Une image contenant texte

Description générée automatiquement

**RAPPORT DE PROJET  
PROGRAMMATION CONCURRENTE ET INTERFACE INTERACTIVES**

YILMAZ Mikail   
&  
 JAROSSAY Thomas

&

MSILNI Yassine

&

TAGUENGAYTE Lina

Janvier 2022 à Avril 2022

**Référent de matière :** Monsieur SABOURET <[nicolas.sabouret@ université-paris-saclay.fr](mailto:nicolas.sabouret@ université-paris-saclay.fr)>  
**Chargée de TD :** Madame NGUYEN <[thi-thuong-huyen.nguyen@](mailto:thi-thuong-huyen.nguyen@université-paris-saclay.fr)[université-paris-saclay.fr](mailto:thi-thuong-huyen.nguyen@université-paris-saclay.fr)>

Table des matières

[Introduction 3](#_Toc100267913)

[Analyse Globale 4](#_Toc100267914)

[Plan de Développement 4](#_Toc100267915)

[Conception Générale 5](#_Toc100267916)

[Conception Détaillée 6](#_Toc100267917)

[Résultats 21](#_Toc100267926)

[Documentation Utilisateur 21](#_Toc100267928)

[Documentation Développeur 22](#_Toc100267929)

[Conclusion et Perspectives 23](#_Toc100267930)

# Introduction

Le projet de PCII a pour but de nous utiliser nos connaissances en programmation objet Java pour réaliser un jeu de survie et de simulation de haut à travers des cases. Nous voulons réaliser un jeu simplifié inspiré du célèbre jeu ***Starcraft*** avec un thème et une version choisie par chacun des groupes, dans notre cas nous avons choisi le jeu ***Age of Empire***. Dans lequel nous avons un plateau de jeu en guise d’interface et de menu avec plusieurs compartiments avec le plateau de jeu, les informations et les ressources. Le but du jeu est de survivre ainsi que de développer son village tout en faisant attention à de potentiel danger extérieur. Pour cela, à l’aide de la souris le joueur déplacera les villageois et décider des actions qu’ils feront. Voici à quoi pourrait ressembler l’interface graphique de notre jeu :

