War.Militar.Game (WMG)

Golebiewski Adrien – Vendramini Thomas



Bienvenue sur ce document présentant notre jeu 2D conçu dans le cadre de l'UE "Programmation Avancée" de la formation d'ingénieur Big Data du CNAM.



Table des matières

[1-Présentation générale du jeu 3](#_Toc56975225)

[1.1 - Objectif du jeu 3](#_Toc56975226)

[2-Présentation de la carte et interactions 5](#_Toc56975227)

[2.1 - Présentation de la carte 6](#_Toc56975228)

[2.2 - Interactions Obstacles-Héro-Ennemis 7](#_Toc56975229)

[3-Construction générale du code 9](#_Toc56975230)

[3.1-Implémentation de l’interface graphique 9](#_Toc56975231)

[3.2-Initialisation de la carte 14](#_Toc56975232)

[3.3 – Intéraction héros/obstacles/ennemis 20](#_Toc56975233)

# 1-Présentation générale du jeu

## - Objectif du jeu

Le joueur doit pouvoir contrôler un héros dans un environnement hostile. L'environnement est représenté par une carte non scrollable constituée d'une zone d'eau, d'une zone de sable et d'une zone d'herbe. Elle constitue la seule carte du jeu. Des décors et autres objets ont également été ajoutés.

Ce héros est représenté par un militaire. Il va devoir faire face à des ennemis (représentés par des avions au nombre de 2) localisés sur la carte.

Avec un taux d'énergie initialisé à 1000, le héros doit rester en vie et attaquer les ennemis représentés par des avions. Il peut également se soigner, donc gagner en énergie. A l’inverse, sa jauge d’énergie est amenée à baisser à chaque déplacement et lorsqu’il subit une attaque.

Une autre caractéristique du héros est le nombre de « couts restants » ce qui constitue une contrainte supplémentaire. En effet le « nombre de cout restant », initialisé à 100, diminue à chaque fois que le héros se déplace, qu’il se soigne ou attaque l’ennemi. C’est-à-dire à chaque fois qu’une touche du clavier est pressée.

L'objectif du joueur est donc de tuer tous les ennemis et faire maintenir en vie le héros (Energie > 0). En tuant des ennemis, le héros gagnera de l’énergie. **Le joueur gagne la partie** si tous les ennemis sont tués, c’est-à-dire ne sont plus la carte.



**Le joueur perd en revanche la partie** si son énergie est à zéro ou bien son nombre de coups restants est à zéro. En effet à chaque attaque ennemie, le héros perd de l’énergie. Il en perd également à chaque déplacement dans la carte dans une moindre mesure. A chaque déplacement, le nombre de mouvement diminue aussi.



*Fenêtre Game Ove du jeu*

Un défi de gagner pour le joueur avec un nombre maximum de « coups restants » du héros est ainsi possible.

Un comptage du nombre de « mouvements restants » et de l’énergie est alors effectué sous forme de jauge écrite dans le coin supérieur droit de la carte.



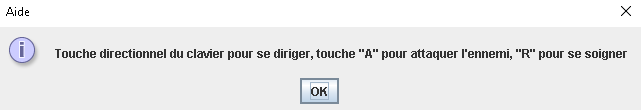
*Jauge écrite des caractéristiques du héro*

En lançant le jeu, le joueur aura la possibilité de choisir 1 héros parmi 2, chacun avec sa caractéristique. Le héros 1 peut nager, tandis que le héros 2 peut voler.



*Fenêtre choix du personnage*

Le déplacement du héros s’effectue grâce aux touches du clavier. Dans la fenêtre du menu principal, les touches du clavier impliquées dans le jeu sont explicitées dans l’onglet « Aide ».



*Onglet Aide*

# 2-Présentation de la carte et interactions

## 2.1 - Présentation de la carte

La carte correspond à une grille constituée de tuiles (appelés tile). Elle est de taille « this.getWidth » largeur et « this.getHeigh » hauteur. Chacune des tiles est de taille 32X32 pixels. Ainsi, les coordonnées du héro augmentent de +32 à chaque déplacement. Aussi, nous avons fait attention à bien dimensionner nos images représentant les décors, animations, héros et ennemis en 32x32 pixels ou bien en multiple de 32.

