RAPPORT
PROJET DE C++

Alexis Proust
Jordane Minet
ET4 Info

Polytech Paris-Sud - Janvier 2019

Sommaire

Compilation et lancement d'une partie

Pour compiler notre projet, un makefile est mis à disposition avec les sources du jeu. Il suffit donc de lancer la commande "make" dans le dossier contenant le makefile et les sources du jeu. Notre projet utilisant des fonctionnalités de la version 2017 de C++, il est par conséquent important et conseillé d'utiliser cette version pour compiler le programme. L'exécutable créé se nommera "game".

Pour **lancer le jeu**, il vaut faut exécuter la commande "./game" dans la console, en étant toujours présent dans le dossier contenant le projet. Une fois le jeu lancé, vous aurez la possibilité de choisir un mode de jeu parmi ceux disponibles, ou bien de charger une partie sauvegardée (option proposée seulement si une partie sauvegardée est détectée par le jeu). Le fait de gérer les choix dans un menu, et non par des paramètres passés au lancement du programme, permet de limiter les erreurs pouvant être commises par l'utilisateur. Pour une meilleure expérience de jeu, nous recommandons de mettre la console de plein écran, l'affichage du plateau étant assez grand.

```
~/Travail/Polytech/S7/CPP/Projet/ make
rm -rf ./src/ofiles/ game
mkdir -p ./src/ofiles/
g++ -std=c++17 -Wall -Wextra -Werror -c src/cpp/archer.cpp -o src/ofiles/archer.o
g++ -std=c++17 -Wall -Wextra -Werror -c src/cpp/soldat.cpp -o src/ofiles/soldat.o
g++ -std=c++17 -Wall -Wextra -Werror -c src/cpp/superSoldat.cpp -o src/ofiles/superSoldat.o
g++ -std=c++17 -Wall -Wextra -Werror -c src/cpp/unite.cpp -o src/ofiles/unite.o
g++ -std=c++17 -Wall -Wextra -Werror -c src/cpp/joueur.cpp -o src/ofiles/joueur.o
g++ -std=c++17 -Wall -Wextra -Werror -c src/cpp/catapulte.cpp -o src/ofiles/catapulte.o
g++ -std=c++17 -Wall -Wextra -Werror -c src/cpp/main.cpp -o src/ofiles/main.o
g++ -std=c++17 -Wall -Wextra -Werror -c src/cpp/plateau.cpp -o src/ofiles/plateau.o
g++ -o game src/ofiles/archer.o src/ofiles/soldat.o src/ofiles/superSoldat.o src/ofiles/unite
```

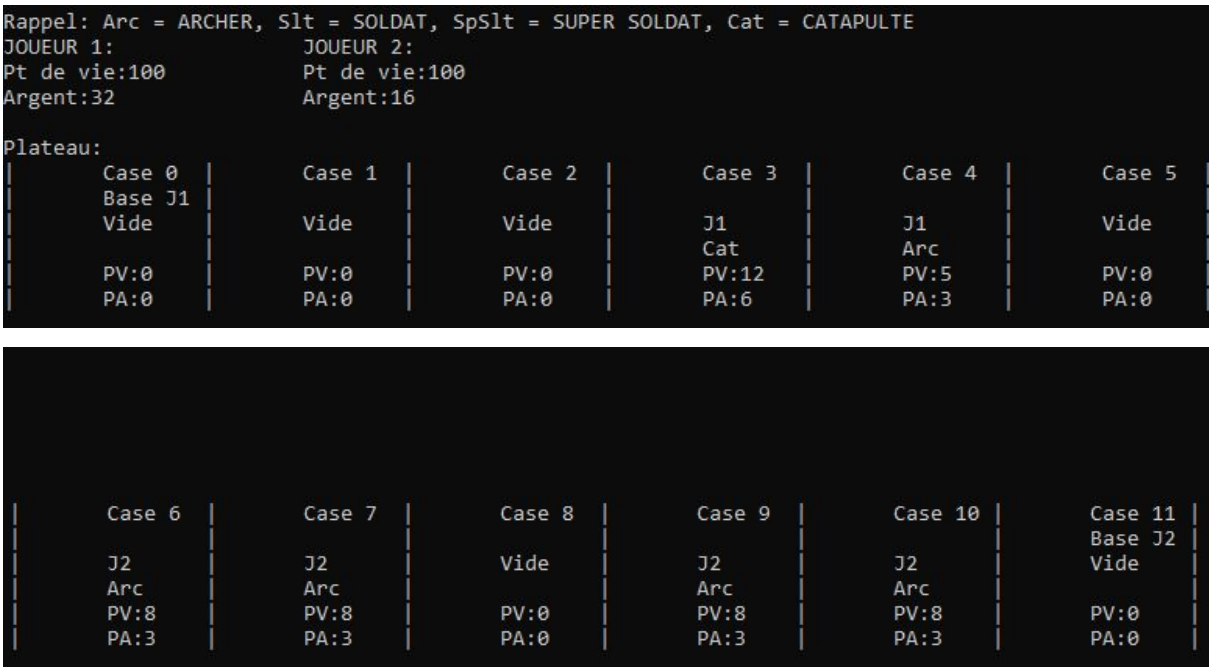
Compilation avec Makefile

```
~/Travail/Polytech/S7/CPP/Projet/ ./game
A quel mode de jeu souhaitez-vous jouer ? :
1 - Joueur contre Joueur
2 - Joueur contre IA
3 - Reprendre la partie enregistree
Votre choix (numero):
```

