. [13]

Le contrôleur récupère l'action demandée par l'utilisateur. On peut distinguer 2 types d'actions :

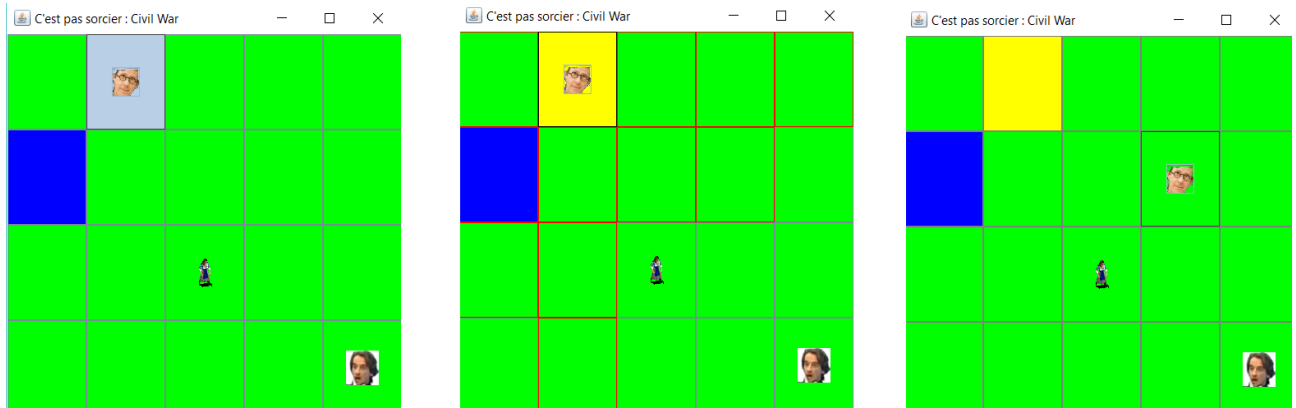


Illustration 7: Affichage du déplacement d'un personnage.

De gauche à droite : sélection du personnage, affichage des cases atteignables par le personnage et nouvelle position du personnage.

- Les actions en une étape, comme fin de tour. Elles sont alors immédiatement effectuées.
- Les actions en 2 temps, comme une attaque ciblée ou un déplacement. L'action est alors stockée dans *actionRequested*, et les cases sur lesquelles l'action peut être effectuée (portée de l'action) sont entourées de rouge. Un autre clic de l'utilisateur sur l'une de ces cases effectuera l'action sur la case désirée. (cf *Illustration 7*)

A la fin de chaque action, l'affichage du *Board* est mis à jour à l'aide de la fonction *update()*.

Une fois qu'un personnage a effectué l'ensemble des actions qu'il souhaitait ou pouvait effectuer, il lui est alors nécessaire de rentrer une commande Fin de Tour. La main passe alors automatiquement au personnage suivant, qui pourra agir de manière similaire.

Dans la version actuelle, la fermeture de *Board* (via la croix) procède également à une sauvegarde de l'état du plateau de jeu.

Ainsi l'utilisateur peut grâce au GUI interagir avec le jeu, déplacer ses personnages, attaquer. Cependant, lors de l'exécution du programme, l'utilisateur doit accéder à un menu qui le dirigera vers ce GUI. De plus, ce menu doit permettre à l'utilisateur de lancer de nouvelles parties, de charger d'anciennes, ou même de quitter l'application.

VI. Menu

VI.1. Principe du menu

Le menu est accessible au joueur dès le lancement du programme. A partir de celui-ci, il peut :

- commencer une nouvelle partie ;
- choisir une partie sauvegardée ;
- créer un niveau à partir de l'éditeur de niveau ;
- changer les options (taille de la fenêtre, plein écran) ;
- quitter.

Le bouton « New game » amène un autre menu qui affiche la liste des différentes cartes disponibles, avec différentes informations.

Le bouton « Load a level » donne à l'utilisateur le choix de la partie sauvegardée à charger sur un autre menu, similaire à celui pour le début d'une nouvelle partie, avec différentes informations, en montrant la liste des différentes parties sauvegardées.

Le bouton « Create new level » qui renvoie le menu pour l'éditeur de niveau demande à l'utilisateur une taille de carte, et un nom pour celle-ci.