1 tile

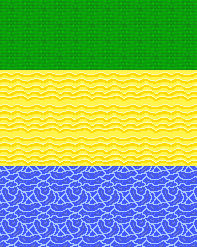
This.getHeight

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
|  |  |  |  |  |
|  | **Héro** | +32 | +32 |  |
|  | +32 |  |  |  |
|  | +32 |  |  | Ennemi |
|  |  |  |  |  |

This.getWidth

Nous avons ensuite rempli cette carte de différents décors et zones :

* Zone Bleue représentant l’eau :
* Zone verte représentant de l’herbe
* Zone jaune représentant le sable



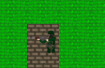
*3 zones possibles*

## 2.2 - Interactions Obstacles-Héro-Ennemis

Ces 3 zones peuvent être accessibles par les 2 personnages. On accède cependant à la zone d’eau de deux différentes manières. Le personnage 1 y accède en nageant tandis que le personnage 2 y accède en volant.

La carte contient également des décors appelés « obstacles » comme par exemple un mur de brick, une forêt sur le sol dure et un bateau ou une bouée dans la zone d’eau.

Ces obstacles ne peuvent être accessibles que par le personnage 2 qui a la capacité de voler. En se déplaçant sur la case de l’obstacle, il s’affichera par-dessus l’image de l’obstacle.



*Personnage 2 en plein vol au-dessus de l’obstacle*

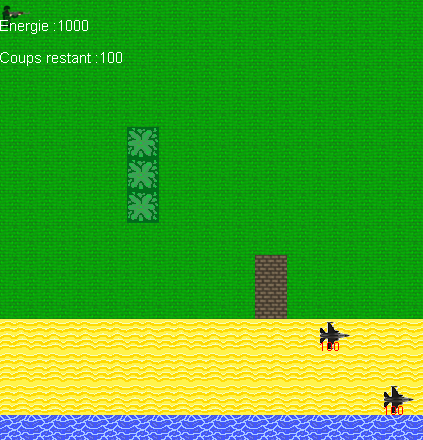
En revanche le personnage 1 ne pourra pas accéder aux cases de l’obstacle. Il devra contourner les obstacles pour accéder à l’endroit voulu.



*Incapacité du personnage 1 à passer l’obstacle forêt car ne sachant pas voler*

Cette carte se caractérise aussi par les interactions entre le héros et l’environnement ennemi avec en l’occurrence : les avions (au nombre de 2) placés sur la carte à des postions fixes ainsi qu’un héros placé aussi à une position fixe.

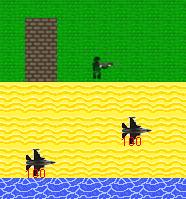
On retrouve également la jauge d’énergie du héro ainsi que celle du nombre de coups restants. Par ailleurs si le joueur appuie sur la touche « R » du clavier, l’énergie du héro augmente. En revanche, le nombre de coups restants diminue.



Comme le héros, chaque avion a sa propre jauge de vie textuelle initialisée, située juste en dessous.



A chaque déplacement du héro dans la carte, un déplacement de l’ennemi s’effectue de manière aléatoire en fonction des déplacements du héros.



Les déplacements de l’ennemi ne subissent pas comme pour les héros les contraintes de collisions engendrées par certains décors. En effet, les avions étant des objets volants, ils passent par-dessus les décors.



Si le héros est a moins de deux cases de l’ennemi, il inflige des dégâts à l’ennemi. Il atteint l’ennemi lorsqu’il est positionné dans la même case de l’ennemi ou bien dans celle à côté.



C’est uniquement à ce moment précis que le joueur, en appuyant sur la touche clavier « A » va permettre à l’héro de tuer l’ennemi. En effet 1 touché de la touche « A » du clavier correspond à une attaque du héros. L’objectif du héro est de vider l’énergie de l’avion, en attaquant plusieurs fois.

Si l’ennemi n’a plus de vie, il disparaît à la suite des attaques du héros. Si le héros arrive à faire disparaitre tous les ennemis, il gagne la partie.