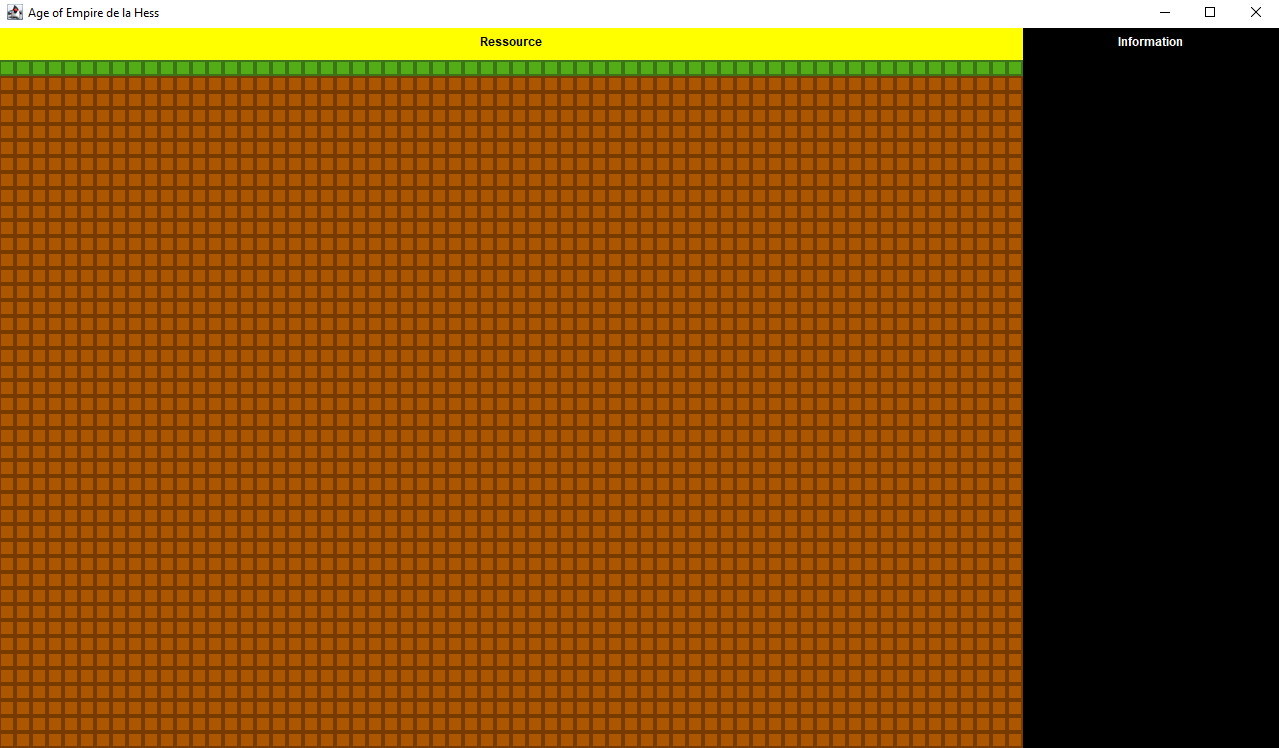


Figure 1 : Premier Rendu du jeu

# Analyse Globale

Nous allons réaliser un projet en *Java*, ce projet devra respecter le *model MVC* comme précédemment vu pendant le tutoriel, nous allons donc répartir nos classes de façon que chacune d’entre respect le *model MVC* c’est-à-dire qu’elles se situe soit dans le contrôleur, la vue ou le modèle. Ce projet devra être réalisé sur une durée de sept semaines. Pour implémenter notre jeu, nous allons devoir implémenter de nombreuses fonctionnalités. Plusieurs interfaces graphiques afin d’afficher notre jeu divisé en 3 parties (Plateau de jeu, Ressource, Information diverses) qui nous servira aussi d’interface avec l’utilisateur. Il faudra également mettre en place une interface faisant le lien entre les clics de la souris par l’utilisateur et notre programme, à l’aide d’un *MouseListener* ainsi qu’une multitude de fonctionnalités plus ou moins nécessaires que nous allons énoncer dans la partie [Plan de développement](#_Plan_de_Développement).

# Plan de Développement

Dans le Plan de Développement nous allons lister toutes les fonctionnalités et sous-fonctionnalités qui vont être développées et implémentées dans le projet. Nous allons suivre la même méthode que celle du tutoriel : diviser notre travail sous deux catégories : les fonctionnalités principales et les sous-fonctionnalités secondaire mais tout de même importante pour avoir un projet plus complet et agréable d’utilisation.

**Fonctionnalités :**

* Affichage de la fenêtre et des différents panels
* Grille de jeu
* Les unités
* Affichage des informations sur l’unité sélectionné
* Affichage des ressources
* Horloge du jeu
* Ciblage des tuiles du jeu
* Tâche des entités
* Déplacement des Unités
* IA des animaux
* Système de récolte des ressources par les villageois
* Système de construction de bâtiment
* Système d’attaque entre les villageois et les animaux
* Améliorations
* Faire finir la partie

# Conception Générale

Ce projet a été réalisé en utilisant le modèle *MVC*. Ce modèle consiste à séparer dans différents packages les classes gérants respectivement la *Vue* (donc la partie affichage), le *modele* (toutes les mécaniques de jeu) et enfin le *Controlleur* (tous les inputs du joueurs). Ce modèle nous permet d’avoir une meilleure lisibilité du projet et de mieux comprendre l’impact de chaque classe sur le projet.

Le package *Modele* contient toutes les classes gérants l’état du jeu, composé des packages suivants :

* *Amelioration* contient les classes : *Amelioration, FermeAm, NiveauAm, StockageVillageoism et VitesseRecolteAm :* l’ensemble des classes gèrent les bonus que le joueur peut débloquer
* La classe *Grille :* Gère la grille du jeu
* La classe timer : Horloge qui met à jour le jeu à temps régulier
* *Tuille*contient les classes : *Eau, EauPoisson, Herbe, Sable, Terre, Tuille*: l’ensemble des classes qui gère le sol de la grille là où les unités se posent
* *Unite*contient les packages Entite et Structure
  + *Entite* qui sont les êtres vivants sont divisés en deux packages également : *Animaux et Villageois*
  + *Structure* qui sont les objets générés naturellement et les constructionssont divisés en deux packages également : *Bâtiment et Environnement*

Le package *Vue* contient toutes les classes gérants la partie affichage du jeu, composé des packages suivants :

* La classe *Ressourcepanel :* Affichage des ressources disponible du joueur
* La classe *jeuPanel*: Affichage du plateau de jeu
* La classe *infoPanel*: Affichage des informations sur l’unité qui a été cliqué dessus par le joueur
* La classe *Vue* : Classe qui rassemble l’ensemble des Panel ci-dessus

Le package Contrôle contient toutes les classes gérants les interactions entre la souris et le jeu :

* La classe *ControleJeu* : Gère les clicks du joueur sur la zone de jeu
* La classe *ControleAmelioration :* Permet de sélectionner les améliorations sur un bâtiment
* La classe *ControleConstruction :* Permet de sélectionner un bâtiment à construire par ***un villageois***

# Conception Détaillée

**Affichage de la fenêtre et des différents panels**

Le JFrame représente notre fenêtre de jeu qui contient les différents JPanel :

* InfoPanel

S’occupant de l’affichage des informations à droite de la fenêtre avec l’affichage des unités sélectionnées et ciblé ainsi que les interactions possibles avec l’HDV comme l’amélioration.

* JeuPanel

S’occupant de l’affichage du jeu en plein milieu du JFrame avec la map et le déroulement du jeu.

* RessourcePanel

S’occupant de l’affichage des ressources du joueur ainsi que le nombre de villageois.

