Haute École Libre de Bruxelles Ilya Prigogine

Enseignement supérieur

Informatique : Développement d’applications



Département Technologies et Économies

Campus de la Plaine, Bâtiment HA

Boulevard du Triomphe 200/01

1050 Ixelles

**HELBArmy**

Réalisé dans le cadre du cours de programmation Java 3

EL BOUCHTILI Imaddine

Professeur : M. Riggio

Année scolaire : 2024-2025

Table des matières

[1. Introduction : 4](#_Toc186395756)

[2. Fonctionnalités : 5](#_Toc186395757)

[2.1 Board 5](#_Toc186395758)

[2.2 Entité 5](#_Toc186395759)

[2.2.1 Ville 5](#_Toc186395760)

[2.2.2 Arbre 6](#_Toc186395761)

[2.2.3 Collectable 6](#_Toc186395762)

[2.3 Unité 6](#_Toc186395763)

[2.3.1 Collecteur 7](#_Toc186395764)

[2.3.2 Déserteur 7](#_Toc186395765)

[2.3.3 Cavalier 7](#_Toc186395766)

[2.3.4 Piquier 8](#_Toc186395767)

[2.4 Raccourci clavier 8](#_Toc186395768)

[3. Analyse : 10](#_Toc186395769)

[4. Limitations et développement futur : 13](#_Toc186395770)

[5. Conclusion : 15](#_Toc186395771)

[6. Annexe : 16](#_Toc186395772)

# 1. Introduction :

Mon application est un jeu de simulation de batailles militaires avec plusieurs acteurs. Chaque acteur à un rôle spécifique avec des comportements différents. Le jeu a été développé en *Java*, à l’aide de la technologie *JavaFX*, pour la plateforme Linux Ubuntu. Son nom est **HELBArmy**.

Ce rapport détaillera la manière dont le jeu a été conçu et développé. Il sera divisé en plusieurs sections dont voici les explications :

* Fonctionnalités
  + Présentation détaillée des fonctionnalités offertes par mon application, d’un point de vue fonctionnel et technique, ainsi que les obstacles et solutions pour celle-ci.
* Analyse
  + Structures fonctionnelles et techniques de certains points spécifiques de l’application expliquée à l’aide de schémas et code.
* Limitations et développement futur
  + Les limites rencontrées lors du développement de l’application. Les fonctionnalités qui peuvent être améliorées ou rajoutées. Et les points de défaillance qui peuvent rendre la simulation impraticable.
* Conclusion
  + Ce que j’en ai tiré de ce projet.
* Annexe
  + Ressources supplémentaires utiles pour comprendre les détails du développement de l’application.

# 2. Fonctionnalités :

**HELBArmy** est une simulation de batailles militaires. Par ce fait, aucun intervenant n’est nécessaire au fonctionnement de la simulation. Néanmoins, des touches du clavier peuvent être actionnées à tout moment pour permettre une action.

Le jeu est composé d’un plateau, *board*, lui-même composé de 3 éléments statiques, les entités et 4 éléments dynamiques qui sont des entités qui peuvent bouger, les unités.

## Board

Le *Board* est le plateau en 2D sur lequel toutes les entités sont présentes. Sa taille est ajustable librement, bien que certaines dimensions soient plus optimales pour rendre la simulation fluide et sans bugs. Par défaut, les entités prennent une case du plateau et ont une position (x ; y) unique.

## Entité

Par défaut, les entités n’ont pas la possibilité de bouger sur le plateau. Elles ont une position fixe déterminée au lancement du programme et ne changent plus leurs positions jusqu’à une réinitialisation de simulation.

## Ville

Les villes sont les seules entités à prendre plus d’une case sur le plateau. Il y a deux villes sur le plateau, une au Nord et une au Sud. Elles ont chacune une taille de 5x5 et deux portes. Une porte dédiée à la sortie des unités, et l’autre qui sert de dépôt de bois.

*Pour mettre en place la ville au Nord, j’ai directement modifié la ressource pour en faire une rotation. La meilleure solution aurait été de la modifier automatiquement lors de son affichage.*

Le bois est la seule ressource des villes. Elle sert de monnaie pour savoir quelle unité sera générée par la ville. La génération d’unité est faite aléatoirement en fonction du nombre de bois total de la ville.