Par ailleurs, si le héro se trouve à moins de 3 cases de l’avion, il subit une attaque de ce dernier et perd ainsi de l’énergie. Si le héro n’a plus d’énergie, il perd la partie.



# 3-Construction générale du code

La source code de ce jeu peut être divisée en plusieurs parties :

* Les fenêtres du jeu : menu principal, choix des personnages, jeu, gameover, victoire, fenetre de la carte
* Les « objects » présents dans la carte (classe mère) contenant notamment
* Les décors (classes filles)
* La gestion de l’ennemi (classe ennemi)
* La gestion du choix des héros, des touches clavier, des attaques/ripostes ennemis et de l’affichage des décors et autres composants de la carte (classe carte)

Les commentaires apportés dans le code permettent d’expliciter l’implémentation du jeu et notre raisonnement.

Cependant, il convient de présenter 3 aspects importants du jeu et leurs implémentations en langage JAVA car ils constituent une base au niveau de l’expérience utilisateurs :

* L’implémentation de l’interface graphique
* L’initialisation de la carte
* Interactions entre le héros, ennemis et obstacles.

## 3.1-Implémentation de l’interface graphique

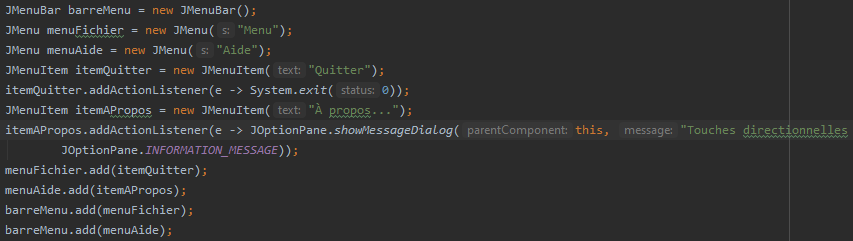
L’interface graphique comporte :

* 1. 1) **Le menu principal du jeu :**
  2. il s’agit de la fenêtre s’affichant au lancement du jeu. Cette fenêtre affiche le titre du jeu et permet de lancer une nouvelle partie ; cela fait disparaître la fenêtre et fait apparaître la fenêtre du choix du joueur.
  3. La position des composants dans la fenêtre « menu » est réalisée grâce à un *JPanel* . Il s’agit d’un conteneur élémentaire destiné à contenir d'autres composants et faciliter la représentation des objets.
  4. En l’occurrence, nous y avons placé une image qui constitue la jaquette de notre jeu :
  5. 
  6. *Ajout de l’image dans le panel de la fenêtre menu*

Ainsi que des boutons. Le composant pour créer un bouton est le JButton.



Et un JMenu étant un bouton que l'on trouve directement dans la barre de tâches et il possède des JMenuITem qui sont eux des éléments du menu.

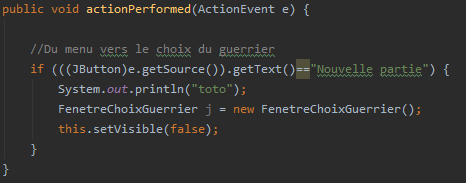


Les JMenu Items et les JBouton sont des éléments graphiques sur lesquels l'utilisateur peut cliquer pour déclencher une action. Le bouton ne fait rien tant que l'utilisateur n'a pas cliqué dessus.

Comme avec les boutons, il y a deux manières de gérer les actions des menus, les listeners ou les Action. Pour les listeners il suffit d'implémenter ActionListener sur la classe, d'utiliser la méthode addActionListener sur le JMenuItem/Jbuttons pour ajouter le listener à l'élément du menu et de tester la source de l'événement pour savoir qui a cliqué.

Aussi, lorsque l'utilisateur clique sur le bouton à l'écran, celui-ci déclenche un événement d'action. Il en résulte l'invocation de la méthode actionPerformed de l'auditeur d'action (la seule méthode dans l'interface ActionListener). L'unique argument de la méthode est un objet ActionEvent qui donne des informations sur l'événement et sa source.

En l’occurrence comme le montre la capture ci-dessous, en appuyant sur le bouton « nouvelle partie », le joueur accède à la fenêtre du choix du héro.