Voici un aperçu :

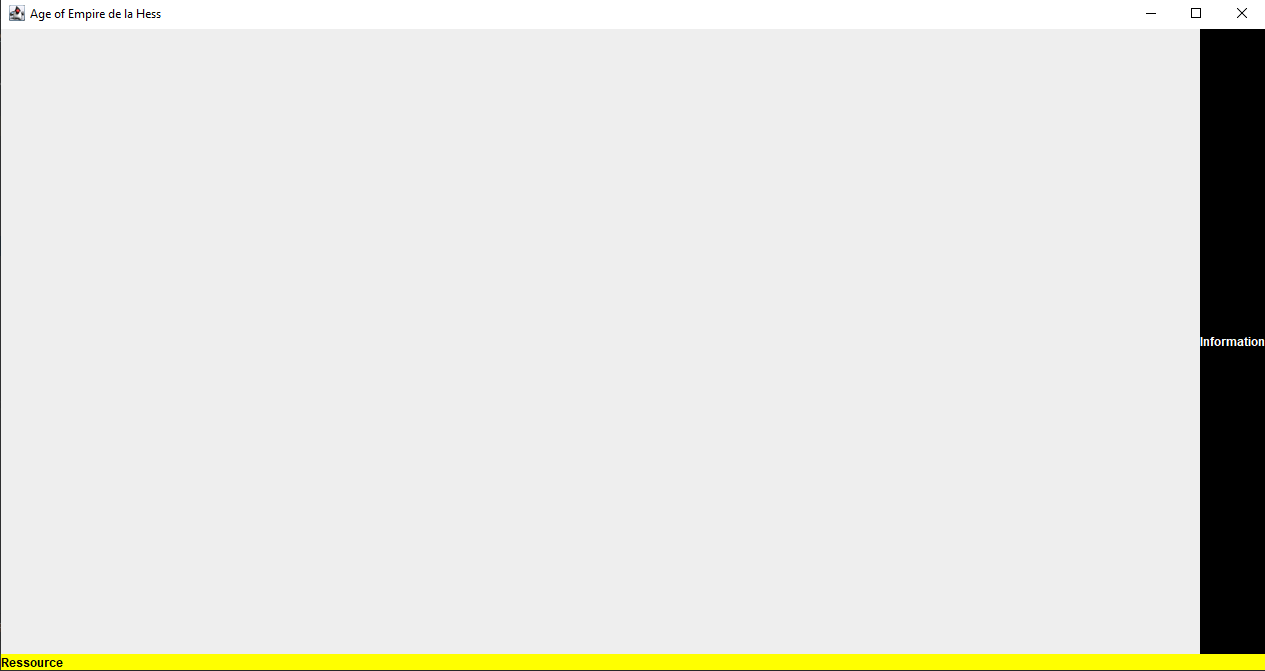


Figure 2 : Séparation des différents Panel

Notre JFrame est créé dans la classe Vue qui utilise aussi trois attributs InfoPanel JeuPanel et RessourcePanel qui font appel aux trois classes citées plus haut pour leurs affichages

Une image contenant texte

Description générée automatiquement

## Grille de jeu

Le zone de jeu est un tableau de tuiles à 2 dimensions où les unités peuvent se poser. Ce tableau est l’attribut tuiles de la classe Grille.

### Tuile

Les tuiles sont définies dans les classes héritant de la classe abstraite Tuile, une tuile est soit solide soit traversable en fonction du boolean solid, si elle est traversable, une unité peut être placé dessus.

Si une unité passe dessus, la tuile devient solide, si l’unité part, elle redevient traversable.

Chaque tuile possède les coordonnées de sa texture dans un tileset dans les attributs x\_texture et y\_texture.

|  |  |  |  |
| --- | --- | --- | --- |
| Herbe | Terre | Sable | Eau |
|  |  |  |  |
| Traversable | Traversable | Traversable | Solide |

### Grille

La taille de la grille est initialisée grâce aux constante HAUTEUR et LARGEUR de la classe Grille et les tuiles sont initialisés dans la méthode construireGrille. (Ça n’est pas aléatoire)

La grille est affichée dans infoPanel en traversant le tableau et en affichant le bon sous rectangle du tileset

On utilise un tileset afin d’avoir très peu d’image chargée en mémoire.

## Une image contenant texte, pièce, graphiques vectoriels Description générée automatiquement **Affichage des informations sur l'unité sélectionnée**

Figure 3 : Rendu de la Grille

On peut sélectionner une unité en faisant un clique gauche dessus, suivant le type de l’unité, différentes informations seront affichés dans le infoPanel :

Toutes les unités ont leurs noms et leurs icônes affichées.

Les entités ont en plus leurs statistiques d’affichées.

Les structures récoltables par les villageois ont le type et la quantité de ressources disponible.