Lancement du jeu

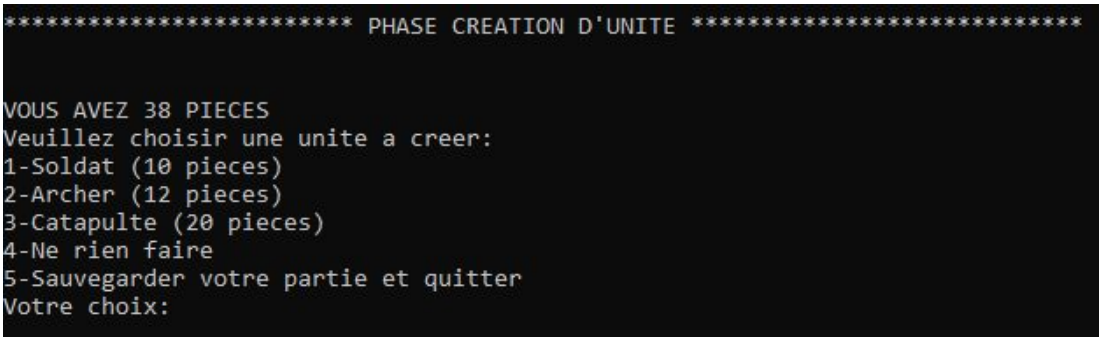
Description du jeu / Fonctionnalités

Nous avons développé ce jeu en suivant les consignes du projet. Ainsi nous avons un jeu de plateau dans le style “Age Of War”. De chaque côté de ce plateau, il se situe la base d’un joueur. Cette base a 100 points de vie au début de la partie. Une partie prend fin quand la base d’un des deux joueurs a été détruite ou que la limite de 100 tours de jeu est atteinte.

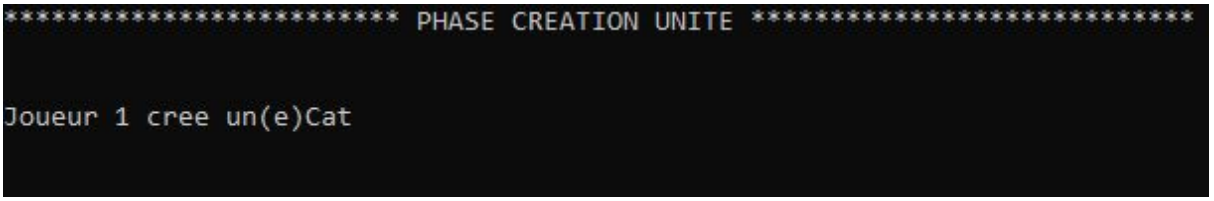


Représentation du plateau de jeu

Chaque joueur a la possibilité de créer des unités, apparaissant dans la base du joueur à la création, à condition d’avoir l’or nécessaire et une base vide de toute unité. Il y a en tout 3 unités pouvant être créées à la demande : Soldat (fantassin), Archer ou bien Catapulte. Les soldats ont la possibilité d'évoluer en un quatrième type de personnages : le Super-Soldat. Pour cela, l’unité soldat devra tuer un personnage adverse, la faisant automatiquement évoluer dans sa forme de Super-Soldat. Chaque unité a des actions différentes qui seront effectuées à chaque tour de jeu, ainsi qu’un coût de création.