Le bouton « Options » amène pour l'instant vers un écran temporaire, montrant que la fonctionnalité n'est pas encore disponible.

Le dernier bouton « Quit » lui permet de quitter le programme.

VI.2. Les différentes classes du menu

VI.2.a. ComingSoon

La classe *ComingSoon* hérite de *JFrame* et affiche une fenêtre annexe par dessus l'autre (cf *Illustration 8*). Celle-ci correspond à un bouton affichant « Coming Soon » pour signaler à l'utilisateur que le contenu n'est pas encore disponible, notamment dans le cas du menu des options.

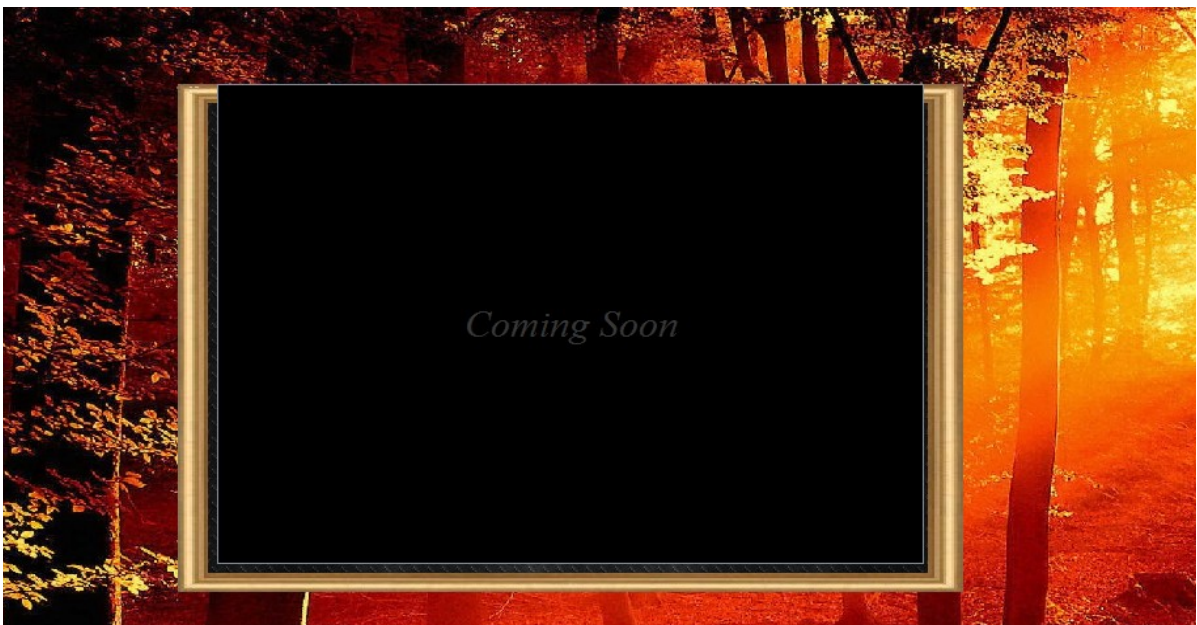


Illustration 8: Bouton temporaire pour prévenir des fonctionnalités futures

VI.2.b. EditMenu

La classe *EditMenu* qui hérite de *JPanel* est celle qui représente l'écran afin de spécifier les données de base pour créer un nouveau niveau (cf *Illustration 9*). En effet, celle-ci demande :

- la hauteur de la carte ;
- la longueur de celle-ci ;
- un nom pour la distinguer.

Après ses informations renseignées, elle les récupère pour les transmettre à l'éditeur.



Illustration 9: Écran pour définir la taille de la carte avec son nom

VI.2.c. LoadLevel

La classe *LoadLevel* qui hérite de *JPanel* est celle qui affiche l'écran de sélection d'une partie sauvegardée (cf *Illustration 10*). L'écran est alors composé :

- d'un *Background* qui constitue le fond de l'écran ;
- d'un menu central qui permet :
 - la sélection d'une sauvegarde dans les fichiers du jeu ;
 - de voir les informations liées à la sauvegarde comme le nom de la carte, sa taille, le nombre de tours, et la date de création de celle-ci, qu'elle récupère dans les fichiers de la sauvegarde ;
 - le choix de mettre les joueurs en tant qu'IA ou non ;
 - deux boutons « Back » afin de revenir sur l'écran précédent et « Load » afin de charger la partie via le GUI.