Les animaux nb nourriture

Les villageois ressources récupéré

Si interaction possible. Bouton

Villageois et bâtiment

Une image contenant texte

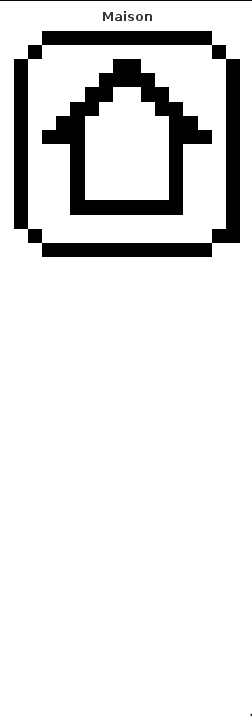
Description générée automatiquement

Figure 4 : Affichage d'une Entité

**Les Unités**

Les unités sont l’ensemble des objets posés sur les tuiles et qui peuvent interagir entre elles.

Arborescence

Une image contenant texte

Description générée automatiquement

Figure 5 : Arborescence du package Unité

Pour la gestion de nos unités de notre jeu, pour l’arborescence nous avons décidé de séparer tous les objets en plusieurs catégories les ***Entités*** et les ***Structures.*** Les ***Entités*** sont composées des ***Villageois*** et des ***Animaux,*** La classe ***Animaux*** est composé du package ***Hostile et Passif***. Le package ***Hostile*** composé de la classe ***Hostile et Loup,*** Le package ***Passif*** composé de la classe ***Passif et Lapin.***

Stockage des Unités dans le Modèle

Nous avons choisi de séparer les Entités et les Structures pour le stockage :

Les Entités sont stockées dans un double tableau et les Unités (Villageois et Animaux) sont stockées dans un ArrayList<Entite> car l’ArrayList est dynamique et nous en avions besoin car ce sont les entités qui se déplacent et effectue d’autre action sur le plateau de jeu.

Toutes les Entités sont affichées via un tailset.png qui constitue l’ensemble des images des entités.



Figure 6 : Image du TailSet

Entités

Les entités sont divisées entre 2 d’une part les villageois et de l’autre les animaux, les entités peuvent se déplacer d’une tuile à l’autre et peuvent effectuer certaine tache. La principale différence entre les villageois et les animaux c’est que les villageois sont contrôlables par le joueur et donc on peut donner des taches à faire.

Liste des taches : Rien, Récolte, Attaque, Construit, Déplace, Déposer, Retour

Villageois :

Les villageois sont définis avec des coordonnées sur le plateau de jeu, des statistiques de vie, attaque et défense. Le villageois peut effectuer toutes les taches possibles. Il peut se déplacer vers un point ciblé, construire une maison, attaquer un animal, récolter des ressources sur les différents environnements ou aller déposer ses ressources dans l’HDV.

Animal :

Les animaux sont divisés en deux parties : les animaux hostiles (les loups) et les animaux passifs (les lapins), tout comme les villageois ils sont définis avec des coordonnées sur le plateau de jeu, des statistiques de vie, attaque et défense tout comme le villageois et lorsqu’un villageois tue un animal le villageois récupère de la nourriture de l’animal. Les animaux hostiles peuvent attaquer une unité si elle est dans son périmètre sinon elle se déplace aléatoirement tandis que l’animal passif se déplace uniquement.

Structure

Les structures sont divisées en deux packages tout comme les Entités, d’un côté les bâtiments et de l’autre l’environnement. Chacun possède des coordonnées pour se repérer sur le plateau de jeu et le type de bâtiment qu’il est.

Bâtiment :

Les bâtiments sont définis avec un booléen s’ils sont en cours de construction (True s’ils sont en construction, False s’ils sont déjà construit), en fonction du bâtiment il a un rôle particulier à jouer (La nourriture pour la ferme, la population max pour les maisons et les ressources pour l’HDV)

Environnement :

L’environnement est divisé est plusieurs classes : Baie, Poisson, Arbre, Rocher

Tout comme le reste des entités ils ont des coordonnées pour se repérer sur le plateau de jeu, un nombre de ressources prédéfini et un type de ressource (Bois,Pierre,Nourriture)

## **Affichage des informations sur l'unité sélectionnée**

On peut sélectionner une unité en faisant un clique gauche dessus, suivant le type de l’unité, différentes informations seront affichés dans infoPanel :

* Toutes les unités ont leurs noms et leurs icônes affichées.
* Les entités ont en plus leurs statistiques d’affichées.
* Les structures récoltables par les villageois ont le type et la quantité de ressources disponible.
* Les animaux ont la quantité de nourriture récoltable sur eux s’ils sont tués.
* Les villageois ont le type et la quantité de ressources stocké sur eux.
* Les villageois ont des boutons pour sélectionner les bâtiments à construire.
* Les bâtiments ont des boutons pour sélectionner les améliorations.

|  |
| --- |
| Une image contenant texte  Description générée automatiquement |

Certaine partie de l’affichage sont mise à jour en temps à certains moment précis :

* Si l’unité sélectionné est une entité, ses statistiques sont mis à jour si elle subit des dégâts
* Si l’unité sélectionné est un villageois :
  + Ses statistiques sont mis à jour si l’amélioration qui augmente la défense est activé.
  + Ses ressources sont mises à jour si elle récolte ou si elle dépose des ressources
  + Les boutons pour sélectionnée un bâtiment à construire sont mis en évidence s’ils sont survolés par la souris
* Si l’unité est un récoltable, ses ressources sont mises à jour si on la récolte
* Si l’unité est un bâtiment :
  + Ses améliorations sont mises en évidence s’il sont survolé par la souris.
  + La barre de progressions des améliorations est constamment mise à jour si elle est lancée.
  + Ses améliorations disparaissent si elles ont fini d’être activé.