* 1. Une option de quitter le jeu depuis la fenêtre du menu est possible avec un Action Listener et un « System.exit ».

1. 

Ce principe d’Action Performed et d’Action Listener présenté précédemment est repris constamment pour chaque fenêtre et chaque passage de fenêtre à une autre du jeu.

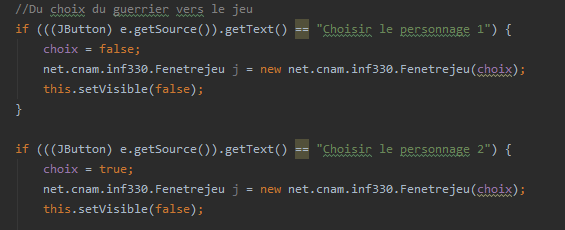
* 1. 2) **La fenêtre de sélection du héros :**
  2. il s’agit de la fenêtre qui s’affiche au lancement d’une nouvelle partie. Elle permet de choisir parmi les 2 héros, qui sera contrôlé par le joueur, et de le sélectionner.

Un Jpannel est également construit contenant les boutons « Choisir le personnage 1 » et « Choisir le personnage 2 ». A la différence du JPannel du Menu principal, celui-ci contient également des JLabel pour écrire du texte.



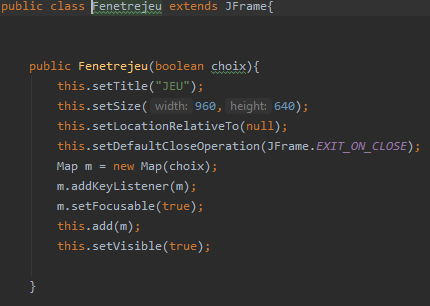
* 1. La sélection du héros par le joueur fait disparaître la fenêtre. Elle fait apparaître la fenêtre du jeu en reprenant comme pour la fenêtre du menu, les principes d’Action et Event Listener.

Une variable « choix » initialisée à False au début du code est ensuite mise à jour dans l’Action Performed pour signifier quel héros a choisi le joueur.



* 2. 3) **La fenêtre de jeu :** il s’agit de la fenêtre principale du jeu.
  3. Cette fenêtre contient uniquement la carte « map » ainsi que tous ses composants.

On y associe également l’évènement « Key Listener » qui quand l'utilisateur tape au clavier. Le listener capte donc l’information. Plus précisément, les événements « Key Listener » sont déclenchés par le joueur à travers les touches du clavier.

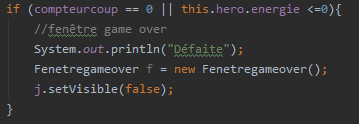
1. 

*Classe « Fenêtre Jeu »*

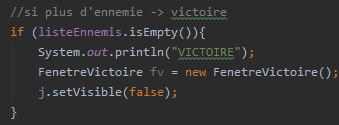
* 1. 3) **Les fenêtres Game Over et Victoire :**

Elles s’affichent à l’écran en fonction de l’avancée du héros dans sa quête. En l’occurrence :

* La fenêtre « Game Over » lorsque le héros n’a plus d’énergie ou plus de coups restants.

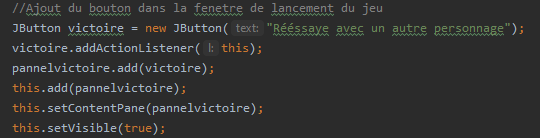


* La fenêtre « Victoire » lorsque tous les ennemis ont été tués

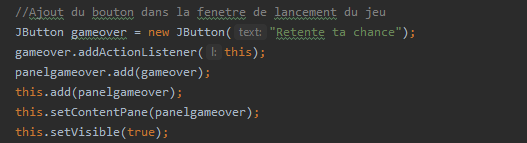


La fenêtre Game Over et Victoire contiennent un JButton permettant d’accéder au menu principal du jeu pour pouvoir retenter une partie après un échec ou une victoire.

Le principe d’« Action Listener» et de «Performed Listener» est une nouvelle utilisée.



