Université de Montpellier II

Master 1 Informatique

2008-2009

Rapport de TER

Algorithme min-max dans les jeux de stratégie historique

Quentin Bresson

Sébastien Long

Cédric Ogive

Table des matières

Introduction

Contexte

Les jeux dits de stratégie historique sont des jeux se jouant au tour par tour, entre deux joueurs ou plus. Ils se déroulent dans un contexte historique, qui peut aussi laisser place à un univers fantastique, ou de science-fiction, la constante étant l'affrontement de plusieurs armées. À la différence de jeux de société classiques, plusieurs coups peuvent généralement être joués à chaque tour de jeu, ce qui les rend bien plus complexes, mais aussi plus riches.

Le min-max est un algorithme très populaire dans les jeux dits de stratégie combinatoire abstraits à information complète, tels que le puissance 4, les dames, et bien d'autres jeux de société. En explorant et évaluant l'ensemble des coups possibles jusqu'à un niveau donné, il permet sans grand effort de faire jouer l'ordinateur à un niveau très convenable, supérieur à la majorité des joueurs humains.

À contrario, cet algorithme est généralement délaissé dans les jeux de stratégie historique, le plus souvent au profit de machines à états finis, ou même de solutions « bricolées ». En reposant trop sur l'intervention des développeurs, ces méthodes sont génératrices de failles, l'ordinateur devient prévisible.

Travail

Dans le cadre de ce TER de première année de M1, nous nous intéresserons à l'applicabilité de cet algorithme aux jeux de stratégie historique, qui entrent dans un cadre bien plus large, les joueurs pouvant effectuer un nombre important d'actions à chaque tour de jeu.

Pour cela, nous commencerons par créer une maquette de jeu, avec des graphismes simples en 2D, permettant le chargement de scénarios de batailles personnalisés, ainsi que la modification des caractéristiques du jeu par l'édition de fichiers texte. Nous tâcherons de mettre en place un gameplay suffisamment riche pour permettre l'émergence de stratégies, tout en gardant une certaine simplicité, pour ne pas complexifier inutilement le travail de l'IA.

Le principal intérêt de cette maquette sera d'offrir un support visuel aux actions de l'ordinateur, et de pouvoir se mesurer à lui.

Dans un second cas, nous nous penchons sur l'applicabilité du min-max à notre jeu de stratégie. Nous en verrons les limites, avant d'explorer d'autres solutions.

Le jeu

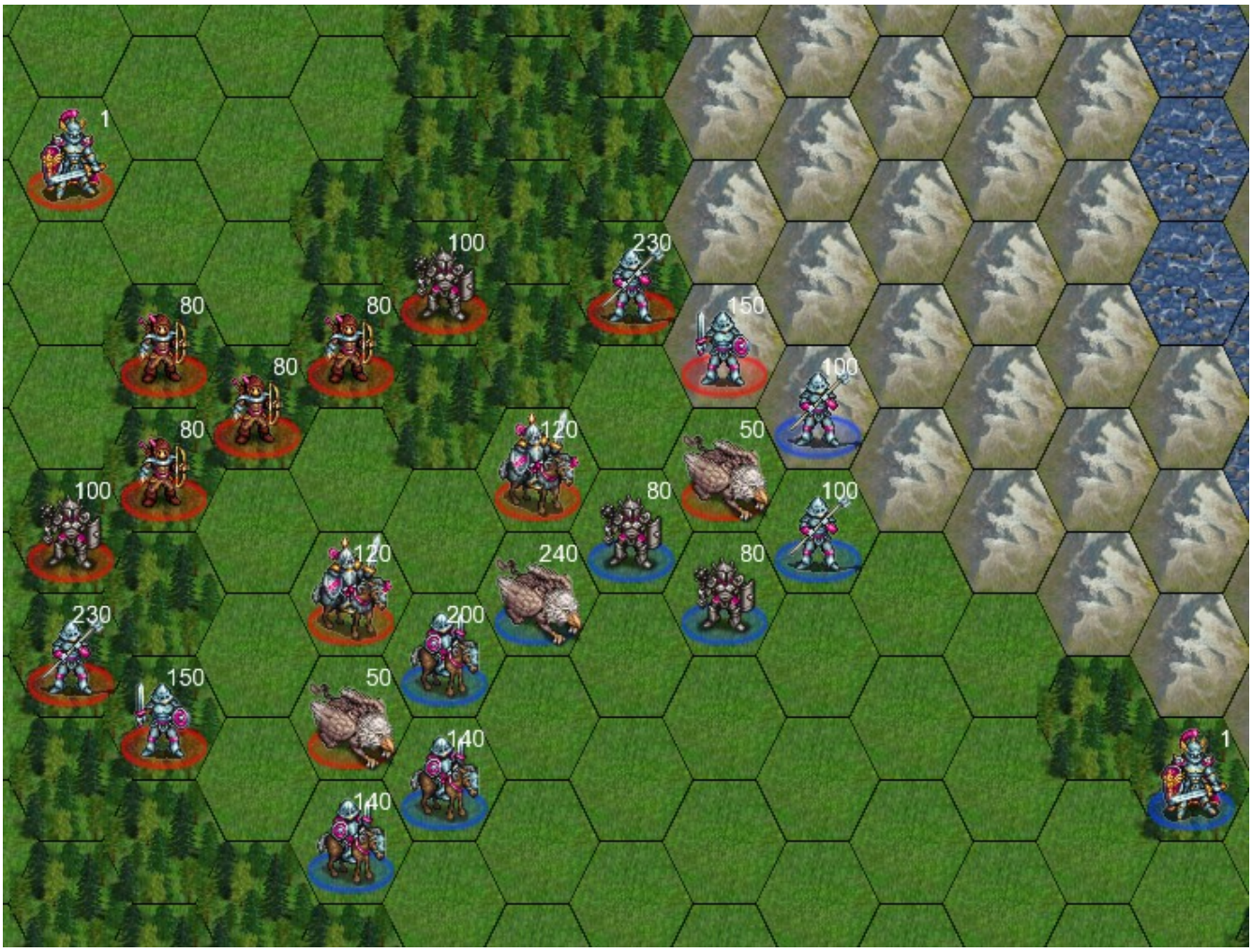
Généralités

Deux joueurs contrôlent des armées s'affrontant au tour par tour, dans un terrain de jeu composé de cases hexagonales. Chaque armée dispose d'un capitaine, et la mission des joueurs est double :

- Protéger son capitaine.
- Éliminer le capitaine ennemi.

La victoire appartient au premier joueur parvenant à éliminer le capitaine ennemi.

Voici une capture d'écran, pour commencer à se familiariser avec le jeu :



Nous voici au milieu d'un affrontement entre deux joueurs. Les troupes du joueur 1 se reconnaissent par la présence d'un ovale rouge sous leurs pieds, alors qu'il est bleu dans le cas du joueur 2. Les unités que l'on voit représentent chacune un groupe : la présence d'un archer sur une case, avec le

chiffre 80 placé à côté de sa tête indique que la case est occupée par 80 archers.

Les tours de jeu se déroulent tous de la même manière : le joueur ayant la main donne des ordres à ses unités : déplacements, attaques, fusion de deux groupes d'unités, ou division d'un groupe. Les déplacements, fusions et divisions se font immédiatement, tandis que les ordres d'attaque n'ont aucun effet immédiat. Lorsque l'utilisateur a fait ce qu'il souhaitait, il décide de finir son tour, ce qui entraîne l'exécution des attaques en attente de traitement. Le résultat des batailles est affiché, puis le joueur suivant prend la main.