Actions possibles du joueur



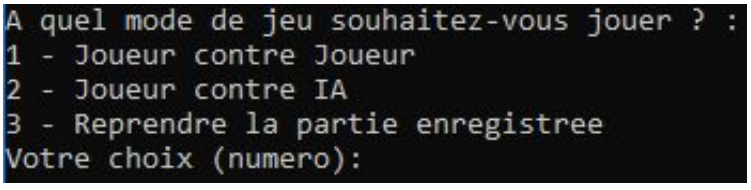
Message de création d'une unité

Il y a deux façons de gagner de l'argent pendant une partie :

- En début de tour : Chaque joueur reçoit 8 pièces d'or
- Lors de la mort d'une unité ennemie : Chaque unité tuée rapporte la moitié de son prix d'achat au joueur propriétaire du personnage tueur.

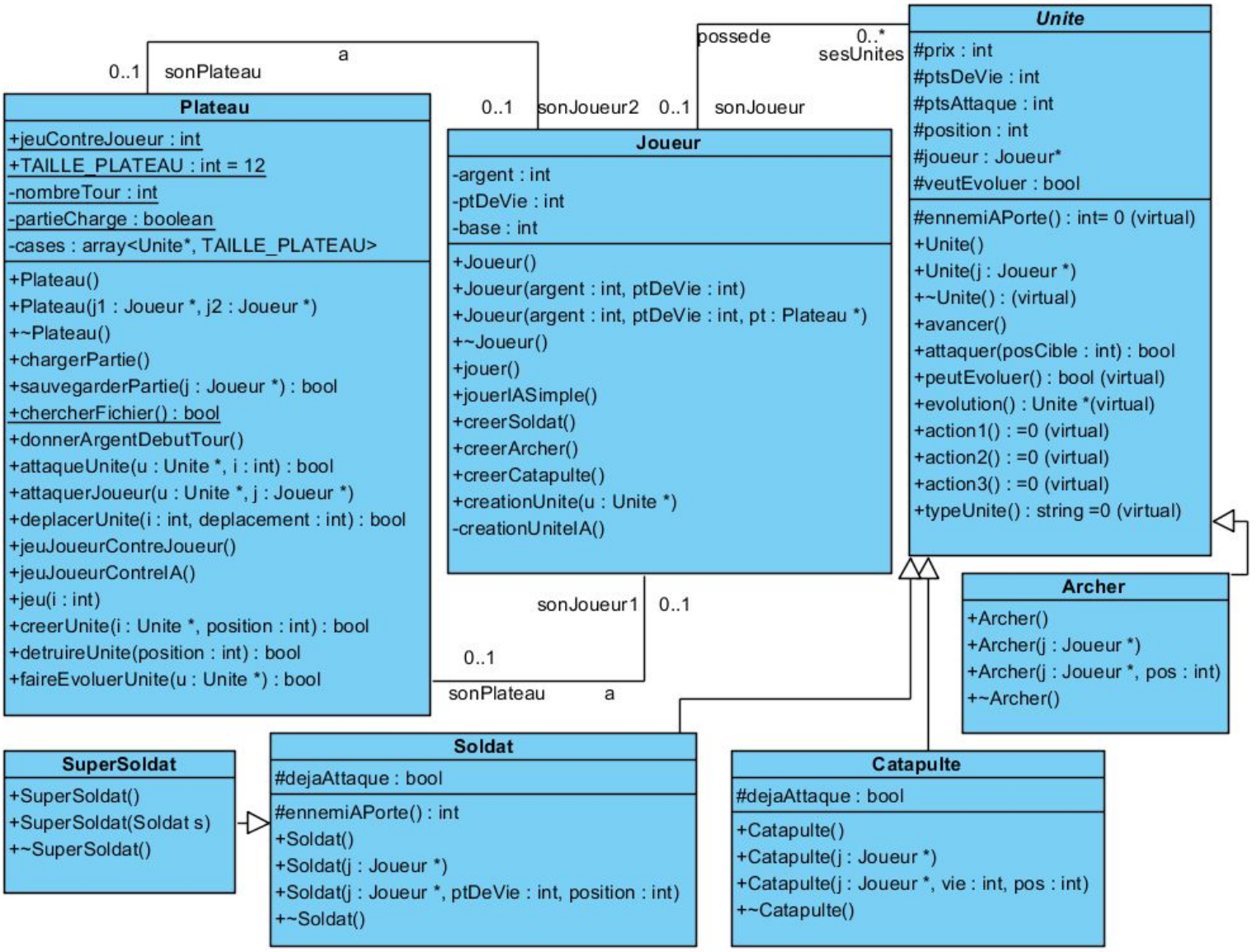
Nous avons mis en place deux modes de jeu pour ce projet. Le premier mode a pour but de se faire affronter 2 joueurs réels, l'un contre l'autre, pendant que le second mode de jeu permettra à une personne de jouer contre une IA. Cette IA est très basique. Elle est conçue pour créer à chaque tour l'unité la plus chère disponible pour son budget.