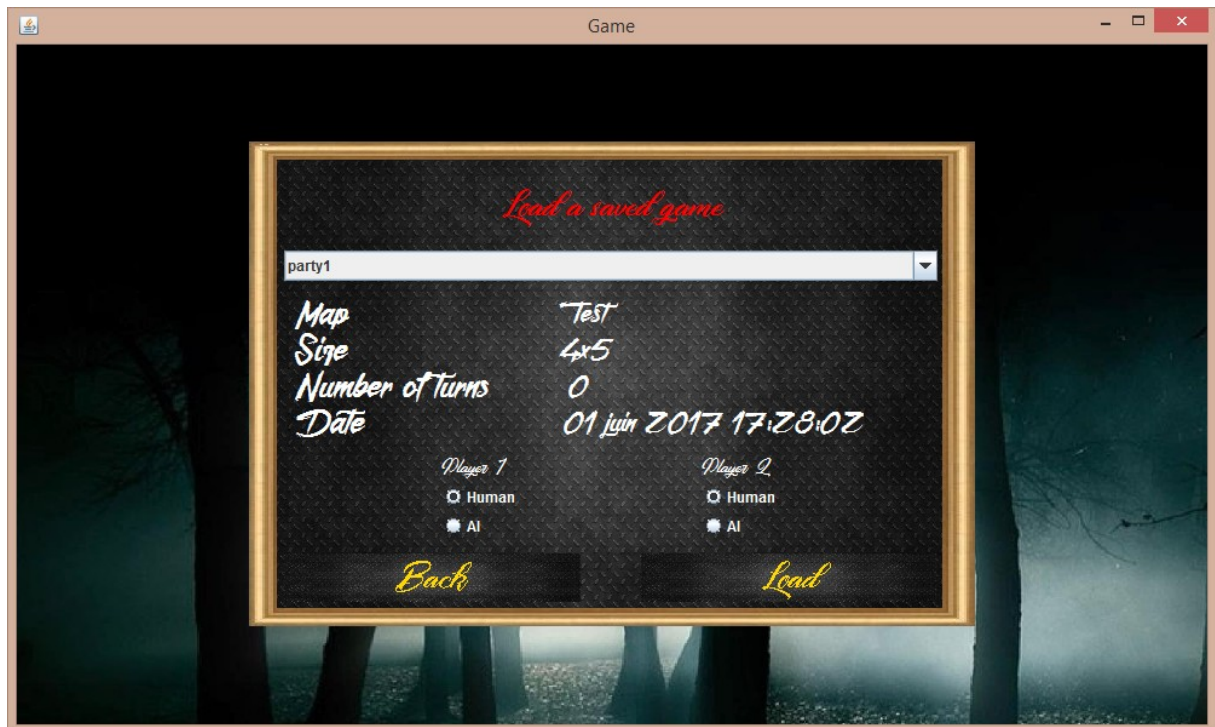


Illustration 10: Écran de sélection d'une partie sauvegardée.

VI.2.d. MenuBackground

La classe *Background* hérite de *JPanel*, afin d'afficher le fond d'écran du menu, avec une image choisie lors de la création. Celle-ci affiche un cadre doré au centre, où s'y trouve les différents menus de sélection avec les différents boutons.

La classe possède une méthode qui permet de redimensionner l'image de fond d'écran pour qu'elle soit affichée de la manière la plus optimale, tout en conservant les dimensions initiales de l'image.

De plus, l'affichage du fond d'écran est actualisé à chaque modification de la fenêtre. Ainsi le cadre reste toujours au centre avec les boutons en son sein.

VI.2.e. MenuButton

La classe *MenuButton* hérite de *JButton* et correspond aux boutons du menu. Ceux-ci possèdent une police qui leur est propre, importée à partir d'un fichier annexe. Ils possèdent aussi un fond et une couleur, choisie à l'initialisation du bouton, qui changent lorsque l'utilisateur passe sa souris dessus. Ils permettent de sélectionner les autres menus.