Commandes de jeu

Commandes permettant d'agir sur la partie :

- Touche Echap : quitter le jeu.
- Touche Entrée : finir son tour de jeu. Les attaques planifiées sont faites, puis le joueur ennemi prend la main.

Commandes permettant d'agir sur la carte :

- Touches directionnelles : déplacement de la carte.
- Molette de la souris : s'approcher ou s'éloigner de la carte.
- Clic sur la molette de la souris : centrer la carte, et revenir au zoom par défaut.

Commandes permettant d'agir sur un groupe non sélectionné :

- Clic gauche sur une case non vide : sélectionne la ou les unités se trouvant sur cette case. Cela colore l'ensemble des cases sur lesquelles ce groupe peut se déplacer.

Commandes permettant d'agir sur un groupe préalablement sélectionné :

- Clic gauche sur une case vide : déplacer ce groupe sur la case choisie. Note : cela n'est possible que si la case n'est pas trop éloignée.
- Clic gauche sur un groupe allié : sélectionner ce groupe à la place du groupe actuel.
- Clic gauche sur un groupe ennemi : l'attaquer. Note : cela n'est possible que si l'unité ennemie est à portée de l'unité sélectionnée.
- Clic droit sur un groupe ami : si le groupe ami est choisi est du même type que le groupe préalablement sélectionné, alors les deux groupes fusionnent, pour avoir plus de force. Note : un groupe ne peut être impliqué dans une fusion qu'une seule fois par tour de jeu.
- Clic droit sur une case vide : le groupe préalablement sélectionné est partitionné en deux groupes d'une taille égale. Note : un groupe ne peut être impliqué dans ce genre d'opération qu'une seule fois par tour de jeu.

Stratégies

Zones de contrôle

Chaque groupe d'unité exerce un contrôle sur les six cases l'entourant. Lorsqu'un groupe ennemi passe sur l'une de ces cases, ses capacités de déplacement sont interrompues jusqu'au tour suivant,

où il pourra, s'il n'a pas été vaincu, prendre l'initiative de s'en éloigner. Cela a plusieurs implications :

- Un seul groupe d'unité bien placé peut obliger les ennemis à faire un détour considérable s'ils ne souhaitent pas se battre dans l'immédiat.
- Pour tenir une ligne de front face à l'ennemi, pour bloquer un passage par exemple, il n'est pas obligatoire d'occuper toutes les cases. Des failles d'une ou deux cases n'empêcheront pas de ralentir fortement l'ennemi.

Caractéristiques des terrains

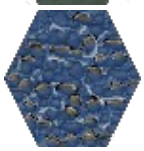
Les cartes du jeu sont constituées de plusieurs sortes de terrain, qu'il est intéressant de connaître pour exploiter leurs spécificités :

Plaine : le terrain le plus commun, sans caractéristiques particulières.

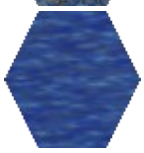
Forêt : un terrain dans lequel les archers parviennent à mieux se défendre, mais dans lequel les déplacements sont légèrement plus lents.

Montagne : les archers qui s'y trouvent voient leur portée augmenter. Les déplacements des unités lourdes y sont considérablement ralentis.

Marais : un terrain difficile pour toutes les unités, mais qui reste néanmoins moins handicapant pour les unités légères.



Eau peu profonde : un type de terrain à éviter à tout prix pour se battre, tant il réduit la mobilité des troupes. Cependant, il est souvent obligatoire de traverser des cours d'eau pour atteindre l'ennemi.



Eau profonde : un terrain impraticable. Seules les unités volantes ont la capacité de se trouver sur les cases de ce type.

Les unités

Les unités ont toutes des forces et des faiblesses, qu'il est important de connaître pour pouvoir inquiéter son adversaire. Voici les unités disponibles :

	**HP :** 80 **ATQ :** 40 **DEF :** 30 **Portée :** 1 (corps à corps) **Points de déplacement :** 7 **Bonus :** aucun **Malus :** aucun
Capitaine : l'unité qu'il faut protéger à tout prix sous peine de perdre la bataille. Bien qu'elle soit puissante, il serait fort imprudent de l'amener au cœur de l'action.	

	**HP :** 40 **ATQ :** 20 **DEF :** 5 **Portée :** 1 (corps à corps) **Points de déplacement :** 5 **Bonus :** aucun **Malus :** aucun
Fantassin léger : une unité relativement faible, mais polyvalente. Étant présente en grand nombre dans les batailles, elle ne doit pas être prise à la légère.	

	**HP :** 50 **ATQ :** 25 **DEF :** 12 **Portée :** 1 (corps à corps) **Points de déplacement :** 4 **Bonus :** aucun **Malus :** aucun
Fantassin lourd : l'élite des fantassins, dotée d'une armure lourde. Sa puissance importante ne doit pas faire oublier ses difficultés à se mouvoir en terrain difficile.	

	**HP :** 50 **ATQ :** 15 **DEF :** 5 **Portée :** 1 (corps à corps) **Points de déplacement :** 8 **Bonus :** aucun **Malus :** aucun
Cavalerie légère : unité de cavalerie faiblement armée dont le principal intérêt réside dans ses capacités de déplacement, parmi les plus importantes qui soient.	

	**HP :** 80 **ATQ :** 30 **DEF :** 12 **Portée :** 1 (corps à corps) **Points de déplacement :** 7 **Bonus :** aucun **Malus :** aucun
Cavalerie lourde : rapide, puissant, ce nouveau modèle avec œillères chromées a tout pour plaire !	

	**HP :** 40 **ATQ :** 15 **DEF :** 5 **Portée :** 1 (corps à corps) **Points de déplacement :** 5 **Bonus :** ATQ+15 contre cavalerie **Malus :** aucun
Lancier : leur longue lance les rend particulièrement puissants face à la cavalerie, et leur permet de faire cuire du poulet sans se brûler. Cette dernière activité ayant sérieusement empiété sur leur temps d'entraînement, ils ne sont guère à leur aise lorsqu'il s'agit de manier leur lance plus finement. Éloignez-les donc de tout ce qui n'a pas quatre sabots.	

	**HP :** 35 **ATQ :** 20 **DEF :** 0 **Portée :** 5 **Points de déplacement :** 5 **Bonus :** Portée+2 en montagne **Malus :** aucun
Archer : L'ami des bêtes, qui devient un monstre sanguinaire lorsqu'il est perturbé dans ses ébats. Il n'en reste pas moins un faiblard peu armé et sans défense, mais ses tirs à distance pourraient vous surprendre.	

	**HP :** 100 **ATQ :** 30 **DEF :** 20 **Portée :** 2 **Points de déplacement :** 5 **Bonus :** aucun **Malus :** aucun
Dragon : animaux mythologiques par excellence, les dragons furent longtemps vénérés par une tribu de crétins congénitaux, qui, au fil des ères, développèrent des techniques de cosplay inégalées à ce jour. Attention ! Leurs chalumeaux, bien que bas de gamme, ne doivent jamais être pris à la légère.	

	**HP :** 80 **ATQ :** 25 **DEF :** 10 **Portée :** 1 (corps à corps) **Points de déplacement :** 7 **Bonus :** peut se déplacer sur tout type de terrain **Malus :** aucun
Griffon : Une envergure de plus de 5 mètres, la puissance d'un char d'assaut, et le cerveau d'un pigeon. Cela explique que la race soit au bord de l'extinction, alors qu'elle est au sommet de la chaîne alimentaire. Ne les négligez pas pour autant : ils pourraient confondre vos soldats avec des miettes de pain.	