Action Listener pour la fenêtre victoire vers la fenêtre du menu principal



Action Listener pour la fenêtre GameOver vers la fenêtre du menu principal

## 3.2-Initialisation de la carte

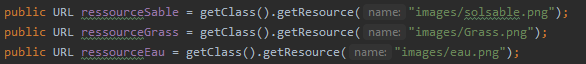
Elle est constituée/caractérisée par

1. **Des zones**

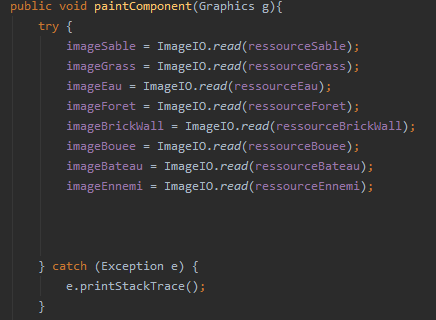
D’eau en bleue, d’herbe en verte et de sable en jaune.

Ces zones correspondent à une duplication d’images placées. 1 image 32X32 pixels pour chaque tile.

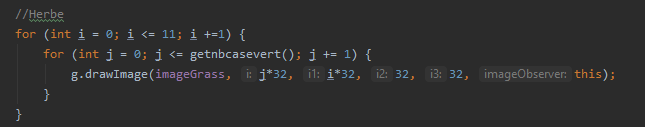
Ces images sont récupérées dans cette partie du code :



Puis lues par la fonction « imageIO.read » à l’intérieur de variables dans une fonction « paint component ».



Enfin, la fonction g.drawImage permet d’afficher les images dans chacune des tile à l’aide de boucles « for » :



*Exemple d’affichage avec l’Affichage de la zone herbe dans la map*

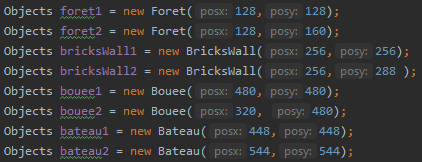
1. **Des objets/décors**

Les objets sont issus d’une classe « Objects » qui est elle-même étendue.

En effet, chaque décor de la carte est représenté par une classe du même nom, une classe fille qui étend la classe mère « Objects ».

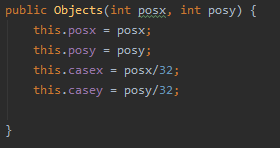
Chaque objet se caractérise par :

* sa position (this.getposx(), this.getposy()) pour l’affichage dans la carte.



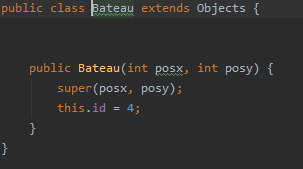
*Coordonnées des objets/décors inscrites dans le dur dans la classe « Map », pas d’aléatoire*

* sa position (this.getCasex(), this.getCasey()) déplacement pour la comparaison avec la position du héro (gestion des obstacles)

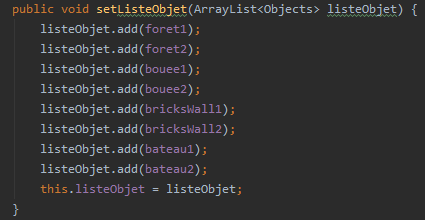


*Caractéristiques de chaque Objects*

Ensuite, chaque décor, à travers sa propre classe (classe fille), récupère ces informations grâce à l’expression « super() ». De plus une autre caractéristique est apportée, propre au type de décor : il s’agit de l’identifiant. 1 id par décor pour faciliter la reconnaissance des décors dans le code. En l’occurrence comme le montre la capture ci-dessous, le décor bateau a comme id , id =4.



Ainsi, nous avons pu afficher ces objets en créant notamment une liste d’objets et ainsi ajouter les objets crées dedans.