## Arbre

*Lorsqu’un arbre est sur le point de réapparaitre, il faut faire attention à ce qu’une unité ne soit pas présente sur la position de l’arbre. En effet, lorsqu’un arbre tombe, sa position n’est plus protégée et les unités peuvent prendre sa position.*

Les arbres sont les seules entités qui donnent du bois. Ils peuvent être coupés pour en obtenir le bois qu’ils contiennent. Une fois que le nombre de bois d’un arbre atteint 0, il disparait pendant 30 secondes avant de réapparaitre au même endroit avec sa quantité de bois initial.

## Collectable

Pour ajouter de l’action à la simulation, deux types de collectables sont présents sur le drapeau. Le drapeau et la pierre philosophale. Ces collectables peuvent être ramassés par les unités en mouvement.

*Lors de la mise en place des collectables, dans la simulation, j’ai dû revoir la manière dont les unités bougent. En effet, même si les collectables prennent une case sur le plateau, les unités peuvent tout de même passer au travers.*

Les drapeaux apparaissent toutes les deux minutes à une position aléatoire sur le plateau. Une fois celui-ci apparu, toutes les unités changent leurs comportements spécifiques pour se diriger vers celui-ci. Le drapeau récupéré confère un bonus à l’équipe de l’unité qui l’a récupéré.

Au début de la partie, deux pierres philosophales apparaissent sur le plateau. Les unités n’ont pas « conscience » d’où elle se situe. Lorsqu’une unité marche dessus, elle a 50% de chance de mourir instantanément ou d’être invincible jusqu’à la fin de la simulation.

## Unité

Les unités sont des entités, qui peuvent changer de position au fur et à mesure de la simulation. Chaque unité a des statistiques prédéfinies :

*J’ai beaucoup réfléchi avant de mettre en place le système de multiplicateur de dégât. J’en suis arrivé à la conclusion que c’était plus simple de mettre cette responsabilité directement sur les unités, plutôt que sur un « contrôleur » de dégât.*

* Point de vie
* Dégât
* Multiplicateur de dégât

De plus, lorsqu’une unité est adjacente à une unité adverse, un combat s’enclenche, jusqu’à la mort d’une d’elles.

## Collecteur

Le seul objectif des collecteurs est de récupérer du bois en coupant des arbres pour leurs villes. Une fois l’inventaire d’un collecteur plein, il se dirigera vers le dépôt de bois de sa ville et recommencera sa route au prochain arbre.

## Déserteur

Les déserteurs quant à eux, ils vont tout faire pour tuer les collecteurs ennemis. Cependant, ils fuiront toute autre unité.

## Cavalier

Les cavaliers sont les unités qui ont le comportement le plus complexe. En effet, ils agissent non seulement, en fonction les ennemis, mais aussi en par rapport à leurs alliers.

*La distance de sécurité par équipe est gérée par une classe statique, HorsemenManager.java. J’ai fait ce choix, car la logique derrière la distance de sécurité est conséquente et prendrait énormément de place dans la classe Horsemen.java.*

Lorsqu’un cavalier est généré par une ville, deux scénarios s’offrent à lui :

* Si le cavalier est plus proche d’un déserteur ennemi, qu’a un cavalier allier, il se dirigera vers le déserteur.
* Si le cavalier allier est plus proche, il ajustera sa distance par rapport à lui en fonction de la distance de sécurité établie entre eux.

Une image contenant capture d’écran, vert

Description générée automatiquementLa distance de sécurité entre les cavaliers d’une même équipe augmente d’une case à chaque « tick ». Lorsqu’un cavalier se trouve en combat, la distance de sécurité des cavaliers de son équipe passe instantanément à un. Cette règle permet d’avoir parfois des situations plutôt jolies, comme dans cette image où les cavaliers Nord étendent au fur et à mesure leurs distances de sécurité.

## Piquier

Quand un piquier apparait, une position lui est attribuée. Son objectif est de se rendre à cette position et de défendre les alentours. Lorsqu’une unité ennemie rentre dans son champ de vision, le piquier ira vers lui pour le tuer. La vision des piquiers d’une équipe est égale au nombre de piquiers qui la compose.