Architecture du jeu

Objectifs

Le développement d'un jeu est une tâche très longue, au contraire du temps dont nous disposions. L'une de nos préoccupations a donc été, en plus des habituelles problématiques de modularité, de créer une architecture pouvant être mise rapidement en place, afin d'avoir davantage de temps à consacrer au développement de l'intelligence artificielle. Mais il n'était pas pour autant question de créer un amas de code informe, aux dépendances dignes d'un plat de spaghetti. Nous avons donc choisi de nous baser sur une structure de moteur de jeu classique, mais minimale, que nous allons présenter.

Caractéristiques principales

Le jeu est bâti autour de deux modules principaux : un moteur graphique, et un moteur de jeu. Il s'ajoute à cela un ensemble de données accessibles via une méthode statique d'une classe Singleton. Le moteur de jeu lit et modifie ces données, alors que le moteur graphique ne fait que les lire. Voici plus en détail le rôle de ces moteurs :

Moteur graphique :

- Afficher l'état de la partie à des moments précis.
- Interpréter les choix de l'utilisateur (clics, touches du clavier), de manière à en informer le moteur de jeu
- Donner à l'utilisateur des informations supplémentaires : indiquer pour un groupe donné quels sont les déplacements possibles par exemple.

Moteur de jeu :

- Déterminer si les demandes du(des) joueur(s) humain(s) enfreignent les règles du jeu ou non.
- Exécuter les demandes de l'utilisateur si elles sont applicables, et mettre le modèle de données à jour en conséquence.
- Prévenir le moteur graphique dès qu'un changement du modèle a eu lieu.
- Gérer les intelligences artificielles éventuellement présentes.

Gestion de l'exécution du programme

Il s'agit de l'un des aspects sur lesquels on a privilégié un gain de temps. Une démarche habituelle est d'inscrire dans une boucle les mises à jour du modèle et de l'affichage, en utilisant un ou plusieurs threads.

Une caractéristique de notre application est de fonctionner au tour par tour, et non en (pseudo) temps réel. Le modèle n'est donc modifié que lorsque l'utilisateur le souhaite, et en l'absence d'animations, l'interface ne doit être mise à jour que lorsque le modèle a changé. Cela nous a permis d'opter pour une solution relativement simple, qui permet d'avoir un impact quasi nul sur les ressources du processeur et de la carte graphique : le modèle n'est mis à jour qu'en réponse à un événement en provenance de l'utilisateur, et l'interface est mise à jour lorsque le modèle change, ou lorsque

l'utilisateur effectue une action (scrolling, zoom, etc). Le reste du temps est passé à attendre une intervention de l'utilisateur. Cela nous amène à nous pencher sur la communication entre les moteurs.

Communication entre les moteurs

La communication entre les moteurs s'effectue via un système d'envoi de messages utilisant le design pattern Observateur. Une petite « originalité » par rapport au schéma de base est qu'un moteur joue à la fois le rôle d'observé et d'observateur. Lorsqu'un moteur reçoit un message, il l'interprète lui-même, ou délègue cette tâche à d'autres objets dont il a la responsabilité. Il est alors susceptible de lui-même envoyer un message à d'autres moteurs pour leur répondre, les informer d'une mise à jour, ou formuler une nouvelle demande.

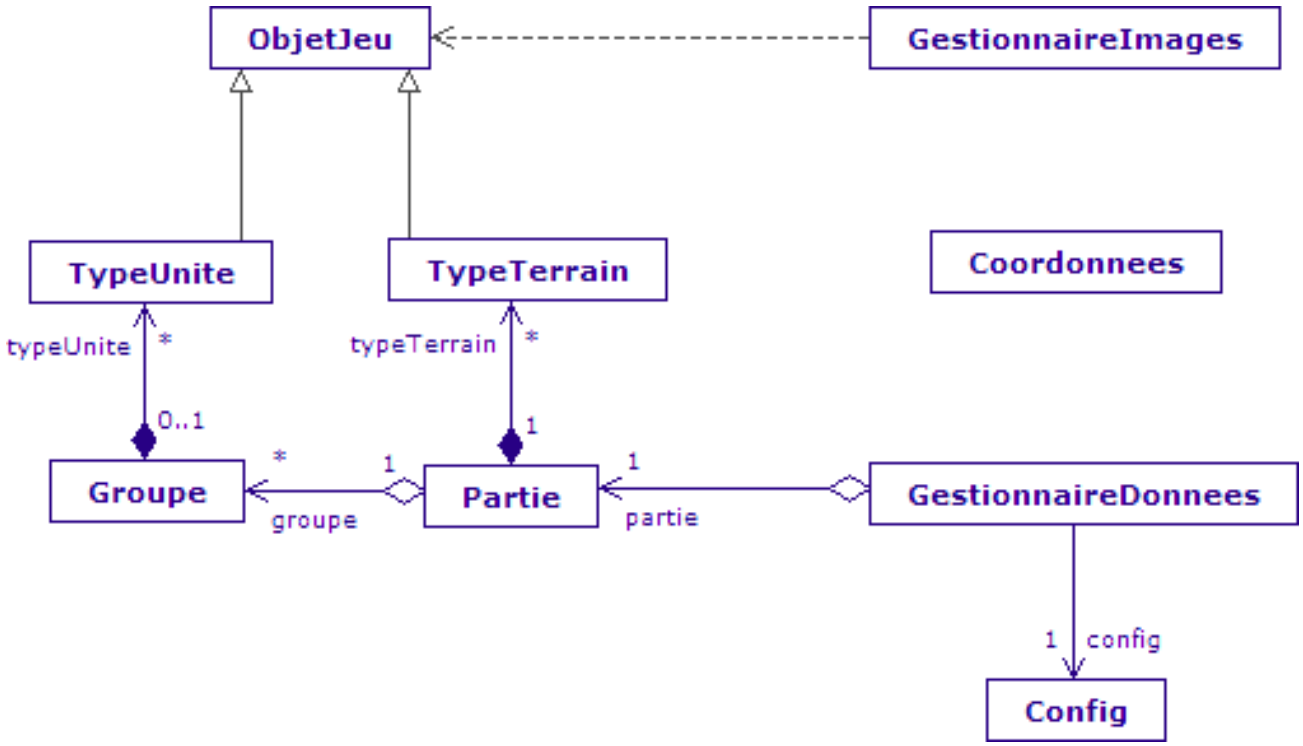
Les messages sont tous encapsulés dans des instances de Message. Grâce aux divers conteneurs qu'ils contiennent, ces objets peuvent contenir n'importe quel type de message, sans préjuger de quoi que ce soit, ce qui permet d'étendre les possibilités de communication sans avoir à modifier son code.

La structure de la classe n'offrant pas de contrôle des données, il est indispensable d'avoir un protocole de communication clairement défini. Il l'est grâce à la classe FabriqueMessage, basée sur le design pattern du même nom, qui cadre la construction des instances de Message en fonction des besoins de l'utilisateur, et donne via sa documentation les moyens de les décoder.

L'envoi et la lecture des messages se produisent dans le même thread, ce qui fait que la fonction envoyant le premier message ne se finit qu'à la fin de la chaîne d'envois et de réceptions de messages. Alors que cela poserait problème dans d'autres circonstances, l'impact de ces « blocages » est ici très faible: en effet, suite à un événement en provenance de l'utilisateur, il n'y a jamais plus d'un envoi de messages suivi d'une réponse. L'application garde donc une réactivité largement suffisante dans le cadre de nos tests.