Un système de sauvegarde a été mis en place, permettant de quitter une partie en cours pour la reprendre plus tard en l'état. Vous pouvez charger la partie sauvegardée dans le menu principal du jeu, lors du choix d'un mode de jeu. Pour sauvegarder, il suffit lors de votre tour de choisir l'option "Sauvegarder et quitter". Attention, vous ne pouvez avoir qu'une seule partie sauvegardée à la fois !
Pour la sauvegarde, nous enregistrons dans un dossier "Ressource" un fichier texte nommé "partieSauvegarde.txt". Les détails concernant la structuration de ce fichier, des données qu'il enregistre, sont expliqués dans la partie Annexe.



Choix du mode de jeu

Diagramme de classe



Ici, nous avons omis les getters et les setters pour la lisibilité. Nous n'avons pas non plus écrit les fonctions héritées par Unité dans les classes Soldat, Catapulte, SuperSoldat et Archer. Nous avons choisi d'utiliser le préfixe "m_" dans le code pour les variables membres de classe (excepté les statics) afin de les différencier des autres variables du programme, ici nous avons omis le préfixe.

Explication de la structure du programme

Classe **Plateau**:

Le plateau est la classe qui nous sert à gérer le terrain de jeu et le déroulement général d'une partie. C'est cette classe qui est en charge d'effectuer / valider les actions demandées par une unité (attaquer / avancer) avec les fonctions suivantes :

- fonction **deplacerUnite**() : permet de déplacer une unité
- fonction **attaquerUnite**() : inflige des dégâts à une unité
- fonction **attaquerJoueur**() : inflige des dégâts à un joueur
- fonction **faireEvoluerUnite**() : permet de faire évoluer une unité (remplace une unité par son évolution)
- fonction **creerUnite**() : créer une unité sur le plateau
- fonction **detruireUnite**() : détruire une unité sur le plateau

La gestion des différents modes de jeu s'effectue avec :

- fonction **jeu**() : organise le type de partie qui sera jouer en fonction de l'entrée de l'utilisateur
- fonction **jeuJoueurContreIA**() : lance une partie de type J vs IA
- fonction **jeuJoueurContreJoueur**() : lance une partie du type J vs J
- fonction **donnerArgentDebutTour**() : donne de l'argent aux joueurs à la fin d'un tour l'attribut **nombreTour** permet de connaître le nombre de tours déjà effectué
- l'attribut **partieChargé** permet de savoir si on a chargé une partie (donc il faut reprendre la partie au bon endroit.)
- l'attribut **jeuContreJoueur** permet de connaître le mode de jeu choisi
- l'attribut **TAILLE_PLATEAU** permet d'initialiser le plateau avec une taille par défaut, de cette manière il n'y a plus de valeur brute dans le code, et nous avons la possibilité d'avoir un plateau de la taille que l'on souhaite.
- l'attribut **cases** est un tableau contenant toutes les unités du jeu

Enfin elle est également responsable de la sauvegarde d'une partie:

- chercher s'il existe une partie sauvegarder : **chercherPartie**()
- charger une partie: **chargerPartie**()
- sauvegarder une partie: **sauvegarderPartie**()

Classe **Joueur**:

La classe Joueur représente un joueur participant à une partie. Elle permet de faire le lien entre le plateau et les unités que le joueur a mis en place durant la partie avec :

- attribut **sonPlateau** : le plateau de jeu de la partie
- attribut **sesUnites** : unités que le joueur possède actuellement sur le plateau de jeu.