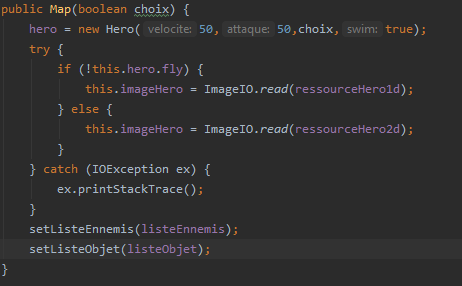
En appliquant une boucle for dans une fonction « paint component » sur cette liste, cela nous a grandement facilité l’affichage :



*Affichage dans la classe map d’un objet seul sur la carte (1 objet = 1 case grâce aux coordonnées de l’objet)*

1. **Un héros :**

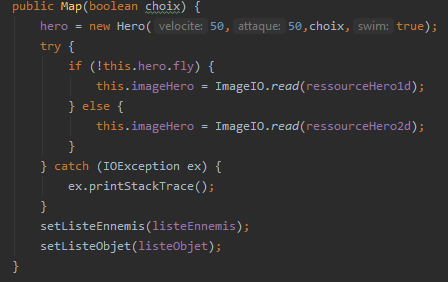
L ’impact du choix du héro réalisé dans la fenêtre « choix du héros » se répercute dans la classe « Map » avec ce bout de code suivant qui constitue le contenu du constructeur de la classe Map :



Notre objectif était en effet de faire apparaître avec la fonction « ImageIO.read » :

* L’image du héro1 si le joueur a cliqué sur « Choisir le personnage 1 » dans la fenêtre associée
* L’image du héro2 si le joueur a cliqué sur « Choisir le personnage 2 » dans la fenêtre associée

Cela se matérialise par le contenu suivant dans le constructeur Map sachant que le héro 1 ne sait pas voler :



Cela se matérialise aussi par l’appel de « Map » dans la fenêtre du jeu :



A chaque partie, le héro est positionné au même endroit : coin supérieur droit de la carte (position 0,0). Les positions du héros sont en effet initialisées à zéro dans le code.



Le héros est représenté par une image lue par une fonction imageIoReader() puis affichée par un g.draw Image aux coordonnées initiales.



En déplacement, l’image du héro switch en fonction du déplacement vers la gauche ou vers la droite (une image héro tourné vers la gauche et une image héro tourné vers la droite).

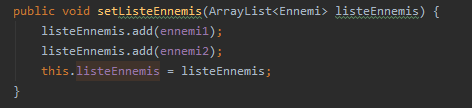
1. **Des ennemis**

Les ennemis sont représentés par des avions militaires et caractérisées par une jauge d’énergie écrite. Comme pour le héros, leurs positions sont initialisées dans le dur dans la classe « Map ».

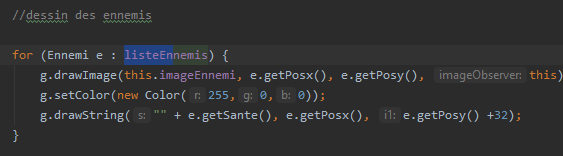


Les ennemis sont représentés par des images lues par une fonction imageIoReader() puis affichées par un g.draw Image aux coordonnées initiales.

Pour faciliter l’affichage et comme pour les décors, nous avons construits une liste d’ennemis contenant les ennemis présents dans la carte.



L’affichage s’effectue ainsi de cette manière avec une boucle for qui parcoure la liste :



Enfin, la santé de l’ennemi est affichée comme ce si, en dessous (+32) de l’avion :



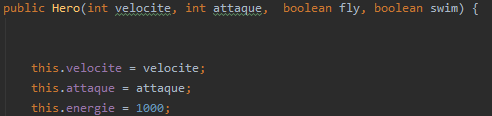
1. **Une jauge écrite d’énergie du héro et un nombre de coups**

Le nombre de coups est initialisé dans le dur, toujours dans la classe « map » à 100



Ce compteur diminuera à chaque fois que le joueur fera pression sur une des touches du clavier impliquée dans le jeu.

L’énergie du héro est en revanche initialisée dans la classe du héro associé :



Elle diminuera à chaque déplacement du héro et à chaque riposte de l’ennemi.