*Lorsqu’une position est attribuée à un piquier, il faut faire attention à ce que la position ne soit prise par aucune autre entité, comme un arbre entrain de réapparaitre.*

## Raccourci clavier

Pour permettre un développement et une présentation plus fluide, une poignée de raccourcis clavier ont été implémentés. Il permet de faire des actions, telles que :

* Générer des unités spécifiques dans une ville
* Stopper des unités
* Tuer toutes les unités
* Faire apparaitre un collectable
* Recommencer la simulation

*Par exemple, pour créer un collecteur je faisais ceci :*

newUnits = new Collector(

new Position(

getUnityExitX(),

getUnityExitY()

),

this.getSide(),

gameboard

);

*En fait, le seul changement qu’il y avait d’une unité à une autre était le nom du constructeur.*

*Lors de l’implémentation du raccourci clavier permettant de créer des unités, je me suis rendu compte que le code permettant de générer des unités dans les villes n’était pas durable. En effet, j’avais de la duplication claire.*

*J’ai donc effectué mes recherches et en suis arrivé à ce modèle :*

public void generateUnity(Class<? extends MovableEntity> className)

{

newUnits = className.getConstructor(

Position.class,

String.class,

HELBArmy.class

).newInstance(

new Position(

getUnityExitX(),

getUnityExitY()

),

this.getSide(),

gameboard

);

}

*De cette manière, lorsque je souhaite créer une unité, je n’ai qu’à spécifier le nom de la classe en paramètre. En effet, les constructeurs de toutes mes unités demandent les mêmes valeurs. Donc je peux tout centraliser en une demande.*

*Néanmoins, il suffit qu'une unité ait un constructeur différent pour que cette solution devienne obsolète.*

*L’utilisation d’objet « Class<T> » est détaillée dans la partie* [*Analyse*](#_3._Analyse_:)*.*

# 3. Analyse :

Une image contenant texte, capture d’écran, diagramme, Parallèle

Description générée automatiquement

Voici ci-dessus, une représentation simpliste de l’architecture de mon application. Celle-ci est sous forme de diagramme de classe. J’ai volontairement tronqué des informations superflues pour discuter en premier temps de l’architecture global.

**HELBArmy** a été développé en suivant une architecture **MVC**. C’est-à-dire que les classes sont séparées en 3 parties.

Le contrôleur, *HELBArmy*, reprend tout ce qui est relatif à la gestion centrale du jeu. Notamment, l’initialisation de paramètres principaux (ex. : taux de rafraichissement du jeu), les raccourcis claviers, la génération des entités statiques, la destruction d’entités ou encore une gestion minime des positions.

La vue, *View*, reprend tout ce qui est relatif à l’affichage graphique du jeu. Elle gère la taille en hauteur et en largeur de la fenêtre et dessine le champ de bataille avec ses entités.

Les modèles, ici représentés par les classes héritant de *Entity*. Ces classes regroupent tout ce qui est relatif aux objets du jeu, qu’ils soient statiques ou dynamiques.

Les classes statiques et la classe *Position* occupent une place particulière dans mon architecture. Elles ne font pas partie des concepts principaux du modèle **MVC**, mais elles ont des rôles essentiels en fournissant des fonctionnalités utilitaires aux autres classes.

La classe *Position* est une classe simple qui me permet de simplifier la gestion des positions. Cette classe centralise des données utiles aux positions. Dans ce cas, la coordonnée en axe x et en axe y. Elle me permet aussi de comparer deux positions directement, en ayant redéfini la méthode et .

Les classes statiques *HorsemenManager* et *PikemenManager* gèrent un comportement bien spécifique de leurs unités respectives.

* *HorsemenManager* gère la distance de sécurité des cavaliers par équipes.
* *PikemenManager* gère la vision partagée des piquiers par équipes.

J’ai décidé de séparer ces comportements dans des classes, car elles occupent un grand nombre d’espaces en ligne de code. De plus, un élément qui est important à noter, c’est que les unités d’une même équipe ne sont pas censées gérer le comportement d’une autre équipe. Ce sont principalement ces raisons qui m’ont fait pencher vers la création de classe spécifique pour ce comportement.