Elle a aussi pour but de représenter la base que les unités ennemies attaquent quand ils sont à l'extrémité du plateau de jeu :

- attribut **argent** : la quantité d'or que le joueur possède
- attribut **ptsDeVie** : le nombre de points de vie restant pour la base
- attribut **base** : numéro de la case du plateau où se situe la base (0 ou autres extrémité)

Les joueurs jouants chacun leur tour, c'est dans cette classe que le tour d'un joueur se déroule, par l'appel à :

- fonction **jouer**() : appel des différentes actions de chacune des unités du joueur, puis par le choix de l'action du joueur à la fin de son tour.
- fonction **jouerIASimple**() : appelée lorsque le joueur est une IA, le déroulement est le même que dans jouer(). Le choix d'une unité est fait selon l'or disponible, dans le but d'acheter l'unité la plus chère pour son budget.

Un joueur a la possibilité d'invoquer une unité à la fin de son tour de jeu, appelant les fonctions :

- fonction **creationUnite**() : Se charge de demander le choix de l'utilisateur concernant la création ou non d'une unité, ou bien de sauvegarder. Appelle ensuite la fonction correspondante au choix.
- fonction **creationUniteIA**() : effectue le choix de l'IA concernant la création d'une unité.
- fonction **creerSoldat**() : crée une unité Soldat pour le joueur sur le plateau
- fonction **creerArcher**() : crée une unité Archer pour le joueur sur le plateau
- fonction **creerCatapulte**() : crée une unité Catapulte pour le joueur sur le plateau

Classe **Unité**:

La classe Unité est une classe abstraite. Elle détermine le comportement standard d'une unité. Ainsi toute classe qui hérite d'unité devra donc implémenter les fonctions abstraites pures. Cela permet ainsi de forcer chaque classe qui hérite à redéfinir ses actions, comment elle repère un ennemi et sa portée.

Cette classe permet de définir les actions des unités :

- fonction **action1**() : les actions de la phase 1
- fonction **action2**() : les actions de la phase 2
- fonction **action3**() : les actions de la phase 3:
- fonction **avancer**() : cette fonction se charge de faire avancer une unité. Elle appelle la fonction du plateau, comme l'unité ne connaît pas l'état du plateau, nous avons fait en sorte que c'est le plateau qui le gère
- fonction **ennemieAPorte**() : cette fonction récupère la position de l'ennemie à attaquer le plus proche
- fonction **attaquer**() : cette fonction permet de faire attaquer une unité. Elle appelle la fonction du plateau qui attaque l'unité à portée.
- fonction **typeUnite**() : elle récupère le nom court du type de l'unité

- fonction **peutEvoluer**() : elle vérifie si l'unité peut évoluer (impossible d'évoluer par défaut)
- fonction **evolution**() : elle permet de faire évoluer une unité, renvoyant l'instance de sa classe d'évolution.
- attribut **veutEvoluer** est un booléen qui nous dit s'il l'unité est prête à évoluer
- attribut **prix** : prix d'une unité
- attribut **ptsDeVie** : points de vie d'une unité
- attribut **ptsAttaque** : points d'attaque d'une unité
- attribut **position** : position actuelle de l'unité sur le plateau
- attribut **joueur** : pointeur sur le joueur possédant l'unité

Classe **Archer**, **Soldat**, et **Catapulte**:
Ces classe héritent d'Unité, et donc implémentent les fonctions obligatoires (abstraites pures), et éventuellement les méthodes concernant l'évolution d'un personnage (comme Soldat)
Pour la classe Soldat :

- attribut **dejaAttaque** : booléen qui permet de savoir si le soldat a déjà attaqué lors de la phase 1, si c'est le cas, il ne pourra pas attaquer en phase 3.
- fonction **peutEvoluer**() : comme le soldat peut évoluer en SuperSoldat, elle doit implémenter la fonction afin d'autoriser l'évolution.

Pour la classe Catapulte :

- attribut **dejaAttaque** : booléen qui permet de savoir si la catapulte a déjà attaqué lors de la phase 1 si c'est le cas, elle ne pourra pas se déplacer en phase 3.

Pour la classe Archer, elle fonctionne comme une unité.

Classe **SuperSoldat**:
Evolution d'un Soldat après qu'il est vaincu une unité adverse. Il est construit à partir d'une instance d'un soldat existant, l'instance de l'unité qui évolue. Le fonctionnement est le même que pour une unité soldat, faisant donc hériter SuperSoldat de la classe Soldat. Les différences se situent au niveau de l'action 3 qui est redéfinie afin de pouvoir attaquer 2 fois par tour. De plus, le Super-Soldat n'ayant pas d'évolution, les fonctions concernant l'évolution sont remises dans le même état que celles que nous pouvons trouver dans la classe Unité.