Les données, plus en détail

Un diagramme de classes sera plus efficace qu'un long discours pour présenter les structures de données. Nous ne présenterons ici que les classes par souci de clarté, les méthodes étant en grande majorité des getter ou setter, donc sans grand intérêt.



Détail des classes :

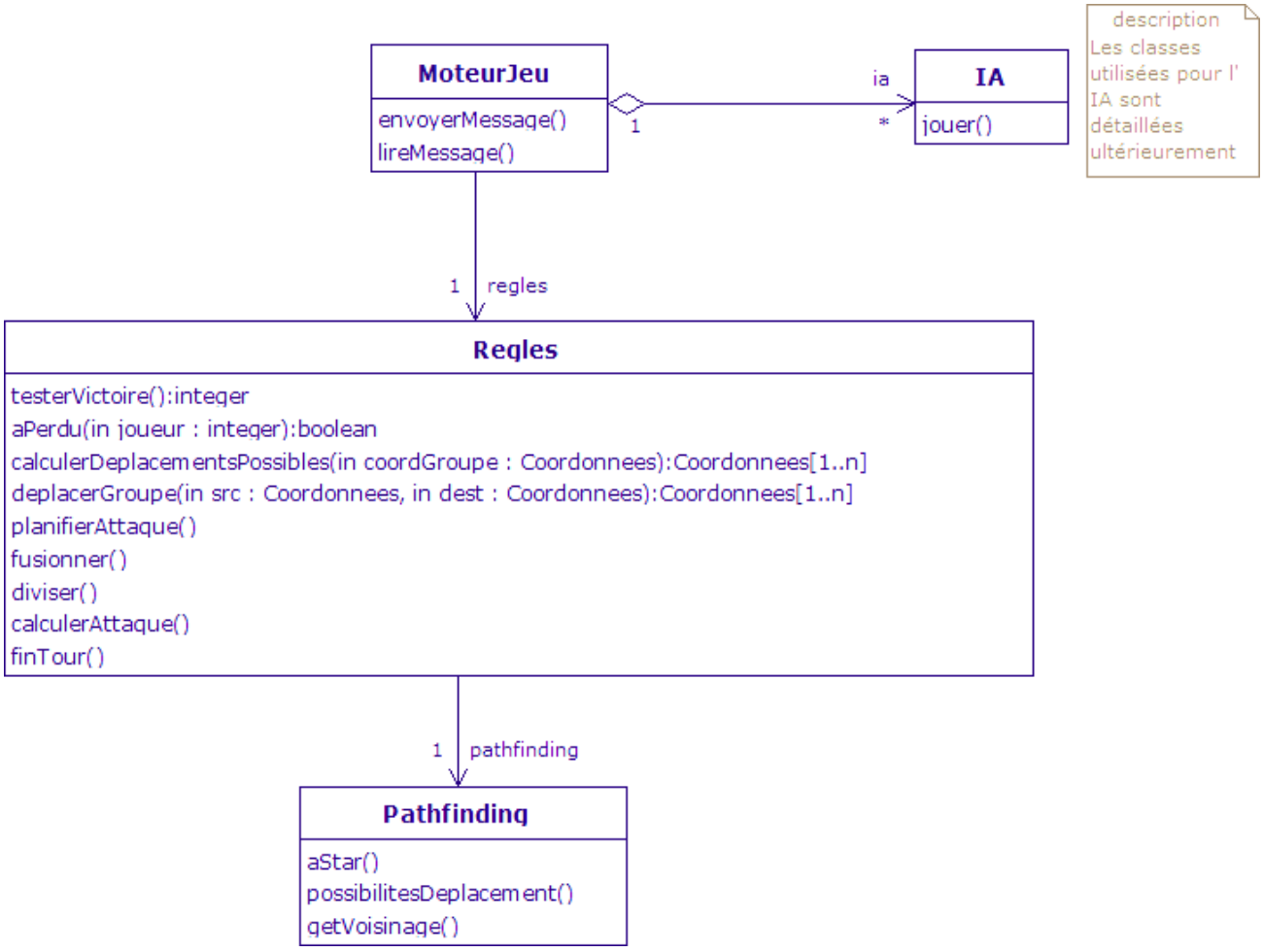
- **GestionnaireDonnees :** la classe principale pour ce qui concerne la gestion des données du jeu. Il s'agit d'un singleton chargeant l'ensemble des données au démarrage, et dont la méthode getInstance() constitue le point d'accès aux données du jeu (encapsulées dans un objet Partie).
- **Config :** le type d'objet auquel l'instance de GestionnaireDonnees délègue le chargement des données, à partir des fichiers de configuration.
- **ObjetJeu :** la base commune à tout objet du jeu. Ils sont de deux sortes : les groupes d'unités, et les cases du terrain de jeu.
- **TypeUnite :** Définit un type d'unité (ex : fantassin léger, archer, etc), avec les statistiques de base allant avec (attaque, défense, bonus ou malus contre d'autres TypeUnite, etc). Ces objets ne sont jamais présents en double. Si 30 groupes sont du même type « archer », alors l'objet TypeUnite attaché à ces groupes sera identique.
- **TypeTerrain :** Définit un type de terrain (ex : plaine, forêt, etc). À l'image des types d'unités, ces objets sont uniques. Si 100 plaines sont présentes sur une carte donnée, alors les 100 cases en questions pointeront toutes sur le même objet TypeTerrain.
- **Groupe :** Les groupes d'unités tels qu'ils sont présents dans la carte du jeu. Un objet Groupe, en plus d'être attaché à un TypeUnite, comporte des informations relatives au tour de jeu actuel (ex : est-ce que le groupe peut encore attaquer ce tour-ci ? Se déplacer ? Etc.)
- **GestionnaireImages :** La seule classe dépendant de la librairie graphique SFML. Il s'agit d'un singleton gérant l'ensemble des images (sf::Image) utilisées pour l'affichage de la partie. En contrôlant l'accès aux images du jeu, elle permet de s'assurer qu'elles ne soient jamais chargées inutilement.
- **Coordonnees :** Une simple classe utilitaire, stockant des coordonnées entières (x,y). Sa présence permet, en évitant d'avoir recours à des sf::Vector2f, de limiter le couplage entre notre projet et la librairie SFML.

Le moteur de jeu, plus en détail

Le moteur de jeu fonctionne toujours suivant le même principe : sa classe principale, « MoteurJeu », lit des messages en provenance du moteur graphique. Il s'agit dans tous les cas de demandes du joueur courant. Certaines peuvent être exécutées systématiquement (ex : l'utilisateur souhaite finir son tour), alors que d'autres nécessitent des vérifications.

Lorsqu'un tour se finit, le moteur de jeu a la charge de faire jouer le joueur suivant, s'il est contrôlé par l'intelligence artificielle.

Tout cela s'accomplit à l'aide des classes suivantes :



Informations sur ces classes :

- **MoteurJeu :** La classe principale du moteur de jeu. Elle lit les messages en provenance de l'interface, délègue les traitements à effectuer aux classes Regles et IA, puis retourne des messages en indiquant le résultat.
- **Regles :** Une classe Singleton dont les méthodes sont chargées de vérifier si les coups proposés par les joueurs sont valides, et s'ils le sont, de les exécuter.
- **Pathfinding :** Contient diverses méthodes utilisant l'algorithme A*.

