

Projet C++

Encadré par Marc Fonvielle et Joël Gay



Voici le rapport concernant ce projet de jeu similaire à Age Of War en C++. Le diagramme UML est fourni en annexe pour raison de lisibilité.

Commentaires sur le diagramme UML :

La classe Field est la classe principale de notre programme, en dessous du main. Elle contient le terrain à afficher ainsi que les deux joueurs.

Un terrain est composé de cases, et chaque case est représentée par une classe Square, sur laquelle peuvent potentiellement se trouver un combattant et une base.

La classe Castle représente une base. Il en existe donc deux instances qui seront contenues dans les instances de Square appropriées.

La classe Fighter est une classe abstraite représentant les combattants. Swordman, Bowman et Catapult en héritent, et redéfinissent leur propre système d'action. La classe Fighter sert aussi à la récupération de données via un fichier config.txt et à la redistribution de ces données aux instances des classes filles.

La classe abstraite Player correspond au joueur. Il peut être humain, représenté par la classe Human ou être un bot, représenté par la classe AI. Chaque type de joueur redéfinit sa manière de jouer.

Les instances d'AI utilisent pour jouer une instance des classes filles de la classe abstraite Strategy. Chaque sous classe de Strategy correspond à une stratégie de jeu prédéfinie.

Difficultés rencontrées :

Au cours du développement de notre projet, nous avons peu été confrontés à de réelles difficultés.

La première vraie difficulté était un souci que nous avions avec les #include pour faire communiquer nos classes entre elles. Mais après s'être penché sur la question, nous avons remarqué qu'il s'agissait d'une simple interversion entre " et <.

La deuxième difficulté a été la gestion des pointeurs dans le code, notamment dans les vector. Nous avons donc restructuré entièrement notre code pour pouvoir l'adapter à l'utilisation des pointeurs.

Enfin la dernière difficulté était un problème d'accès à la mémoire particulièrement bien caché. En réalité, un vecteur mal initialisé était caché derrière cette erreur.

Heureusement, ces trois problèmes ont été résolus, et ne nous ont pas empêché de terminer le code du projet à temps.

Règles du jeu :

Le jeu commence par proposer 2 choix au joueur :

- Commencer une nouvelle partie
- Reprendre la dernière partie coupée avant la fin du jeu

Si le joueur choisit 1 alors qu'aucune partie n'a été jouée auparavant, une sauvegarde par défaut est lancée en nouvelle partie de joueur contre joueur.

```
new game : 0 ; load game : 1
```

Dans le cas d'un nouveau jeu, le joueur peut choisir 2 modes de jeux :

- Mode joueur contre joueur pour jouer à 2
- Mode Joueur contre l'ordinateur pour joueur seul. 2 stratégies sont proposées dans ce cas :
 - un ordinateur qui achètera toujours l'unité la plus cher à envoyer au combat
 - un ordinateur qui effectue le même pattern d'achat d'unité

```
new game : 0 ; load game : 1
0
Player VS Player : 0 ; Player VS PatternBot : 1 ; Player VS RicherBot : 2
```

Une fois ces choix faits , le jeu à proprement parlé commence avec l'interface suivante:

```
Turn 1
=====
It is your turn LEFT player
=====

your castle has 100 health points left
enemy's castle has 100 health points left

M_____M

gold : 13
0 = end turn ; 1 = swordman (10) ; 2 = bowman (12) ; 3 = catapult (20) ;
```

Le joueur de gauche commence en premier.

Le jeu propose 3 unités différentes comme indiqué dans le sujet avec chacune des compétences différentes.

Tous les éléments en bleu représentent les possessions de l'équipe de gauche, et en rouge celles de l'équipe de droite.

tour courant	joueur courant	actions des personnages durant ce tour
Turn 5		

It is your turn LEFT player		

swordman 0 moved to 4		
swordman 1 moved to 3		
your castle has 100 health points left		
swordman 1 has 10 health points left		
swordman 0 has 10 health points left		
catapult 0 has 12 health points left		
bowman 0 has 8 health points left		
enemy s castle has 100 health points left		
ss c b	M _____ M	
gold : 25		
0 = end turn ; 1 = swordman (10) ; 2 = bowman (12) ; 3 = catapult (20) ;		

informations sur l'état des bases et unités

Terrain

Magasin

Les actions indiquent qui a bougé où, qui a attaqué , qui est tombé au combat, qui est devenu super soldat et combien le joueur courant a gagné de pièces d'or suite à la défaite d'un ennemi, dans l'ordre précis é par l'énoncé.

L'affichage des points de vie se fait en partant de la gauche du terrain vers la droite.

Sur le terrain les bases sont représentées par "M" , les catapultes par c , les archers par b et les soldats par s (S pour un super soldat). Les différentes cases sont représentées par les 2 bases.

Le magasin affiché à chaque tour rappelle les unités existantes et leur prix, et l'argent du joueur. En cas de fonds insuffisants ou d'impossibilité d'invoquer une unité, un message adapté indique que le tour est passé.