Technologies particulières utilisées

Afin de représenter le plateau dans le jeu, nous étions partis au départ sur l'idée d'un tableau fixe de 12 cases de type Unite**, chaque case stockant donc un pointeur vers une instance d'une unité disponible dans le jeu ou bien tout simplement nullptr quand la case est vide. Cependant nous nous sommes rendu compte, après avoir écrit une très grande majorité du code, qu'il était demandé dans l'énoncé d'utiliser une structure de données issue de la STL pour le stockage des données. Nous avons commis cette erreur tout simplement par une faute d'inattention lors de la lecture de l'énoncé. Ainsi, nous avons opté pour un stockage du plateau sous la forme d'un std::array, qui est un tableau fixe pouvant contenir divers types de données et étant issu de la STL. L'opérateur [] étant redéfini pour les std::array, le changement de structure s'est fait très simplement et nous a même permis d'améliorer certaines fonctions et boucle par l'utilisation d'itérateurs.

Nous avons aussi eu l'envie d'améliorer le jeu en permettant d'avoir une taille de plateau autre que 12 cases. Ainsi nous avons mis en place une "constexpr", sous forme d'un attribut static de la classe Plateau. Cette valeur est calculée à la compilation du programme par le compilateur, qui va remplacer partout dans le programme les appels à cet attribut par la valeur calculée. Cela permet ainsi de pouvoir changer la taille que l'on souhaite pour le plateau en ne changeant qu'une seule valeur à un endroit du programme et en le recompilant.

Ainsi, nous avons au sein du programme une déclaration des cases du plateau sous la forme suivante : std::array<Unite*, TAILLE_PLATEAU> qui est un tableau fixe de la STL donc la taille est calculée à la compilation.

Déroulement du projet

Nous avons réalisé ce projet en binôme (Alexis Proust et Jordane Minet). Nous avons commencé par réfléchir tous les deux au diagramme de classe indépendamment puis nous avons mis en commun et gardé nos meilleures idées. Lors du développement, nous avons travaillé tous les deux sur toutes les classes, chacun repassant sur le travail de l'autre et apportant un autre avis et de potentielles corrections. Les seuls parties réalisées par une seule personne à part entière sont le makefile réalisé exclusivement par Jordane, et la partie sauvegarde/chargement par Alexis.

Nous n'avons pas rencontré de problème majeurs pouvant nous bloquer sur l'avancée du projet. Cependant, nous avons eu quelques erreurs assez agaçantes, notamment des problèmes sur les inclusions de fichiers. Comme les headers s'incluent entre eux, nous nous sommes retrouvé avec des redondances et des masquages de classe.

Un des points améliorables dans notre projet serait qu'une partie du code aurait sûrement pu être synthétisée, c'est à dire ne pas avoir des "if" un peu partout afin de traiter chaque cas selon le côté du plateau et l'appartenance à un camp ou l'autre pour une unité. Cela aurait permis d'obtenir un code plus court et lisible.

Nous avons voulu rester fidèle au diagramme de classe modélisé en UML. Ainsi nous avons, pour la plupart des classes, des liaisons à double sens entre les objets. Par exemple, le Plateau connaît ses deux Joueurs, et les Joueurs connaissent le Plateau. Nous pouvons donc facilement circuler entre les classes et accéder aux fonctions

Le code est plutôt modulable, on peut ajouter autant de type d'unité différente que l'on souhaite car l'implémentation se base par rapport à la classe abstraite Unité. Il en est de même si l'on souhaite faire évoluer un Archer en un arbalétrier ou en un super-Archer. Nous avons permis d'implémenter l'évolution nos unités en mettant une structure inspirée du design pattern State sur son principe de promotion. Nous obtenons ainsi un programme modulaire verticalement et horizontalement pour l'ajout de nouveaux types d'Unité.

Nous nous somme assurés d'avoir le moins d'entrer de l'utilisateur possible. Toutefois, nous permettons au joueur de passé le tour sans jouer même quand il n'a pas assez d'argent pour l'unité la moins chère, car cela permet d'aider à la compréhension du déroulé du jeu. Le joueur ne pourra rentrer que des chiffres. Toute autre entrée sera rejeté et il vous sera demandé de rentrer à nouveau un chiffre.