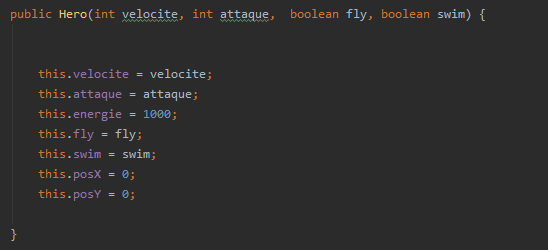
Les deux caractéristiques du héros sont ensuite affichées dans la classe « Map » avec la fonction g.drawString.



## 3.3 – Intéraction héros/obstacles/ennemis

Déplacement du héro

La classe Héro permet de caractériser ce dernier :



On retrouve notamment l’énergie, présentée précédemment ainsi que ces positions initiales posX et posY.

Dans la classe « map », les positions de héro « this.hero.getposx » et « this.hero.getposY » correspondent donc à la position du héro dans la carte. Nous les utilisons **uniquement** pour l’affichage du héros de manière visuelle sur la carte. Elles apparaissent dans le code uniquement dans la fonction « paint component »

En revanche, le déplacement du héros se réalise par les deux variables « this.casex » et « this.casey ». Elles sont initialisées à 0 en début de code dans la classe « Map ».



Nous avons décider d’avoir ces deux types de position du héros afin de gérer au mieux la confrontation du héro avec l’ennemi et la gestion de collisions présentées plus loin dans le rapport.

Pour gérer de manière précise le déplacement du héros, nous avons opté pour détecter les événements issus du clavier, tels que les touches appuyées ou relâchées. Cela est mis en œuvre par l’intermédiaire de **l’interface KeyListener** implémentée par la classe Map :



L’interface est définie dans le package java.awt.event qu'il faut donc importer. Elle contient des méthodes intéressantes pour notre jeu comme la méthode suivante :



Cette méthode a comme paramètre un objet de type KeyEvent; celui-ci possède des méthodes permettant de savoir quelle touche a été utilisée. En l’occurrence, les méthodes :

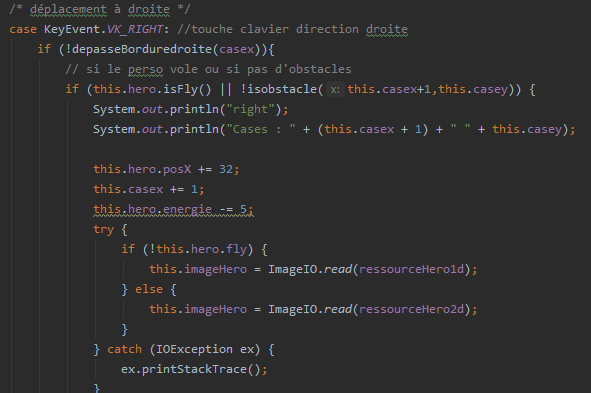
* KeyEvent.*VK\_LEFT*
* KeyEvent.*VK\_RIGHT*
* KeyEvent.*VK\_UP*
* KeyEvent.*VK\_DOWN*
* KeyEvent.*VK\_R*: //touche clavier lettre "R" pour gagner de l’énergie
* KeyEvent.*VK\_A*: //touche clavier lettre "A" pour attaquer

Pour pouvoir gérer les déplacements, l’énergie et les attaques du héros.

Pour chacune de ces méthodes, est mis à jour/associée :

* La position d’affichage this.hero.posY (ou X) du héro
* La position de déplacement this.casex(ou Y) du héro
* Une perte d’énergie -5 du héro
* Changement d’image du héro (en fonction du déplacement gauche ou droite)
* Affichage des coordonnées de position de déplacement du héro dans la console

Exemple de mise à jour lorsque le héro fait un déplacement vers la droite :

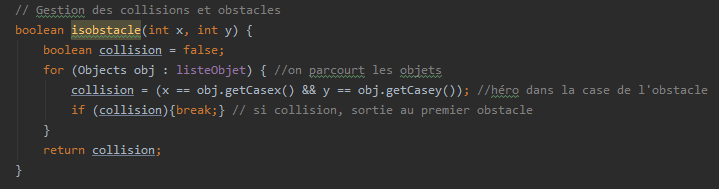


Gestion des obstacles/bordures

Dans notre jeu, nous avons défini 2 types d’obstacles :

* Les obstacles représentés par les objets

La gestion des obstacles est gérée par la fonction « is obstacle » de la classe « Map ». En parcourant la liste de tous les objets, elle vérifie si le héro se trouve dans la case de l’obstacle ou non.