Les rubriques vont toujours tout au long de la partie garder le même code couleur (sauf l'annonce du joueur courant qui est à la couleur du joueur).

```
catapult0damaged enemy s castle

enemy s castle has -2 health points left
catapult 0 has 12 health points left
bowman 0 has 8 health points left
bowman 1 has 8 health points left
your castle has 100 health points left

cbb
M_____M

gold : 177
0 = end turn ; 1 = swordman (10) ; 2 = bowman (12) ; 3 = catapult (20) ;
0
next turn

=====
Right player wins
=====

cbb
_____M
```

A la fin d'une partie, si elle a été remportée par un joueur, le programme l'annonce et la base du perdant est détruite. Sinon, un message informe qu'il y a égalité.

Configuration :

Une fonctionnalité optionnelle qui a été prise en compte dès le début du projet a été la possibilité de configurer le jeu via un fichier config.txt : choix de la taille du terrain, argent reçu par tour, argent de départ, spécification des paramètres relatifs aux unités, points de vie des bases.

Les paramètres dans ce fichier suivent le format suivant :
paramètre=valeur

ATTENTION : aucune vérification sur les valeurs données au fichier config.txt n'a été implémentée. De même pour la sauvegarde.

Le fichier save.txt conserve les paramètres de la dernière partie jouée dans son état exact, ainsi que le contenu du fichier config.txt utilisé lors de cette partie.

Répartition des tâches :

La très grande majorité du code a été faite en binôme. L'un demandait à l'autre son avis sur la structure à adopter et nous codions ensemble. Cela nous a évidemment coûté du temps, mais cela nous a permis d'avoir une structure de code solide utilisant les bonnes idées que chacun de nous pouvait avoir, sans avoir à régler de problèmes de compatibilité ou d'accessibilité entre les classes comme nous les aurions eus si nous avions travaillé chacun de notre côté.

Les seules exceptions sont l'implémentation du système de sauvegarde, qui a intégralement été fait par Maxime, quand les données relatives à l'affichage et au confort de l'interface (sauts de ligne, caractères utilisés, couleurs...) ont été ajoutées au code par Clément.

Conventions de codage :

Dans le code de ce jeu, nous avons adopté plusieurs conventions:

1. le code se fait en anglais exclusivement
2. les commentaires se font en français exclusivement
3. le nom des classes, constructeurs etc commence par une majuscule (imposé par le langage), mais tout le reste (nom de variable, nom de méthode...) commence par une minuscule et se forme de la sorte : variableEnPlusieursMots.

Choix de réalisation et remarques :

Le projet est fourni avec un fichier config.txt contenant les valeurs par défaut de l'énoncé du projet, ainsi qu'un fichier de sauvegarde par défaut, qui correspond à une partie non entamée.

Notre main pour ce projet est NotAgeofWar.cpp. Il commence par initialiser le jeu avec les données dans le config.txt et save.txt.

Le int turn avant la boucle principale prend pour valeur 1 ou -1, ce qui nous sert par la suite dans tous nos index pour adapter les actions des joueurs. Le même principe a été repris dans l'attribut firstPlayer de la classe Player.

Le joueur indique son mode de jeu et ensuite la boucle principale du jeu est lancée.

La map statique dans la classe Stratégie permet d'ajouter plus facilement des stratégies supplémentaires dans le jeu à l'avenir. Les commentaires dans le fichier Strategy.cpp définissent la démarche exacte à suivre pour cela.

Dans la stratégie pattern, l'index du pattern n'est pas sauvegardé, ce qui fait qu'en cas de reprise de partie, l'IA recommencera depuis le début son schéma d'invocation.

Les 2 grandes classes IA et Human utilisent la même méthode principale update(square) qui permet d'invoquer une unité sur la base (le square en question).

Une classe Castle à part a été créée pour les bases. Une classe Entité aurait pu être créée pour contenir les points de vie et le int firstPlayer. Castle et Fighter hériteraient alors de cette classe mais nous avons jugé que cela n'était pas nécessaire pour uniquement ces 2 attributs.

Toutes les données relatives aux unités dans le config.txt sont stockées dans une map statique et c'est la méthode setStats() qui attribue à chaque type d'unité ses caractéristiques depuis cette map.

Une deuxième map statique sert à associer à chaque sous-classe de Fighter un nom qui sert à diverses fonctionnalités, et une instance du Fighter en question.

Une troisième quant à elle sert à associer à chaque type de Fighter son affichage sur le terrain (exemple caractère s ou S pour un soldat).

On identifie une unité de manière unique par son nom et un numéro issu d'un compteur static incrémenté à chaque création du type de l'unité.

En cas de nouvelle partie, la classe field instancie le terrain et les joueurs à partir du fichier de configuration. Sinon tout se fait à partir du fichier de sauvegarde. La sauvegarde se passe dans la classe Field. Elle récupère les unités existantes, les points de vie... et écrit ces valeurs dans le fichier save.txt. La méthode save est automatique, le main l'appelle juste après qu'un joueur ait terminé son tour.