VI.2.f. NewGame

La classe *NewGame* qui hérite de *JPanel* est celle qui affiche l'écran de sélection d'une nouvelle partie à partir d'une carte déjà existante (cf *Illustration 11*). L'écran est alors composé :

- d'un *Background* qui constitue le fond de l'écran ;
- d'un menu central qui permet :
 - la sélection d'une sauvegarde dans les fichiers du jeu ;

- de voir les informations liées à la sauvegarde comme le nom de la carte, sa taille, le nombre de tours, et la date de création de celle-ci, qu'elle récupère dans les fichiers de la sauvegarde ;
- le choix de mettre les joueurs en tant qu'IA ou non
- deux boutons « Back » afin de revenir sur l'écran précédent et « Load » afin de charger la partie via le GUI.

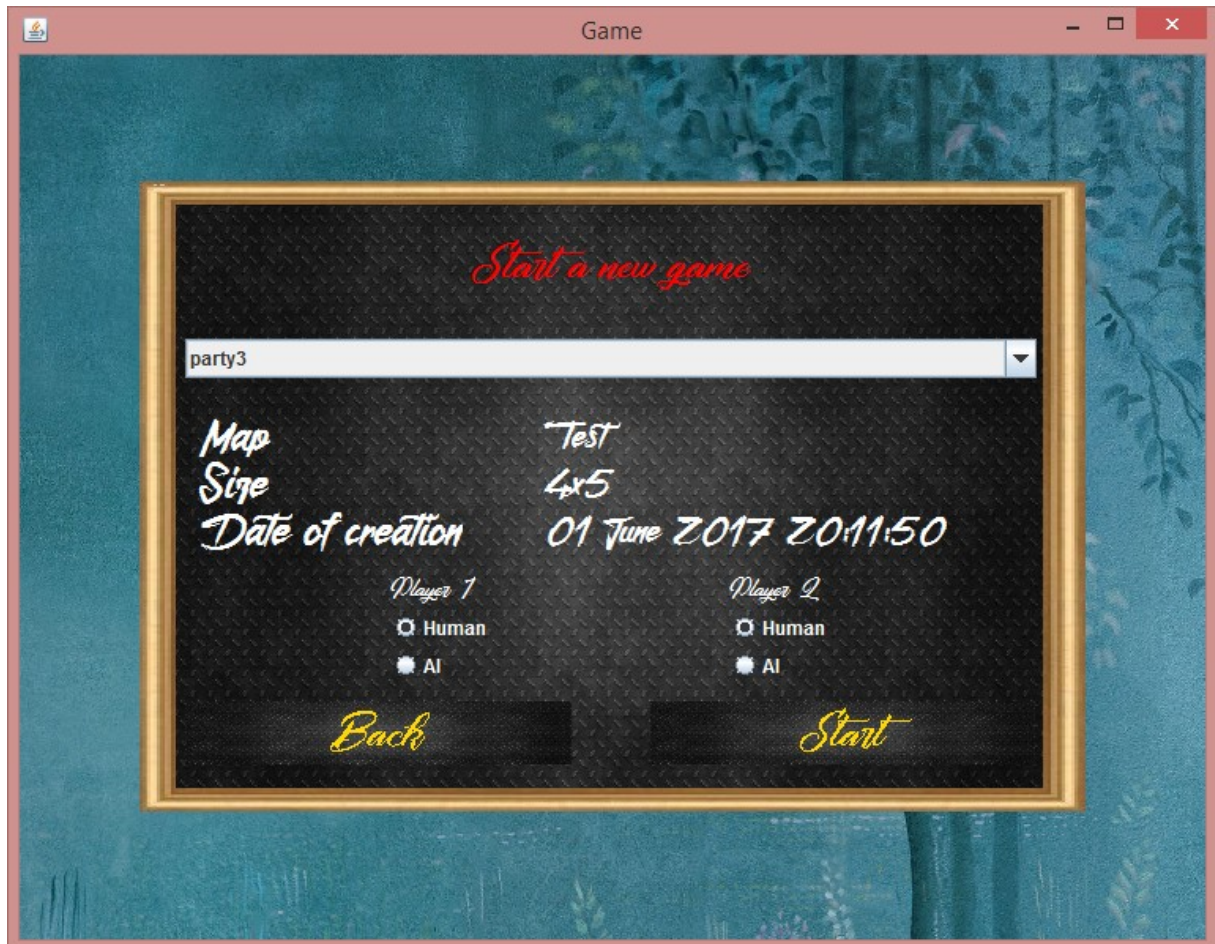


Illustration 11: Écran pour le début d'une nouvelle partie

De plus, on voit le nom du fichier, et le nom de la carte, car le nom du fichier ne correspond pas forcément à celui de la carte.