**Horloge du jeu**

Pour faire avancer le jeu, on utilise une horloge qui va mettre à jour le jeu à intervalle de temps régulier :

A chaque « tick » d’horloge, le jeu est actualisé dans cette ordre :

Mise à jour des entités

Le développement des améliorations consommés

La baisse de la nourriture en fonction du nombre de villageois

La vitesse de l’horloge est calculée en fonction du tickrate : c’est le nombre de tick qui se produise en une seconde.

Algorithme de l’horloge :

dureTick = tikrate/60\*1000

t = 0

Tant que le jeu tourne

t <- t + temps écoulé

Tant que t >= dureTick

t -= dureeTick

Mettre à jour le jeu

**Ciblage des tuiles du jeu**

Notre classe ControleJeu dans le package contrôle contient le MouseListener MouseClicked(MouseEvent e) qui nous permet de récupérer l’unité qui est sur la tuile sur laquelle on clique gauche dans le JeuPanel, après cela la méthode select de Modele est appelée qui permet d’affecter l’unité aux coordonnées cliquées à uniteSelectionee, ensuite si l’uniteSelectionee est un Villageois alors un clic droit sur une autre tuile permet l’appel de la méthode cible de Villageois permettant l’exécution des différentes taches possibles du villageois.

**Taches des entités**

Nous avons une énumération Tache.java qui contient toutes les actions des différentes entités du jeu, l’entité que nous contrôlons étant le villageois. Si la Tache définis en utilisant le MouseListener est ATTAQUER, alors le villageois va attaquer l’unité ciblé seulement si elle est à côté d’elle, sinon elle se déplace vers la cible sinon, le villageois utilise la méthode deplacer qui lui permet de se déplacer vers l’unité ciblé pour ensuite l’’attaquer, après avoir tué la cible, le villageois repasse en tache = RIEN.

Si nous avons un récoltable, la tâche sera RECOLTER, encore une fois s'il est à coté il récolte, lorsque son inventaire est plein alors il passe en tache DEPLACER puis va à l’HDV ou il passe en tache DEPOSER pour déposer les ressources, ensuite il repasse en tache DEPLACER et repart vers le récoltable qu’il récolter puis repasse en tache RECOLTER et ainsi de suite.

Nous avons aussi la tache CONSTRUIT qui permet grâce aux conditions dans le MouseListener de construire des bâtiments sur les tuiles ciblées si elles sont vides, le principe est le même si le villageois est à coté alors il construit sinon il passe en tache DEPLACER pour y aller puis passe en mode CONSTRUIT pour construire le bâtiment.

## Déplacement des Entités

Le déplacement des entités est géré par deux fonctions : la méthode deplacer et la méthode calculerChemin

### Deplacer

La méthode Deplacer permet de bouger l’unité d’une case dans une direction, elle prend en paramètre une valeur de l’enum Direction.

Avant de faire un déplacement, on vérifie si la case est dans les limites de la grille, puis si elle n’est pas solide.

Si le déplacement est solide, on modifie les attributs x et y de l’unité pour correspondre à la nouvelle case et dans le tableau d’unité unites dans l’instance classe Modele, on fait pointer la nouvelle case par l’unité déplacé, et on met l’ancienne à null.

On met aussi à jour les tuiles de la grille, la tuile où se trouvait l’unité devient traversable alors que celle où est allée l’unité deviens solide.

### CalculerChemin

La méthode calculerChemin est l’implémentation de l’algorithme A\* :  
<https://fr.wikipedia.org/wiki/Algorithme_A>\*

Cet algorithme permet de calculer un chemin entre deux nœuds d’un graphe, on a donc utilisé notre grille comme un graphe où chaque tuile représentait un nœud du graphe.

On a créé une classe privée Nœud uniquement accessible par la classe Entite pour l’utiliser dans la méthode calculerChemin.

Une image contenant texte

Description générée automatiquementLe principe de A\* est de choisir à chaque itération le nœud qui semble le plus rentable grâce à son heuristique puis d’estimer leurs coût heuristique de tous les nœuds voisins à ce nœud jusqu’à tomber sur le nœud d’arrivé.

Explication :

On créer une file prioritaire openList qui trie les nœuds grâce à l’attribut heuristique et une liste closedList qui stock tous les nœuds déjà vérifié.