La classe *Board*, quant à elle gère tout ce qui est relatif aux positions sur le plateau, qui n’implique pas d’entités. Comme le calcul de distance entre deux positions.

Les classes du diagramme sont disponibles [en annexe](#_7._Annexe_:).

J’aimerai aussi discuter de l’utilisation d’objet « *Class<T>* » dans mes classes. J’utilise cet objet pour 2 raisons principales.

Premièrement, cela me permet de mettre une condition qui génèrera une erreur dès la compilation du programme. En effet, dans les méthodes, je précise que l’objet passé en paramètre doit hériter de la classe *MovableEntity*. Voici un exemple de code qui génère une erreur à la compilation :

// MovableEntity.java

// called by units that have a multiplicator attack on constructor of caller

protected void setAttackMultiplicator(HashMap<Class<? extends MovableEntity>, Double> attackMultiplicatorMap)

{

this.attackMultiplicatorMap = attackMultiplicatorMap;

}

setAttackMultiplicator(new HashMap<>() {{

// Generate an error because Position does not extend MovableEntity

put(Position.class, 3.0);

}});

// Corrected code called by Pikemen Constructor

setAttackMultiplicator(new HashMap<>() {{

put(Horsemen.class, 3.0);

}});

Enfin, cela me permet de générer des instances de classes différentes, mais avec un constructeur similaire. Lors de la création d’unité par les villes, j’utilise ce genre de procédé.

// to know what movable entity will be generate

Class<? extends MovableEntity> classNameNewUnits;

// get constructor with param (Position, String, HELBArmy)

Constructor<? extends MovableEntity> constructorOfNewUnits = classNameNewUnits.getConstructor(Position.class, String.class, HELBArmy.class);

// create instance with precedent constructor

MovableEntity newUnits = constructorOfNewUnits.newInstance(

new Position(getUnityExitX(), getUnityExitY()),

this.getSide(),

gameBoard

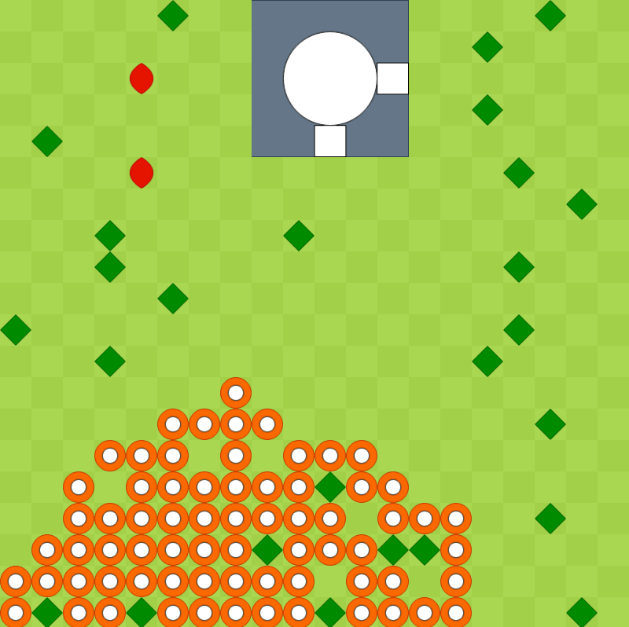
);

# 4. Limitations et développement futur :

Pour garantir une expérience de jeu optimale, certaines contraintes techniques doivent être respectées. L'une des principales limitations concerne les dimensions du plateau. Elles doivent être comprises entre **20 x 20** et **100 x 100**. À noter que le nombre de colonnes doit être égal au nombre de ligne, sinon le rendu ne sera pas parfait. Certaines ligne ou colonne peuvent ne pas être affichées. Ce « *pool* » de dimensions garantit que le jeu reste fluide tout en évitant les bugs. Un plateau de taille inférieure à **20 x 20** rendrait le jeu injouable, tandis qu'un plateau supérieur à **100 x 100** prendrait beaucoup trop de ressource matérielle pour garantir un jeu fluide.