Afin de mettre plus en évidence le fonctionnement du moteur de jeu, nous allons maintenant présenter ce qu'il se passe dans des exemples d'exécution, à travers trois exemples :

Exemple : L'utilisateur a sélectionné une unité

- La méthode « lireMessage() » du moteur de jeu réceptionne le message (elle est appelée par son expéditeur).
- Cette méthode voit que le message est du type « veut_liste_deplacement », ce qui lui permet d'effectuer le traitement approprié. Le contenu du message (les coordonnées de la case) en est extrait.
- La méthode « calculerDeplacementsPossibles() » de la classe Regles est appelée pour déterminer l'ensemble des déplacements possibles pour ce groupe.
- Pour cela, la méthode « possibilitesDeplacement() » de la classe Pathfinding est appelée.
- Un message de type « liste_deplacement » est diffusé par le moteur de jeu.

Exemple : L'utilisateur souhaite finir son tour de jeu

- La méthode « lireMessage() » du moteur de jeu réceptionne le message (elle est appelée par son expéditeur).
- Cette méthode voit que le message est du type « veut_finir_tour ».
- La méthode « finTour() » de la classe Regles est appelée.
- Cette méthode appelle plusieurs fois la méthode « calculerAttaque() » de la même classe, afin de lancer l'ensemble des attaques qui avaient été planifiées précédemment par le joueur.
- Enfin, un message « changement_joueur » est diffusé par le moteur de jeu pour indiquer au reste de l'application – en particulier le moteur graphique – que le joueur ayant la main a changé.

Le moteur graphique, plus en détail

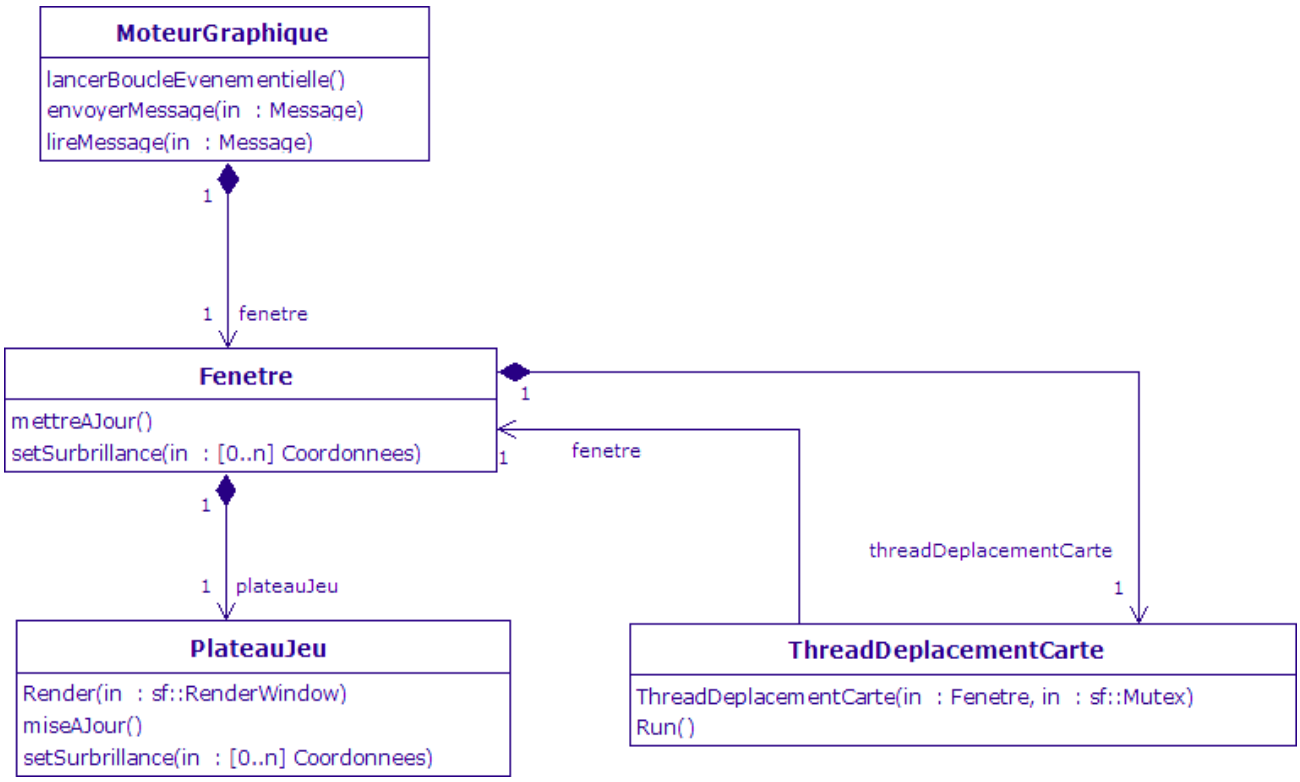
Graphismes

Le principal a été dévoilé à travers les images précédentes. Certaines remarques :

- Les sprites sont intégralement extraits du jeu libre « The Battle for Wesnoth »
- L'affichage se fait à l'aide de la librairie SFML dans sa version 1.3. Elle utilise openGL pour effectuer le rendu des scènes.

Fonctionnement

Comme nous l'avons vu précédemment, le moteur graphique a pour rôle de gérer tout ce qui concerne les interactions entre le joueur et l'interface. Il affiche l'état du jeu, et interprète les commandes de l'utilisateur (souris ou clavier) en vue d'un traitement par le moteur de jeu. Voyons plus en détail les classes mises en œuvre à cette fin :



Le moteur graphique se base sur les classes suivantes :

- **PlateauJeu :** un objet graphique SFML (sf::Drawable) qui représente le plateau de jeu, le met à jour en fonction du modèle, et l'affiche.
- **Fenetre :** La fenêtre de l'application. L'interface se voulant simple, elle ne contient qu'un objet PlateauJeu, auquel pourraient être adjoints divers menus dans une version plus complète. Elle dispose également d'une fonction permettant la mise à jour de la carte, celle-ci consistant principalement en l'appel de la fonction de mise à jour de la classe PlateauJeu.
- **MoteurGraphique :** la classe principale du moteur graphique. Elle écoute les évènements de l'utilisateur (clics, touches du clavier) grâce à sa fonction « lancerBoucleEvenementielle() », les interprète et les envoie au moteur de jeu. Elle se charge également de réceptionner des messages en provenance du moteur de jeu, afin d'effectuer les mises à jour nécessaires.
- **ThreadDeplacementCarte :** L'application ne s'exécutant que dans un seul thread, les évènements sont traités les uns à la suite des autres. Un traitement long pourrait donc figer l'application en l'attente de son exécution. Ce thread permet d'éviter cela : il consulte à intervalle régulier l'état de la souris et des touches du clavier, et met à jour la vue (sf::View) de la fenêtre en conséquence.

Afin de mieux cerner l'utilité des différentes fonctions de ces classes, étudions le fonctionnement du moteur graphique à travers plusieurs exemples :

Exemple : L'utilisateur clique sur une unité précédemment désélectionnée

- L'évènement est repéré par la méthode « lancerBoucleEvenementielle() » de la classe

MoteurGraphique.

- Il est transmis à la méthode privée « traiterEvenement() » de la même classe.
- La méthode « traiterEvenement() » détecte le fait que l'évènement est un clic. Elle transmet alors l'évènement à une nouvelle méthode privée de cette même classe : « interpreterChoixCase() ».
- La méthode « interpreterChoixCase() » identifie l'action de l'utilisateur comme étant la sélection d'une unité auparavant désélectionnée. Elle procède alors à l'envoi d'un message de type « veut_liste_deplacement » au moteur de jeu.
- La suite se passe dans le moteur de jeu.