On met le nœud initial dans openList puis à chaque itération on prend le premier nœud de openList. Pour chaque nœud voisin traversable à ce nœud on calcule le coût pour y accéder et son coût heuristique et on l’ajoute à openList s’il n’était pas déjà dans closedList et s’il n’était pas dans openList avec un coût heuristique inférieur (pour calculer le meilleur chemin possible passant par un nœud), puis on met le nœud initial dans la closedList car on a fini de l’évaluer.

On répète cette étape jusqu’à évaluer le nœud d’arrivée ou jusqu’à ce qu’il n’y ait plus de nœuds dans openList.

Si on tombe sur le nœud d’arrivé. On reconstruit le chemin grầce à la méthode reconstruireChemin :Cette méthode remonte les nœuds grâce à l’attribut pred des nœuds et remplit l’attribut chemin qui est une pile de direction représentant le chemin en cours de l’entité.

Cette pile est dépilée à chaque mise à jour du jeu quand une entité dois se déplacer.

Le chemin est recalculé dans 3 cas :

* Si l’entité veut attaquer une autre entité (on recalcule en permanence car la cible peut bouger).
* Si l’entité avait un chemin déjà prévu mais un déplacement n’est pas possible (par exemple si une case est devenue solide entre temps)
* Si l’entité change de tache.

**IA des animaux**

Nous avons 2 IAS, tout d’abord l’IA commune à tous les animaux qui permet leurs déplacements aléatoires dans une zone limitée :

Si la tâche d’un animal n'est RIEN alors il se déplace aléatoirement avec une chance 10% de se déplacer dans une direction aléatoire dans un rayon maximal de 4 tuiles à partir de ses coordonnées de spawn (1 chance sur 4 pour chaque direction) de la sorte :

Une image contenant texte

Description générée automatiquement

Nous avons ensuite l’IA des animaux hostiles qui comporte les déplacements aléatoires de la première IA de la manière suivant : S'il ne trouve pas de cible (donc un villageois ou un autre animal dans un rayon de 3 tuiles) alors il fait les déplacements aléatoires via deplacementAleatoire(), s'il trouve une cible alors il passe en tache ATTAQUER qui attaque la cible trouvée s'il est à coté ou sinon il se déplace vers la cible puis passe en tache ATTAQUER, si l’animal hostile suit sa cible à plus de 10 tuiles à partir de sa tuiles de focus de départ, alors sa tache passe en RETOUR qui permet à l’animal hostile de retourner directement à la tuile où il a spawn.

**Système de récolte**

Le système de récolte est principalement géré par la méthode *void récolte(Recoltable cible)*, en effet celle-ci s’occupe de récolter les ressources présent dans les différents environnement dans les différent cas. Pour cela on a besoin d’une méthode *enlever* qui retire des ressources à l’environnement où le villageois cible.

Algorithme *récolte*:

Si cible null

Ne rien faire

Si Les ressources du joueur = ressource de la cible

Si ressource joueur au max

Aller deposer les ressources a l’HDV

Si reccource joueur non-max

Si cible à côté

Recolte ressource de la cible

Sinon

Aller vers la cible

SI Les ressources du joueur différent de ressource de l’environnement

Si cible à côté

Ressource joueur = 0

Ressource joueur = ressource cible

Recolte ressource de la cible

Système de construction

Le système de construction appelé dans la classe *Villageois* permet de continuer la construction d’un bâtiment où la variable batiment.enConstruction = True. Lorsque cette méthode est appelé celle-ci fait appel à la méthode *batiment.seConstruire()* qui augmente *batiment.tickActuel* a chaque appel. Le bâtiment sera en cours de construction tant que *batiment.tickActuel* n’aura pas atteint la valeurde *batiment.tickRequis.* Lorsque le villageois est à côté du bâtiment en construction il construit et la valeur de *batiment.tickActuel* augmente jusqu’à que le bâtiment soit construit. Il y’a ensuite le second cas où le villageois dois créer le bâtiment qui n’est pas encore en construction, Pour cela dans la classe *ControleJeu* on selectionne un villageois et on clique sur un type de bâtiment à construire pour passer en mode construction, ceci est définis dans la classe ControleConstruction. Enfin dans la classe ControleJeu on passe en mode Construction : on clique gauche sur une case qui n’est pas solide (où il n'y a aucune unité) pour poser le bâtiment à construire et retire les ressources nécessaires pour la construction du bâtiment.