De plus, le taux de rafraichissement doit être ajusté en fonction de la taille du plateau. Pour un plateau de petites dimensions, un taux de rafraichissement élevé, comme 10 images par seconde, est possible. À l'inverse, pour un plateau à grande dimension, il est conseillé de réduire ce taux de rafraichissement afin d'éviter des bugs et l’incapacité de l’ordinateur à faire les calculs nécessaires en peu de temps.

Une image contenant capture d’écran, cercle, clipart, Graphique

Description générée automatiquement 

**20 x 30**, la ville Sud n’est pas affichée et les collecteurs Nord cherche un drapeau qui n’est pas accessible sur la carte.

.

**10 x 10**, les villes sont collées l’une à l’autre, la simulation n’avance jamais.

Le nombre d’arbres a aussi beaucoup d’importance sur le plateau. Avec un taux d’apparition des arbres de 0%, les collecteurs ne pourront pas récolter du bois pour générer d’autres unités, ce qui rendrait la simulation remplie uniquement de collecteurs. De même, avec un nombre d’arbres trop important, les unités de combat ne pourront pas aller bien loin.

Un point qui est compliqué à aborder, c’est la gestion des positions. En principe, il ne peut pas y avoir plus d’une entité sur la même position. En réalité, c’est plus complexe que ça.

Que faire lorsqu’une unité n’a pas d’action à faire ? Ça peut être le cas d’un collecteur lorsqu’il n’y a pas d’arbres sur le plateau ou alors le cas d’un cavalier qui n’a aucun allié cavalier ou déserteur ennemi. Dans ces cas précis, j’ai décidé délibérément de ne pas les faire bouger et par conséquent ils occuperont une même position pendant plus d’un tour. En temps normal, ce n’est pas un problème, car les piquiers ont un comportement similaire. Mais là où il y a un problème, c’est si une unité occupe la même position et que cette position est la position de l’entrée des unités de son équipe.

Dans ce cas précis, les unités bloquent l’entrée d’unité de leur ville. Lorsque la ville fera apparaitre une nouvelle unité, deux unités partageront la même position pendant 1 tour minimum.

Pour régler ce problème, une solution peut être envisagée. Au lieu de stopper les unités quand ils n’ont rien à exécuter, on peut les faire déplacer à une position aléatoire et continuer jusqu’à ce que leur comportement principal puisse être exécuté.

Enfin, un point duquel je voudrais discuter, c’est la présence des déserteurs. À force d’effectuer des simulations, j’ai remarqué que le comportement des déserteurs n’est pas complet. Pour ajouter un réel impacte à la simulation, les déserteurs ne devraient pas craindre les piquiers. En effet, il suffit d’un seul piquier au milieu du champ de bataille pour empêcher tous les déserteurs adverses d’avoir un impact. De plus, les seules unités réellement capables de tuer un piquier sont les piquiers eux-mêmes. Le multiplicateur de dégâts des piquiers envers les cavaliers est si grand qu’il faudrait seulement 5 coups pour tuer un cavalier, contre 18 coups d’un cavalier pour tuer un piquier.

De plus, je n’ai pas pris en compte les bonus de point de vie des drapeaux. Car oui, plus une unité a de points de vie, plus le drapeau lui en rapportera.