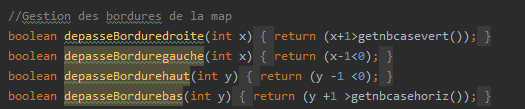


Cette fonction est ensuite implémentée dans la gestion des déplacements des héros à travers une condition « if » pour vérifier qu’il n’y a pas d’obstacle lors du déplacement



* Les contours de la carte permettant au héros de ne pas sortir de la carte

Cette caractéristique se matérialise par les méthodes booléennes suivantes :



Puis réimplémentée dans la gestion des déplacements des héros à travers une condition « if » pour vérifier que le héro ne dépasse pas le cadre de la carte.

Cette notion de collision et de bordure est également respectée par l’activité de déplacement de l’ennemi.

Déplacements des ennemis

Le manque de temps nous a empêché de développer une petite IA permettant à l’ennemi de se déplacer de manière autonome.

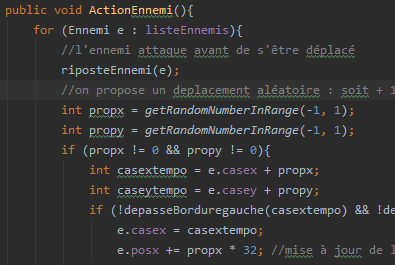
Nous avons ainsi privilégié l’aspect aléatoire pour permettre à l’ennemi de se déplacer sur la carte.

Pour cela, nous avons décidé de lier le déplacement de l’ennemi au déplacement du héros pour coller le plus à la réalité.

Nous avons dans un 1er temps généré un nombre aléatoire



Ensuite, notre idée est d’appliquer ce nombre aléatoire dans une fonction « Action Ennemi() » dans laquelle on additionne ce nombre aléatoire par l’ancienne position de l’ennemi (ancienne position + nombre aléatoire). Enfin on le multiplie par 32 étant la taille d’une tuile et permettant le déplacement du héros.



Ainsi la nature du déplacement du héro provient de ce nombre aléatoire :

* **0** = pas de déplacement, l’ennemi reste statique
* **1** = déplacement du héro vers l’avant
* **-1** = déplacement du héro vers l’arrière

Cette fonction prend également en compte la gestion des collisions comme expliqué précédemment avec le mouvement du héros.

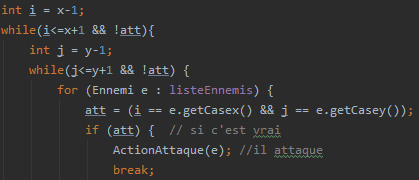
Gestion Attaque du héro – riposte ennemi

Nous avons décidé de ne pas réaliser d’animations dans la gestion des attaques. Elles se matérialisent uniquement par :

* la perte d’Energie du héro représentée par une baisse de la jauge d’énergie du héros (située en haut à gauche de la carte)
* la perte d’énergie de l’ennemi représentée par une baisse de la jauge d’énergie de l’ennemi (situé juste en dessous de ce dernier)

La mort de l’ennemi se matérialise par l’apparition de la fenêtre GAME OVER. La mort de l’ennemi se matérialise par sa disparition de la carte (fonction remove)

Le héros ne peut attaquer l’ennemi si et seulement s’il se trouve à moins de 2 cases de l’avion. On peut donc concrètement appuyer sur la touche « A » du clavier lorsque le héros se trouve dans la même case de l’avion ou dans la case à côté :



Chaque attaque du héro implique une baisse de santé de l’ennemi mais un gain d’énergie pour le héro comme l’exprime ces 2 lignes de code de la fonction « void ActionAttaque »



Concernant l’ennemi, il ne peut attaquer le héros si et seulement s’il se trouve à moins de 3 cases de l’avion :



Chaque attaque de l’ennemi implique une baisse de l’énergie du héros :



Après chaque attaque de l’ennemi, ce dernier se déplace (de manière aléatoire) pour se protéger d’une riposte du héros.