Algorithme de *Villageois.Construire(Batiment batiment)* :

Si bâtiment est null

Ne rien faire

Si bâtiment à côté et est enConstruction

Tache du villageois = construire

SeConstruire()

Sinon Calculer le chemin jusqu’au bâtiment

Système D’attaque

Le système d’attaque est unique aux unités car les structures ne peuvent attaquer. La méthode définis dans la classe Unité permet de cibler une unité et de l’attaquer. Cette méthode sous entends donc l’implémentation de deux autres méthodes nécessaires pour celle-ci. La méthode *subirDegat(int dégât)* qui retire le nombre point de vie en fonction la défense et la valeur de dégât. Puis il y’a la méthode *mourir()* qui une fois une entité arrivé à 0 point de vie meurs et disparaît du plateau de jeu. La méthode *attaquer(Entite cible)* est une méthode qui une fois appelé si la cible passer en paramètre si elle est null l’entité ne fait rien, si la cible est a déjà ses point de vie à 0 l’entité ne fait rien également. Si la cible est à côté alors celle-ci attaque en faisant un appel à la méthode de *surbirDégât*. Mais si la cible n’est pas à côté alors celle-ci se déplace jusqu’à celle-ci.

Algorithme de *Attaquer(Entite cible)* :

Si cible = nul ou pv de la cible = 0

Ne rien faire

Si la cible est à côté

Attaquer

Sinon

Se déplacer vers la cible

## Amélioration

Le joueur a la possibilité de développer des améliorations contre des ressources pour obtenir des bonus, comme par exemple pour augmenter la vitesse de récolte de ses villageois.

Les améliorations qu'il peut prendre dépendent du niveau du joueur, et éventuellement d'une amélioration à déjà avoir récupérer avant.  
Elle sont disponible sur les bâtiments dans le l’affichage à droite. Elles sont sélectionnables si le joueur à assez de ressources et assez de villageois. Une fois sélectionné, les ressources sont diminuées et une barre de progression commence, quand la barre de progression est terminée, l’amélioration est activée.

Voici toutes les améliorations :

Une image contenant texte, reçu

Description générée automatiquement

Figure 7 : Différentes améliorations

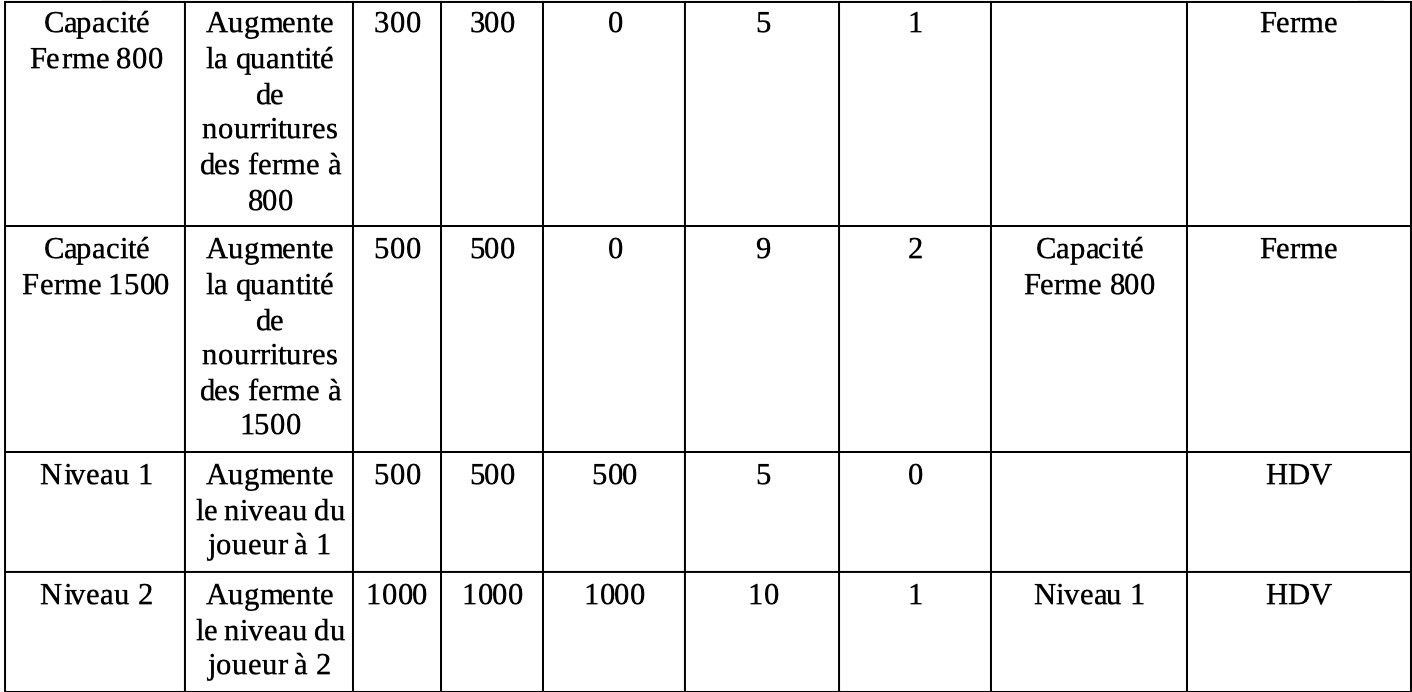


Figure 8 : Amélioration de la ferme et du Joueur

La création de villageois se fait par une amélioration qui ne peut pas être définitivement activé (pour pouvoir en faire plusieurs). Le villageois est créé à côté de l’HDV.