VI.2.g. ScreenMenu

La classe *ScreenMenu* qui hérite de *JPanel* représente l'écran initial du jeu (cf *Illustration 12*). En effet, celui-ci est composé :

- d'un objet *Background* ;
- possède 5 *MenuButton* qui permettent au joueur de :
 - naviguer à travers les différents menus « New Game », « Load a level », « Create a level » ;
 - afficher le menu temporaire des « Options » (cf *ComingSoon*) ;
 - quitter le jeu.

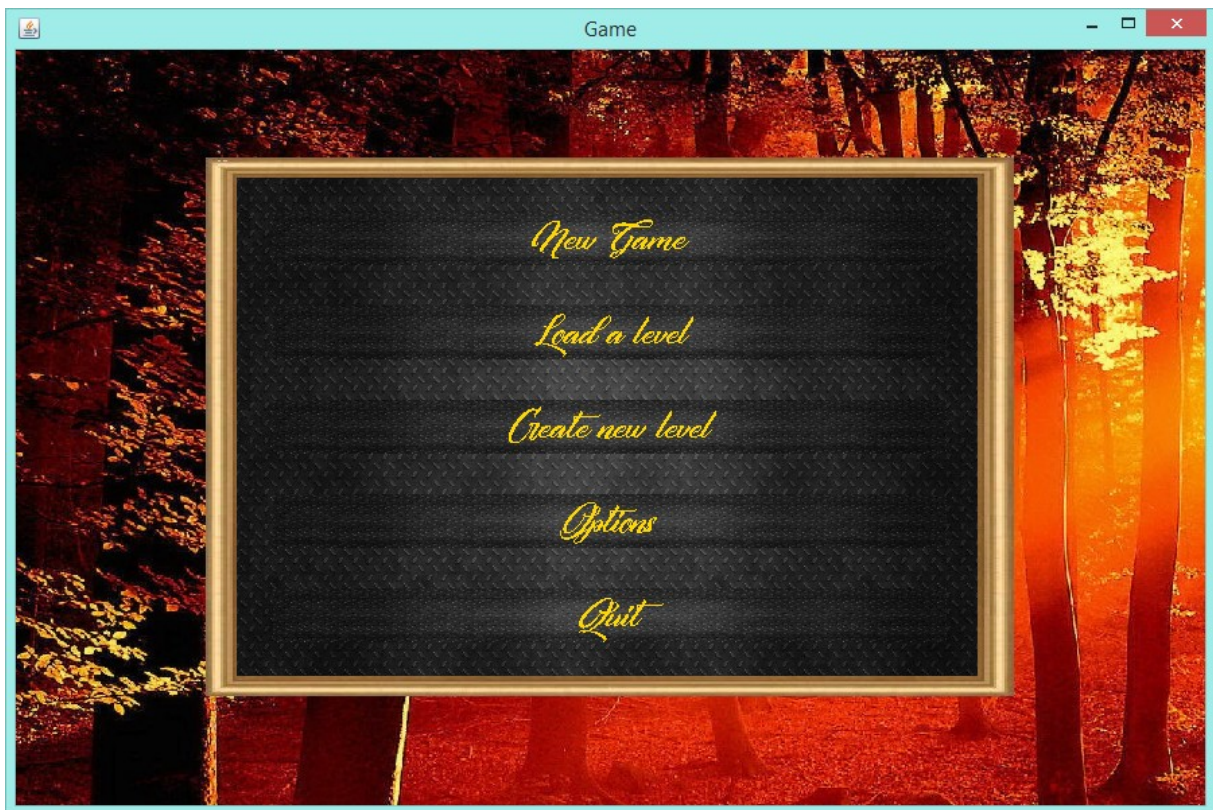


Illustration 12: Écran initial du jeu affichant les différents boutons pour la navigation.