# 5. Conclusion :

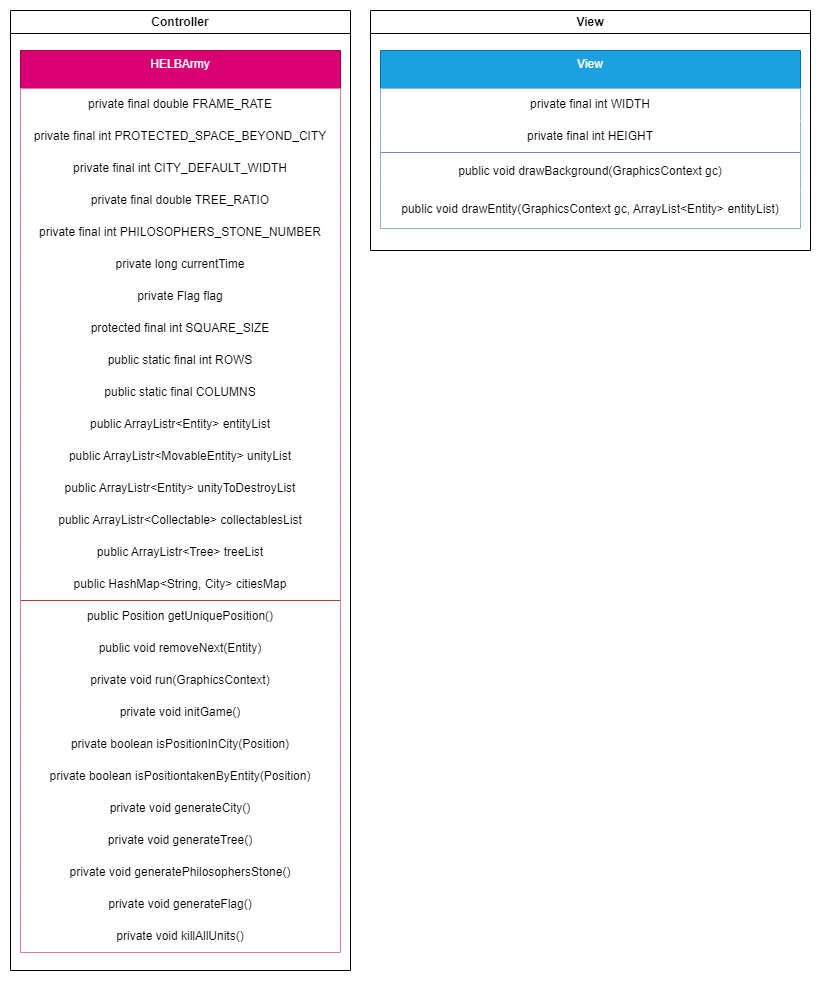
Ce projet a été une expérience incroyablement enrichissante. Il s'agit de ma seconde application concrète développée avec Java et la technologie JavaFX, et je suis fier d’avoir pu implémenter toutes les fonctionnalités demandées. La première étant Erase & Defend, un projet de groupe réalisé lors du Hackaton organisé par l’entreprise Odoo. Travailler sur ce projet m’a permis de découvrir et de mettre en pratique l’architecture MVC, qui m’a aidé à structurer mon code de manière claire et efficace. Je suis convaincu que ces apprentissages me seront précieux à l’avenir.

Ce projet m’a donné envie de continuer à l’améliorer quand j’aurai du temps libre, pour y ajouter des fonctionnalités supplémentaires et corriger certains comportements comme énoncé dans la partie [Limitations et développement futur](#_4._Limitations_et).

Dans l’ensemble, je suis très satisfait du résultat obtenu et de tout ce que j’ai appris en réalisant cette application.

# 6. Annexe :

Classe présente dans le diagramme de classe de la section [Analyse](#_3._Analyse_:). Attention, seulement certains attributs et méthodes sont présents. Les éléments n’ayant pas d’utilité de compréhension ont été omis.

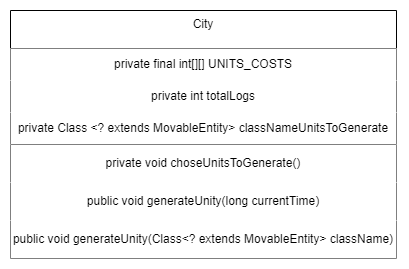
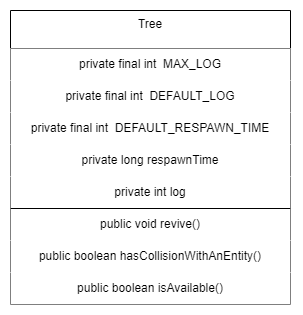


Une image contenant texte, capture d’écran, Police, nombre

Description générée automatiquementUne image contenant texte, capture d’écran, Police, document

Description générée automatiquement

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

Description générée automatiquementUne image contenant texte, Police, capture d’écran, ligne

Description générée automatiquementUne image contenant texte, capture d’écran, Police, ligne

Description générée automatiquementUne image contenant texte, capture d’écran, Police, nombre

Description générée automatiquement

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

Description générée automatiquement