Figure 9 : Ressource pour création d'un villageois

Faire finir la partie

Pour que la partie puisse avoir une fin nous devons implémenter une méthode qui fait finir la partie dans la classe *Modele*, pour cela on choisit nos conditions de défaite : Ne plus avoir de villageois et ne plus avoir assez de ressource pour avoir de nouveau villageois. Pour cela nous avons définis une méthode qui diminue la nourriture à chaque tick de l’horloge appelé *ReduireNourriture()* . Enfin la méthode *testPerdu()* est testé à chaque tick de l’horloge si la population est égal à 0 et qu’il n’y a plus assez de ressource pour crée de nouveau villageois. Une fois la partie perdu la méthode *affichageFinPartie()* affiche une fenêtre avec un message de fin de partie définis dans la classe *Vue*

# Résultats

# Une image contenant carte Description générée automatiquement

# Documentation Utilisateur

Pour que l’utilisateur puissent jouer à « Age of Empire de la Hess » plusieurs prérequis sont nécessaire : En premier lieu un IDE est nécessaire de préférence mais il est tout de même possible de lancer le jeu tout même uniquement avec Java si on exporte le fichier en .jar en exécutable. Une fois dans l’IDE, l’utilisateur dois importer le fichier, aller dans la classe Main puis d’exécuter le fichier et ainsi le jeu se lancera et aura juste à cliquer avec sa souris ou le pavé tactile. Dans le cas où l’utilisateur n’utilise pas d’IDE, il aura juste à lancer le fichier exécutable et le jeu sa lancera et comme pour l’IDE, l’utilisateur aura juste à cliquer avec sa souris ou pavé tactile dans la fenêtre affichée. Pour rappel que l’utilisateur est un IDE ou non il devra avoir une version de Java 1.8 ou supérieur pour pouvoir exécuter le code.

# 

# Documentation Développeur

Le jeu est assez simple à comprendre mais il possède malgré tout quelques complexités. Les packages contenant les classes les plus importantes sont :

* *Unite* qui contient l’ensemble des entités (Villageois et animaux) et des structures (Bâtiments et Environnement)
* *Panel* qui contient l’ensemble des affichages de fenêtres
* *Controle* qui contient l’ensemble des améliorations et le contrôle du Jeu
* *Tuile* qui contient l’ensemble des tuiles qui sert de base pour l’affichage
* La classe Modele

Il y’a également plusieurs constantes qui peuvent jouer un rôle important pour le fonctionnement du jeu :

* maxPopulation : le maximum de villageois possible
* population : Le nombre de villageois présent sur le plateau de jeu
* bois : le nombre de bois détenu par le joueur
* pierre : le nombre de pierre détenu par le joueur
* nourriture : le nombre de pierre détenu par le joueur

Il y a de nombreuses fonctionnalités que nous n’avons pas pu implémenter par faute de temps mais que nous implémenterons sans doute lors de la prochaine mise à jour de notre projet, parmi ces fonctionnalités on retrouve un système d’animation en temps réel lorsque les unités se déplacent ou un environnement plus complexe comme avec l’ajout d’outils et donc de bâtiments permettant la création de ces outils.

# 

# Conclusion et Perspectives

Nous voilà au terme de ce projet de PCII dans laquelle l’implémentation de nombreuses fonctionnalités malgré ces efforts il nous reste encore beaucoup de chose à faire que ça soit niveau code pour l’optimiser ou l’ajout de nouvelles fonctionnalités pour une meilleur expérience utilisateur qui nous permettra de rendre ce jeu plus complet. Les difficultés rencontrées m’ont permis de comprendre nos mauvaises habitudes sur le manque de commentaires sur le code car nous sommes plusieurs à lire le même code ce qui freine beaucoup la compréhension du code des autres sans commentaires et m’ont permis d’apprendre l’utilisation de méthode sur les bibliothèques jamais utilisée. L’algorithme de calcul de chemin qui est très complexe nous a également fait comprendre l’importance des mathématiques en général au sein des projets de programmation. Ce projet nous a permis d’apprendre à programmer à plusieurs tout en respectant une méthode de programmation qui est commune à tous qui dans notre cas est le modèle MVC. Le fait de devoir séparer la vue, du modèle et de l’affichage à quelque peu changé notre façon de voir l’ensemble des potentiels classes à créer en début de projet, de programmer de manière que chacun puisse travailler de son côté sans déranger les autres, nous en ressortons une expérience non négligeable qui nous aidera sans l’ombre d’un doute dans notre avenir. Toutefois, il reste encore des fonctionnalités à implémenter sur notre jeu comme par exemple l’ajout de difficulté qui permettrait qu’il y’est plus d’animaux hostiles et moins de ressources disponibles, une animation plus agréable qui rendrait le jeu plus fluide ou encore des fonctionnalités agréables pour l’utilisateur comme relancer une partie sans à avoir rerun les fichiers à chaque partie.