Exemple : L'utilisateur fait tourner la roulette de sa souris vers le bas

- L'évènement est repéré par la méthode « lancerBoucleEvenementielle() ».
- Il est transmis à la méthode privée « traiterEvenement() ».
- La méthode « traiterEvenement() » détecte le fait que l'évènement est un mouvement de molette, ce qui se traduit par une demande de zoom sur le champ de bataille. La portée de cette demande se limite à la vue du champ de bataille, donc le moteur de jeu n'a rien à y faire.
- La méthode « traiterEvenement() » se charge donc de zoomer.
- L'affichage doit être mis à jour. La méthode « mettreAJour() » de la classe Fenetre est donc appelée. Elle appelle alors la méthode de même nom au seul objet graphique qu'elle manipule : le plateau de jeu.

Exemple : Réception d'un événement « liste_deplacement »

- La méthode « lireMessage() » de la classe MoteurGraphique réceptionne le message, ou plus exactement, elle est appelée par le moteur de jeu.
- Le message est identifié comme étant de type « liste_deplacement », ce qui signifie que les coordonnées des cases passées avec le message sont les déplacements possibles du groupe sélectionné, donc les cases devant être mis en surbrillance.
- La méthode « setSurbrillance() » de la classe Fenetre est donc appelée. Elle appelle à son tour la méthode de même nom de la classe PlateauJeu.
- Il ne reste plus qu'à mettre à jour l'affichage de la fenêtre, ce qui se fait via la méthode « mettreAJour() » de la classe Fenetre.

Intelligence artificielle

Problématique

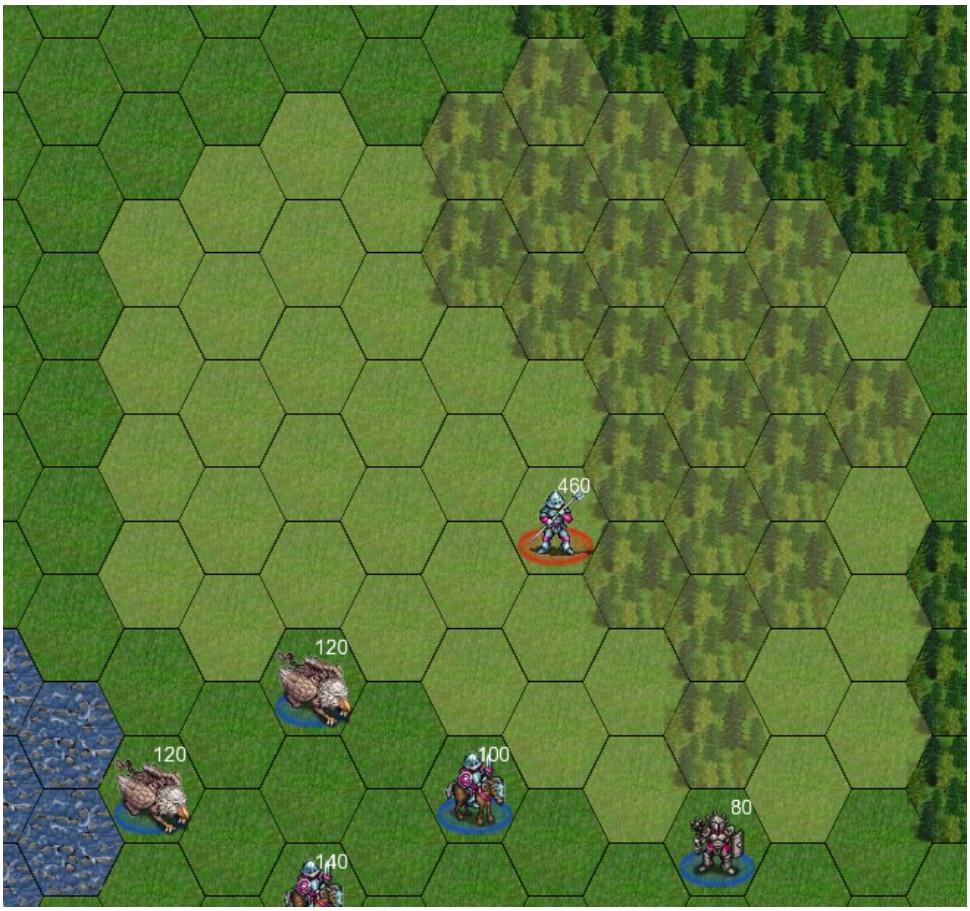
Le jeu de stratégie que nous avons développé se base sur un espace discret. Les déplacements d'unités se font toujours sur un ensemble de cases fini, et le temps s'écoule lui aussi par à-coups, du fait du mécanisme de jeu par tour. Lorsque la combinatoire d'un tel espace est faible, il est possible de l'explorer entièrement, ce qui facilite grandement l'élaboration d'une intelligence artificielle (qui ne mérite du coup pas vraiment son nom).

Dans notre cas, la combinatoire est énorme, puisque nous avons à faire à des dizaines de groupes qui peuvent chacun se déplacer sur des dizaines de cases différentes, mais aussi fusionner (avec plusieurs groupes au choix), se diviser (plusieurs cases de destination possible), attaquer des cibles (grand nombre de choix possibles si le groupe peut attaquer à distance), et ce, à chaque tour. Une approche « frontale » est donc exclue, mais il reste plusieurs choix :

- Tenter par élagage de revenir à un nombre de choix acceptable. Mais est-ce possible ?
- Considérer, dans les calculs d'IA, que l'univers du jeu est continu. On échappe au problème de combinatoire, mais les choix de l'IA deviennent imprécis. À quel point ?

Les limites du min-max

Nous avons commencé à nous pencher sur l'applicabilité du min-max à ce genre de jeu. Cela implique un élagage très important des coups jouables par chaque groupe. Prenons un exemple pour faciliter les explications :



Cette image montre l'ensemble des actions possibles pour l'unité rouge :

- Ne rien faire (1 action)
- Se déplacer (55 déplacements possibles)
- Se diviser en deux groupes égaux, ce qui est possible en chacun des 55 points accessibles. Le nouveau groupe pouvant être créé dans l'une des six cases voisines du groupe d'origine, on a un total d'environ 300 possibilités si la division se fait après les déplacements. Mais elle peut aussi se faire avant, ou pendant, ce qui donne un nombre de combinaisons extrêmement important (plusieurs milliers).
- Attaquer le griffon, le cavalier ou le fantassin. Mais ces groupes peuvent être attaqués de plusieurs manières possibles, suivant que le groupe rouge soit resté seul ou se soit divisé, et suivant les fronts choisis. Au final, on a là aussi des dizaines de possibilités.