VI.2.h. Window

Cette classe, qui hérite de *JFrame*, crée une instance de *ScreenMenu* afin de le mettre dans le contenu de la fenêtre. Les interactions des menus sont gérés par ceux-ci.

VI.2.i. Menu

Cette classe correspond au *Main*. Celle-ci appelle une instance *Window* et spécifie sa taille initiale et les différents paramètres de la fenêtre à partir des informations fournies par défaut, comme la taille minimale de la fenêtre ou sa position initiale.

VII. Logger

Le *Logger* permet de récupérer les données lors des tests d'exécution du programme. Celui-ci envoie alors des données relatives au test dans la console afin de pouvoir être analysées afin de régler les problèmes.

Le *Logger* a été implémenté à partir des méthodes déjà présentes dans Java avec le package *java.util.Logging* afin de proposer un *Logger* personnalisé qui répond aux besoins du jeu, ainsi que les différents tests à effectuer pour permettre aux développeurs de vérifier leur code.[1]

De plus, celui-ci peut envoyer les logs dans un fichier, qui peut alors être récupéré plus tard.

Il renvoie aussi un niveau d'erreur permettant d'en savoir le type : information, erreur.

VIII. Tests et résultats

Différents tests ont été mis en place afin de vérifier la fonctionnalité du logiciel proposé. En effet, la multitude de tests a permis de vérifier et corriger les différents éléments spécifiques du programme, notamment pour le système de jeu, les principes de ligne de vue, les déplacements, les attaques qui devaient être conforme aux attentes du cahier des charges.

Ensuite d'autres tests ont été effectués afin de créer une carte et de l'enregistrer dans les fichiers du jeu, puis de créer une partie avec trois personnages, séparés en deux équipes, avec différentes statistiques, dont deux personnages étant similaires.

Puis, en lançant le jeu, l'interface graphique du menu est affichée, le déplacement dans celui-ci fonctionne, avec des tests manuels car on ne peut pas vraiment mettre de tests en place pour la partie graphique. Les différents écrans communiquent bien avec les fichiers du jeu et récupèrent bien les données prévues à cet effet.

Le chargement de niveau affiche bien la carte correspondante, et les déplacements s'effectuent correctement au niveau de l'interface graphique, et les données sont actualisées au niveau des fichiers sur la position des personnages.

De plus la sauvegarde lors de la fermeture du jeu est fonctionnelle, et remet la partie à l'état précédant la fermeture lors du chargement ultérieur de celle-ci.

IX. Conclusion

Dans le cadre de notre projet, nous devons réaliser un prototype de jeu vidéo en Java. Pour cela, dans un premier temps, nous avons dû spécifier toutes les fonctionnalités et comportements désirés pour notre prototype. Puis dans un second temps, nous avons pu développer le logiciel en suivant deux axes principaux. D'une part nous avons réalisé toutes les classes et toutes les interactions internes nécessaires au fonctionnement du jeu. D'autre part, grâce à la bibliothèque *Swing*, nous avons mis en place une interface Homme-Machine, permettant à un utilisateur de pouvoir interagir avec le logiciel de la manière souhaitée.

Cela nous a permis de comprendre en profondeur les différentes phases de conception avant l'obtention d'un prototype fonctionnel. De plus nous avons dû apprendre à travailler efficacement et de manière organisée, notamment lors des phases d'intégration de code.

Le prototype est fonctionnel, et remplit tous les critères jugés vitaux lors de la rédaction du cahier des charges au début du projet.

Cependant, ce logiciel peut être enrichi afin d'ajouter de nouvelles fonctionnalités qui ne sont pas encore présentes, comme la gestion d'une intelligence artificielle. A court terme, ce prototype pourra être peaufiné afin d'implémenter tout le cahier des charges. A long terme, il pourra être utilisé comme base pour un véritable jeu, qui sera peut-être un jour commercialisé.