Déroulement du jeu (étape)

********* Tour X *********

******* Affichage du plateau avant les actions du joueur 1 ******

********* Action 1 *********

********* Action 2 *********

********* Action 3 *********

******* Affichage du plateau après les actions du joueur 1 ******

******** Phase de création *******

******* Affichage du plateau avant les actions du joueur 2 ******

********* Action 1 *********

********* Action 2 *********

********* Action 3 *********

******* Affichage du plateau après les actions du joueur 2 ******

******** Phase de création *******

Annexes

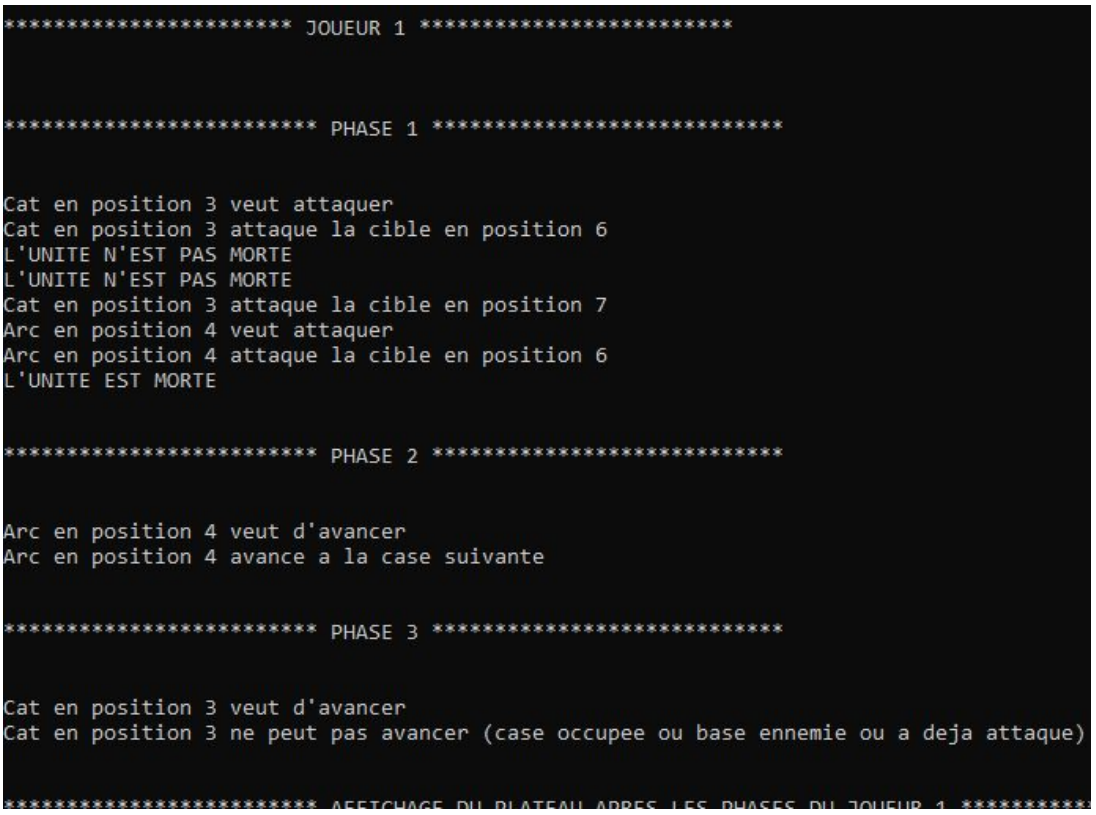
Explication du fichier de sauvegarde:
Ici nous avons mis des commentaires pour expliquer le fichier, en temps normal, il n'apparaît pas.

```
2                // mode de jeu (2 = J VS IA, 1= J VS J)
38 100           // Joueur 1 (38 pieces, 100 points de vie)
16 100           // Joueur 2 (16 pieces, 100 points de vie)
0                // Base du joueur courant (0=Joueur 1,11=Joueur 2)
-1               // Case 0 (-1=vide)
-1               // Case 1
-1               // Case 2
3 Cat 0 12       // Case 3
// type unite (Cat=Catapulte)
// base du joueur proprietaire (ici J2)(0=J1,11=J2)
// points de vie restant (ici 12)
-1               // Case 4
5 Arc 0 5        // Case 5
// type unite (Arc=Archer)
// base du joueur proprietaire (ici J1)(0=J1,11=J2)
// points de vie restant (ici 5)
-1               // Case 6
7 Arc 11 2       // Case 7
-1               // Case 8
9 Arc 11 8       // Case 9
10 Arc 11 8      // Case 10
-1               // Case 11
7                // Nombre de tour passé
```