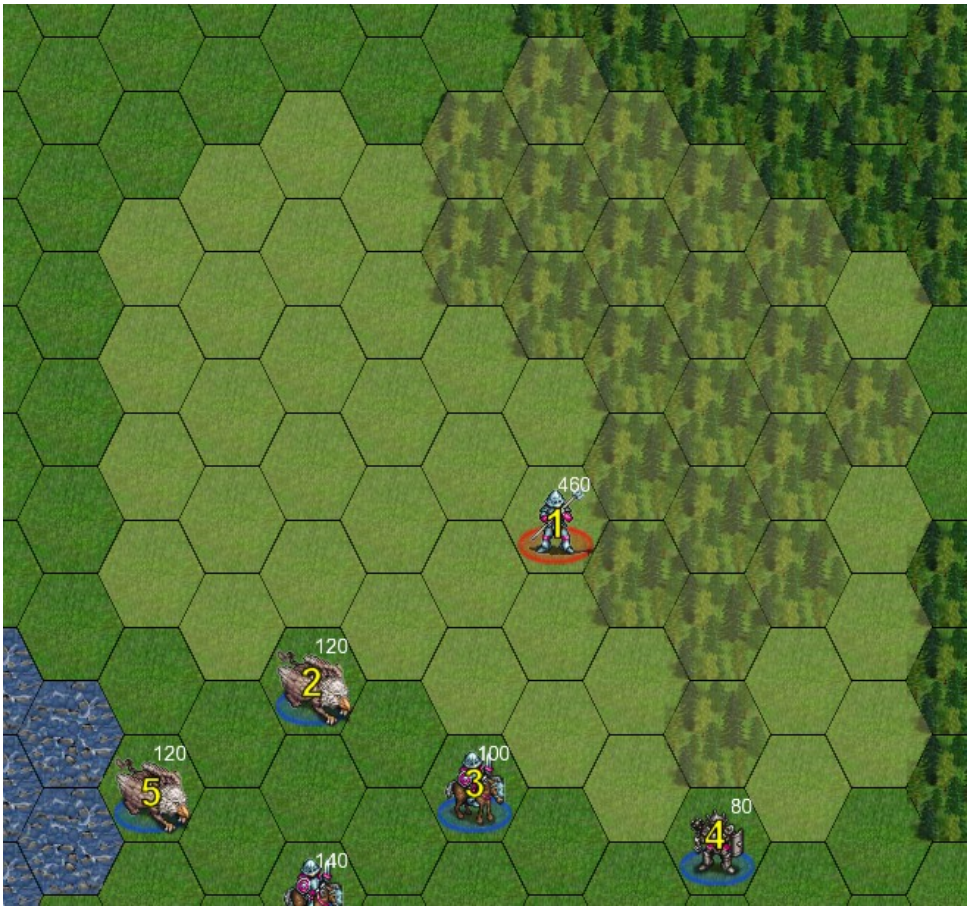
Cet exemple montre le besoin évident d'un élagage poussé, les parties pouvant de plus mettre en jeu

des dizaines de groupes différents. Sans cela, et même avec une profondeur minimale, le min-max est totalement inutilisable.

Notre première approche a été de considérer qu'il n'y a que trois sortes de coups réellement intéressants, pour chaque groupe :

- Rester sur place
- Attaquer directement un groupe
- Se déplacer en direction d'un groupe inaccessible pour le moment, en vue d'une attaque ultérieure.

Exemple :



Dans ce cas, et en ne retenant qu'une manière d'accomplir les cinq objectifs, on a 5 possibilités pour le groupe choisi.

On rencontre alors de gros problèmes :

(a) Nous avons retenu 5 possibilités pour le groupe choisi dans l'exemple, mais une armée n'est jamais constituée d'un seul groupe. Supposons que tous les groupes aient en moyenne 5 coups possibles pris en compte, et qu'un joueur en contrôle 10. On peut alors supposer qu'il y a environ 5 ^ 10 = 9765625 coups à prendre en compte. Aussi performante que soit la fonction

d'évaluation, aucun ordinateur de bureau ne pourrait l'exécuter près de 10 millions de fois en un temps raisonnable.

(b) En limitant autant les actions possibles, les unités deviennent incapables de coopérer entre elles. Par exemple, elles ne peuvent pas se déplacer en formation serrée, ce qui est pourtant un mouvement de base pour n'importe quel joueur humain débutant.

(c) Les actions se faisant les unes après les autres, leur ordre est extrêmement important. Un mauvais ordonnancement empêche toute action coordonnée (ex : unités qui se bloquent entre elles lors d'un déplacement). Ce n'est donc pas quelque chose que l'on peut négliger si facilement, et cela fait que l'estimation de 10 millions de coups à évaluer par tour est, en fait, bien naïve. Cela peut, au pire, être multiplié par le nombre de permutations possibles entre les actions du tour.

(d) Dans l'exemple choisi (simpliste), l'élagage des coups possibles se base sur le postulat que les coups optimaux soient toujours ou presque des attaques. Les déplacements sans but apparent peuvent du coup être supprimés sans problèmes, puisque les chances d'y trouver le coup optimal sont minimales. Or, nous nous sommes rendu compte que c'est le plus souvent faux. La profondeur stratégique du jeu réside avant tout dans le placement des unités. Une gestion habile du positionnement de ses unités permet de se battre plus efficacement, mais aussi – et surtout – de limiter les capacités de déplacements de l'adversaire. Se faisant, leur capacité à défendre leur capitaine diminue, ce qui peut suffire à remporter la victoire.

Au final :

- Le problème (a) pourrait être en partie résolu en limitant encore plus le nombre de possibilités offertes à chaque groupe. Mais leur niveau serait alors tout aussi ridicule que leur vision du jeu. Un min-max n'a pas un grand intérêt s'il produit des coups quasi aléatoires... On pourrait également tenter de constituer des ensembles de groupes, afin de rassembler leurs actions. Mais là aussi, on se heurte à des limites : ces « super groupes » ne peuvent pas toujours être créés, et cela détruirait encore plus la faible vision du jeu de l'IA.
- Les problèmes (b), (c), (d) et (e) viennent du fait que seule une fraction infime des coups possibles sont calculés. Ils ne peuvent donc pas être résolus en continuant dans cette voie.

Au final, l'algorithme du min-max serait applicable à condition de n'analyser qu'une partie infiniment faible des possibilités, et avec une profondeur de 1. Mais que resterait-il de l'algorithme du min-max dans ce cas ? Il ne nous semble donc guère raisonnable d'appliquer un algorithme de type min-max à notre jeu.

Nouvelle approche

Principe

Puisque l'étude des coups possibles s'est avérée insatisfaisante, nous avons cherché à nous abstraire de ces problèmes de combinatoire. Pour cela, le monde réel, et en particulier l'organisation des armées constitue une source d'inspiration intéressante. Évidemment, les militaires ne calculent pas l'ensemble des coups possibles, étant donné que la notion même de coup n'a aucun sens en réalité. Les choix se font plutôt de la manière suivante :

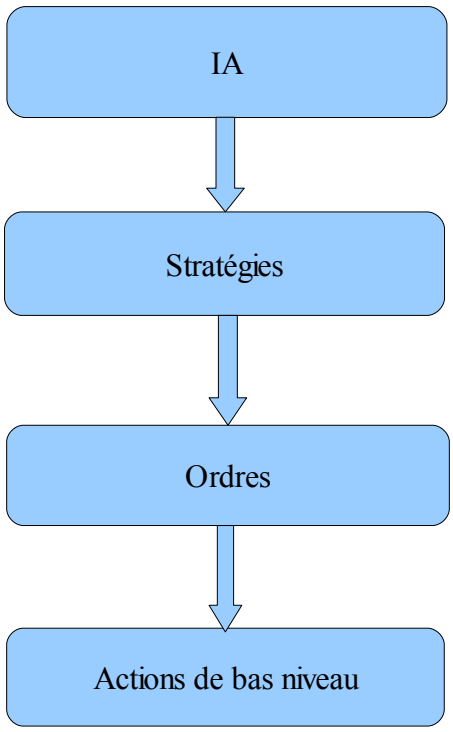
- Président : « Attaquez l'Italie ! »
- Général : « Le 30ème régiment d'infanterie prendra la ville de Turin. Pendant ce temps, le 28ème régiment protègera la frontière, etc. »
- Marcel, commandant du 30ème régiment : « Le troisième bataillon prendra le centre-ville à 15 heures. A ce moment, le cinquième bataillon détruira le pont machintruc, etc. »
- Robert, membre du troisième bataillon : « René, couvre-moi ! »

(note : ne pas tenir rigueur des termes employés; nous ne sommes en rien des spécialistes du domaine).