X. Bibliographie

- [1] Jean-Michel Doudoux. Développons en Java : <https://www.jmdoudoux.fr/java/dej/chap-logging.htm> [Consulté le 1^{er} mars 2017]
- [2] JButton (Java Platform SE 7) : <https://docs.oracle.com/javase/7/docs/api/javax/swing/JButton.html> [Consulté le 15 mars 2017]
- [3] JFrame (Java Platform SE 7) : <https://docs.oracle.com/javase/7/docs/api/javax/swing/JFrame.html> [Consulté le 15 mars 2017]
- [4] JPanel (Java Platform SE 7) : <https://docs.oracle.com/javase/7/docs/api/javax/swing/JPanel.html> [Consulté le 15 mars 2017]
- [5] A visual guide to Layout Managers : <https://docs.oracle.com/javase/tutorial/uiswing/layout/visual.html> [Consulté le 20 mars 2017]
- [6] BufferedWriter (Java Platform SE 8) : <https://docs.oracle.com/javase/8/docs/api/java/io/BufferedWriter.html> [Consulté le 28 mars 2017]
- [7] BufferedReader (Java Platform SE 8) : <https://docs.oracle.com/javase/8/docs/api/java/io/BufferedReader.html> [Consulté le 28 mars 2017]
- [8] ArrayList (Java Platform SE 7) : <https://docs.oracle.com/javase/8/docs/api/java/util/ArrayList.html> [Consulté le 28 mars 2017]
- [9] Java et la réflexivité – Apprenez à programmer en Java : <https://openclassrooms.com/courses/apprenez-a-programmer-en-java/java-et-la-reflexivite> [Consulté le 24 avril 2017]
- [10] TP3 – LE PATTERN MVC – LE JEU DU TAQUIN : https://svn.telecom-bretagne.eu/repository/ens-eleves/tc/TC131D/TP03_MVC.pdf [Consulté le 10 mai 2017]
- [11] Lambda expressions : <https://docs.oracle.com/javase/tutorial/java/javaOO/lambdaexpressions.html> [Consulté le 12 mai 2017]
- [12] File (Java Platform SE 7) : <https://docs.oracle.com/javase/8/docs/api/java/io/File.html> [Consulté le 13 mai 2017]
- [13] Hashtable (Java Platform SE 7) : <https://docs.oracle.com/javase/7/docs/api/java/util/Hashtable.html> [Consulté le 25 mai 2017]

XI. Glossaire

ActionCommand : Nom de l'action associé à un bouton

ActionListener : objet à ajouter à un composant swing pour récupérer les actions de l'utilisateur sur ce composant