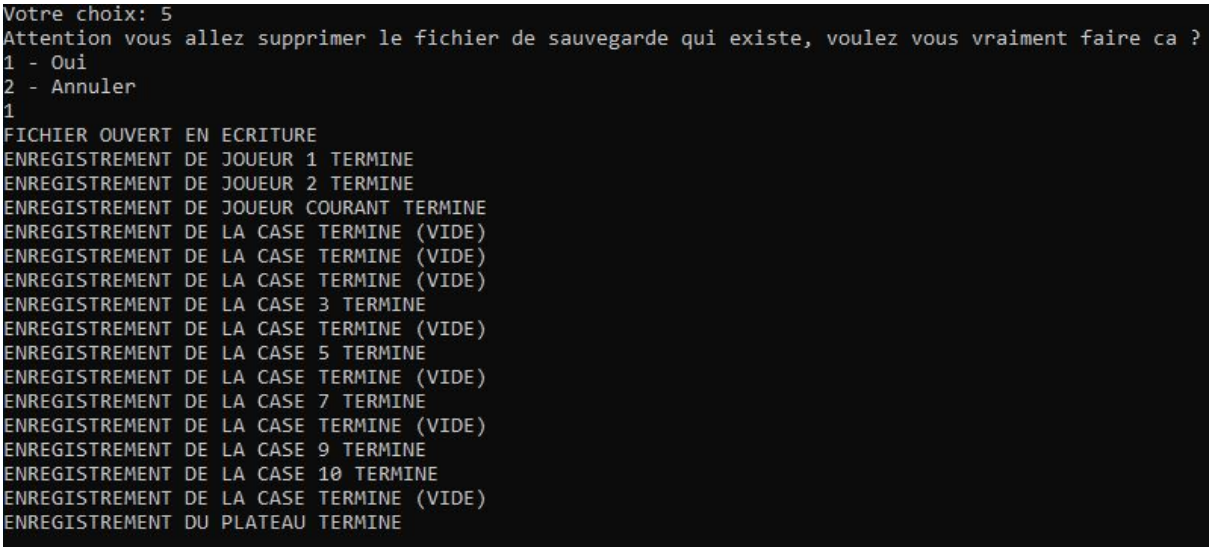
Image du jeu :

Début de tour:

Phases d'actions:



Sauvegarde d'une partie:



Chargement d’une partie (ici contre l’IA):

```
3 - Reprendre la partie enregistree
Votre choix (numero): 3
FICHIER OUVERT EN LECTURE
CHARGEMENT DE JOUEUR 1 TERMINE
CHARGEMENT DE JOUEUR 2 TERMINE
CHARGEMENT DE JOUEUR COURANT TERMINE
CHARGEMENT CASE ACTUELLE 0
CHARGEMENT DE LA CASE TERMINE (VIDE)
CHARGEMENT CASE ACTUELLE 1
CHARGEMENT DE LA CASE TERMINE (VIDE)
CHARGEMENT CASE ACTUELLE 2
CHARGEMENT DE LA CASE TERMINE (VIDE)
CHARGEMENT CASE ACTUELLE 3
Catapulte:
(Joueur 1)
Prix: 20
Points de Vie: 12
Points d attaque: 6
Position: 3

CHARGEMENT CASE ACTUELLE 4
CHARGEMENT DE LA CASE TERMINE (VIDE)
CHARGEMENT CASE ACTUELLE 5
Archer:
(Joueur 1)
Prix: 12
Points de Vie: 5
Points d attaque: 3
Position: 5

CHARGEMENT CASE ACTUELLE 6
CHARGEMENT DE LA CASE TERMINE (VIDE)
CHARGEMENT CASE ACTUELLE 7
Archer:
(IA)
Prix: 12
Points de Vie: 2
Points d attaque: 3
Position: 7

CHARGEMENT CASE ACTUELLE 8
CHARGEMENT DE LA CASE TERMINE (VIDE)
CHARGEMENT CASE ACTUELLE 9
Archer:
(IA)
Prix: 12
Points de Vie: 8
Points d attaque: 3
Position: 9

CHARGEMENT CASE ACTUELLE 10
Archer:
(IA)
Prix: 12
Points de Vie: 8
Points d attaque: 3
Position: 10

CHARGEMENT CASE ACTUELLE 11
CHARGEMENT DE LA CASE TERMINE (VIDE)
CHARGEMENT DU PLATEAU TERMINE
```