On remarque qu'au sommet de la hiérarchie, les ordres sont donnés sur la base d'une vue d'ensemble du théâtre d'opérations. Les tactiques sont du ressort de leurs subordonnés, tandis que les soldats les moins gradés se contentent d'exécuter sans discuter les ordres de leur hiérarchie. Ainsi, chaque élément de la hiérarchie ne traite qu'une quantité d'informations limitées.

Adaptation

Nous avons donc opté pour un algorithme de décision reposant sur le principe exposé dans la partie précédente, mais que nous avons bien sûr grandement adapté, du fait des spécificités de notre projet. Voici ses principales caractéristiques :



L'intelligence artificielle pour un joueur. Elle emploie une stratégie pour gagner, choisie en fonction de l'analyse du terrain.

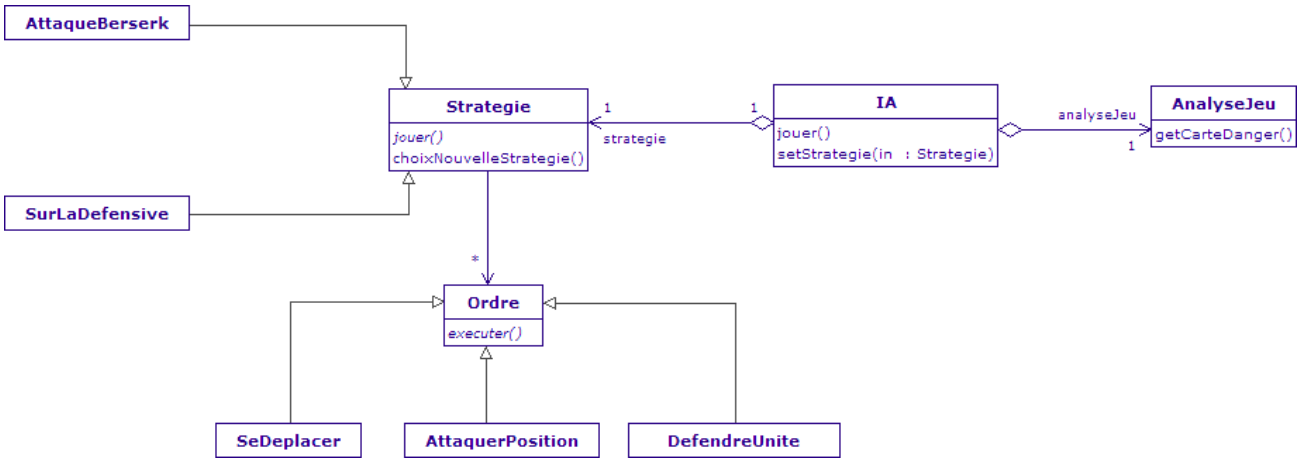
Définit le comportement général de l'IA (agressif ? Sur la défensive ? Etc.)

Ordres donnés à haut niveau (p.ex : que les quatre groupes d'infanterie se déplacent vers le capitaine), sans tenir compte des détails algorithmiques (pour que les unités ne se bloquent pas entre elles par exemple).

Les actions de base, mises en œuvre à bas niveau pour l'accomplissement d'un ordre. Il s'agit par exemple de l'ensemble des déplacements élémentaires à faire pour faire se déplacer un ensemble de groupes.

Implémentation

L'intelligence artificielle se base sur les classes suivantes :



Expliquons le fonctionnement de ces classes :

- Les classes IA, Strategie, et les classes dérivant de Strategie sont organisées suivant le design pattern État. Concrètement, à chaque tour de jeu, l'objet IA applique la stratégie courante, puis étudie la nécessité ou non d'un changement de stratégie, en fonction des données à sa disposition (encapsulées dans une instance de AnalyseJeu). Si cela s'avère nécessaire, elle est modifiée, ce qui se traduira par l'accomplissement d'objectifs différents au tour suivant.
- Les objets dérivant de Strategie choisissent **dans les grandes lignes** les actions à appliquer. Cela signifie qu'un objet Strategie peut ordonner à un ensemble de groupes d'unités d'attaquer une position, mais il ne décidera pas des actions à faire pour chaque unité, individuellement. Les objets Ordre sont là pour cela.
- Les objets Ordre ont à leur charge l'ensemble des actions à bas niveau nécessaires à l'accomplissement d'un objectif. A leur construction, ils ont connaissance d'un objectif (des coordonnées à atteindre par exemple), et de l'ensemble de groupes pour lesquels il s'applique. Le principal travail de l'objet Ordre est de coordonner les actions individuelles des groupes afin qu'elles forment un tout cohérent. Dans le cas du déplacement coordonné de 4 groupes par exemple, il s'agira de faire se déplacer les groupes à la vitesse du plus lent, pour qu'ils restent solidaires, et de faire avancer en premier les groupes les plus proches de l'objectif, sans quoi les groupes de derrière seraient gênés (voire bloqués) par ceux de devant.

Déroulement du travail, conclusion

Après une analyse commune du sujet, nous avons décidé que, avant de commencer l'intelligence artificielle, il nous fallait en premier lieu un jeu et une interface graphique fonctionnelle, et ce, afin de faciliter les tests lors du développement de l'IA.

Au niveau de l'organisation du travail, nous avons commencé par envisager la création d'un dépôt SVN, pour simplifier la mise en commun du travail. Cette option à rapidement été abandonnée pour cause de problème technique (ports bloqués par le service internet de la cité universitaire d'un des membres du groupe). Nous avons donc décidé de créer un forum pour discuter et nous échanger les fichiers du projet. Ce fut un moyen efficace de garder des traces de nos réflexions au fil de son avancement.

Rapidement, un des membres du groupe quitta le projet, nous laissant plus que 3 membres. Après avoir fini l'implémentation du jeu, nous avons réfléchi au système d'intelligence artificielle, et à la manière d'implémenter le min-max pour ce sujet-ci. Après plusieurs discussions sur les manières possibles de rendre cet algorithme applicable à notre jeu, nous avons finalement décidé de ne pas l'implémenter et de nous tourner vers un autre système d'intelligence artificielle. Nous avons alors également décidé de changer nos méthodes de travail, en nous réunissant souvent pour travailler ensemble, ce qui nous a permis de rapidement nous entraider en cas de problèmes, et avancer ainsi beaucoup plus vite.

Au final, notre application est fonctionnelle. Elle permet de jouer des parties joueur contre joueur, ou d'affronter l'intelligence artificielle.

Par manque de temps, nous n'avons cependant pas pu développer l'IA autant que nous l'aurions souhaité. Nous avons était obligés de nous limiter à une seule stratégie, basique, qui cherche avant tout à attaquer pour finir la partie le plus vite possible. Elle lancera donc la plupart de ses forces sur les ennemies en limitant la défense au strict minimum, et ceci, quelle que soit la situation.

Les possibilités d'amélioration sont donc nombreuses. Dans le jeu en lui-même tout d'abord, l'interface pourrait être grandement améliorée, puisque nous nous sommes contentés du minimum nécessaire aux tests de l'intelligence artificielle. L'ajout de menus et de feedback plus poussé sur les actions de l'utilisateur serait obligatoire pour faire de notre maquette un jeu à part entière. L'intelligence artificielle est elle aussi facilement améliorable, en ajoutant d'autres stratégies selon les situations qui se présentent.