AWT = Bibliothèque Java pour les interfaces graphiques

GUI = Graphical User Interface

IA = Intelligence Artificielle

id = identifiant

Java = Langage de programmation

JButton = Bouton sous Java programmable, et utilisable

JFrame = Fenêtre sous Java programmable et utilisable

JPanel = Panneau sous Java, pouvant contenir d'autres éléments

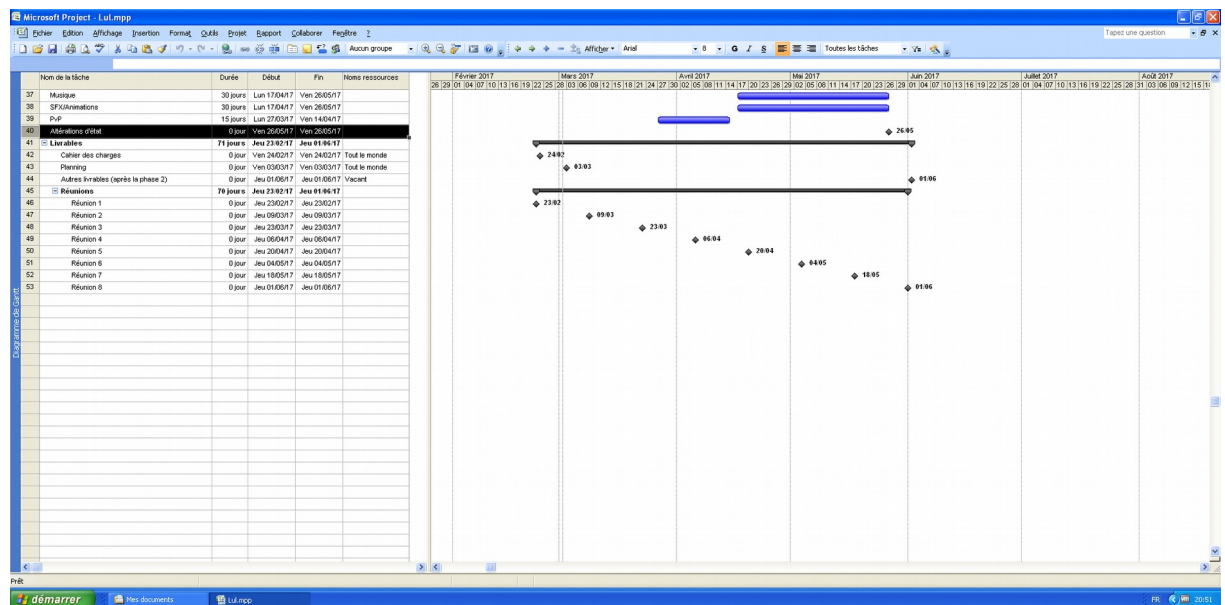
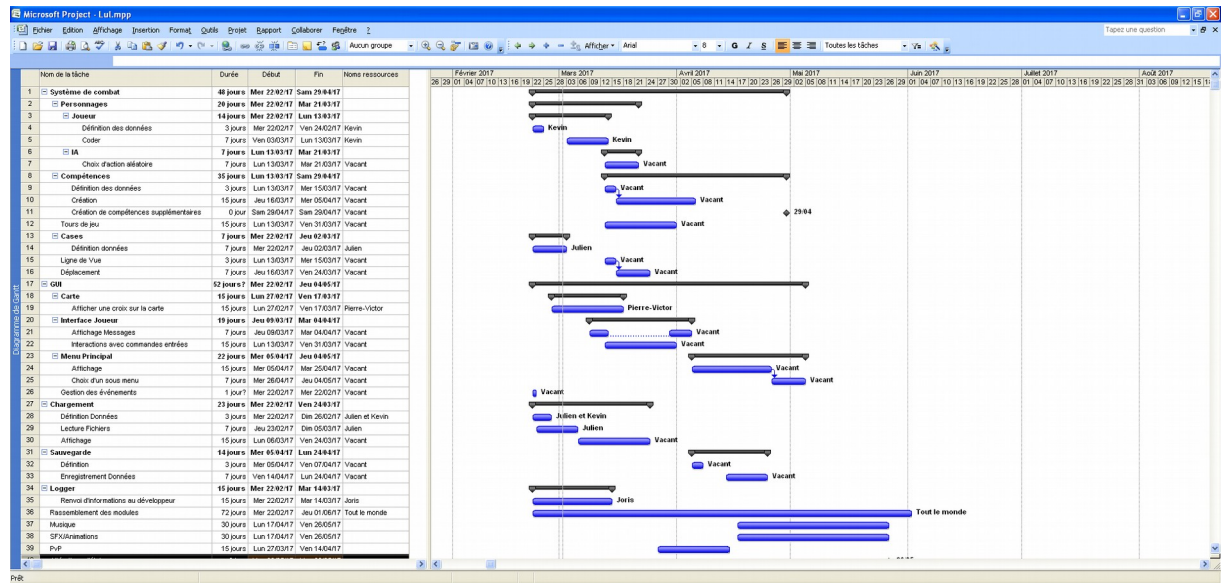
Layout = Organisation des éléments, comme une mise en page

Logger = Protocole définissant un service de journaux d'évènements d'un système informatique, c'est-à-dire un historique

PvP = Player versus Player, soit Joueur contre Joueur

XII. Annexes

XII.1. Planning initial



XII.2. Analyse du planning et comparaison du prévisionnel avec le réel

Dans l'ensemble, le planning n'a été que très peu respecté.

Les causes de cet écart sont :

- la difficulté d'évaluer même grossièrement le temps de développement de chaque partie, par manque d'expérience ;
- des dépendances entre des parties : il a été plusieurs fois impossible d'avancer une fonctionnalité du code sans que l'autre soit finie.
- l'intégration progressive du code : elle découle elle aussi de la dépendance entre les parties, par exemple une modification d'une des fonctionnalités entraîne des modifications dans d'autres fonctionnalités, qu'il est alors nécessaire de réécrire alors qu'elles devraient être finies selon le planning.
- des motivations variables.

Cependant le projet a tout de même été fini dans les temps, les livrables ont tous été rendus à la date prévue, et toutes les réunions ont été faites.