Projet Logiciel Transversal

Antoine Delavoypierre – Nicolas Leteinturier

Table des matières

1 Objectif

1.1 Présentation générale

Le but de ce projet est de programmer un jeu basé sur l'archétype du Risk. Le but du jeu sera de respecter le cahier des charges :

-être un jeu multijoueur afin de pouvoir avoir des applications serveur et programmer une IA
-le jeu doit inclure un IA avec 3 niveaux de difficultés
-le jeu doit être un jeu d'état, pour stocker les données du jeu à tout moment
-le jeu doit pouvoir être mis en réseau.

Le Risk est une stratégie tour par tour qui permet à plusieurs joueurs de jouer les uns contre les autres. Ceci inclut la possibilité de jouer contre un ordinateur. Le but du jeu est de développer une stratégie militaire pour gérer les troupes et prendre le contrôle du monde.

1.2 Règles du jeu

Bien que basée sur les règles de l'archétype, cette version du jeu va prendre quelques libertés.
Par exemple, la mise en place de plusieurs types d'unités : plus d'unités défensives, et plus d'unités offensives qui permettent d'avoir un coefficient multiplicateur sur les jets de dé en fonction de la situation.
Au début du jeu :

-Une carte est générée composée de N tuiles
-La carte est divisée entre les joueurs du jeu.
-Le même nombre d'unités militaires est assigné à chaque joueur pour retourner sur son territoire.

Dans le jeu, chaque joueur joue tour à tour. Les visites guidées sont les suivantes :

-Renforts :
en fonction du nombre de territoires qu'il possède dans la journée, il reçoit des renforts à distribuer sur les territoires de son choix
-Technologie :
Si le jour a assez de points d'expérience, il peut investir ses points dans une technologie pour gagner un bonus.
-Attack :
si un joueur a plus d'une unité sur un territoire, il peut attaquer un territoire voisin. Le combat se déroulera en fonction du jet des dés en appliquant les bonus technologiques.
-Déménager :
Pour terminer le tour, le joueur peut déplacer des unités du territoire A au territoire B si les territoires sont connectés par le territoire du joueur.

Fin de la partie :

-Un joueur perd s'il n'a plus de territoires.
-Un joueur gagne s'il possède 90% des territoires.

1.3 Conception Logiciel

Afin de pouvoir programmer le jeu. La feuille de sprite a été créée. Une carte du monde a été divisée en 14 territoires (le nombre était limité pour simplifier le jeu), chaque territoire a été isolé afin de pouvoir le traiter individuellement au niveau des couleurs. En effet, la couleur du territoire sur l'interface graphique sera déterminée par le joueur propriétaire du territoire. Afin d'éviter d'avoir à préparer la feuille de sprites de chaque couleur. La carte mondiale sera une superposition de territoires colorés individuellement grâce à SFLM.

2 Description et conception des états

2.1 Description des états

L'état du jeu à un moment donné est défini par la possession de chaque territoire par un seul joueur, et par le nombre d'unités que ce joueur possède dans chaque territoire. Ainsi que l'ordre dans lequel les joueurs passent, et le joueur actuel qui doit jouer. Ainsi, à tout moment, il est possible d'interrompre et de reprendre le jeu uniquement grâce à :

-la liste des joueurs.
la liste des pays avec le nombre d'unités dessus
-le joueur actuel

2.2 Conception logiciel

Pour définir ces états, nous avons défini plusieurs classes.

Tout d'abord la classe GameState, qui contient le vecteur des joueurs, c'est-à-dire l'ordre de passage, le vecteur des pays (ces vecteurs contiendront des pointeurs vers les objets définis dans les classes Player et Country), et le joueur actuel qui doit jouer.

La classe GameState gère l'ajout et la suppression de joueurs soit lors de la création du jeu, soit plus tard lors d'une défaite gérée par le moteur de jeu. Une autre méthode dans la classe est de déplacer les unités.

La classe Player définie par un identifiant unique.

Les méthodes de cette classe vous permettent de gérer les renforts reçus par un joueur en fonction du nombre de territoires dont il dispose.

La catégorie Pays contient des renseignements sur chaque territoire. Comme le nombre d'unités et leurs types dans un territoire (comme vecteur de type d'unité). Afin de pouvoir attaquer les pays voisins pour chaque territoire, un vecteur contenant les pointeurs vers les pays voisins et les pays définis. Pendant la phase d'attaque, le moteur n'autorise que les attaques contre les territoires ennemis voisins. Ainsi, le pays doit également contenir des informations sur le propriétaire du pays.

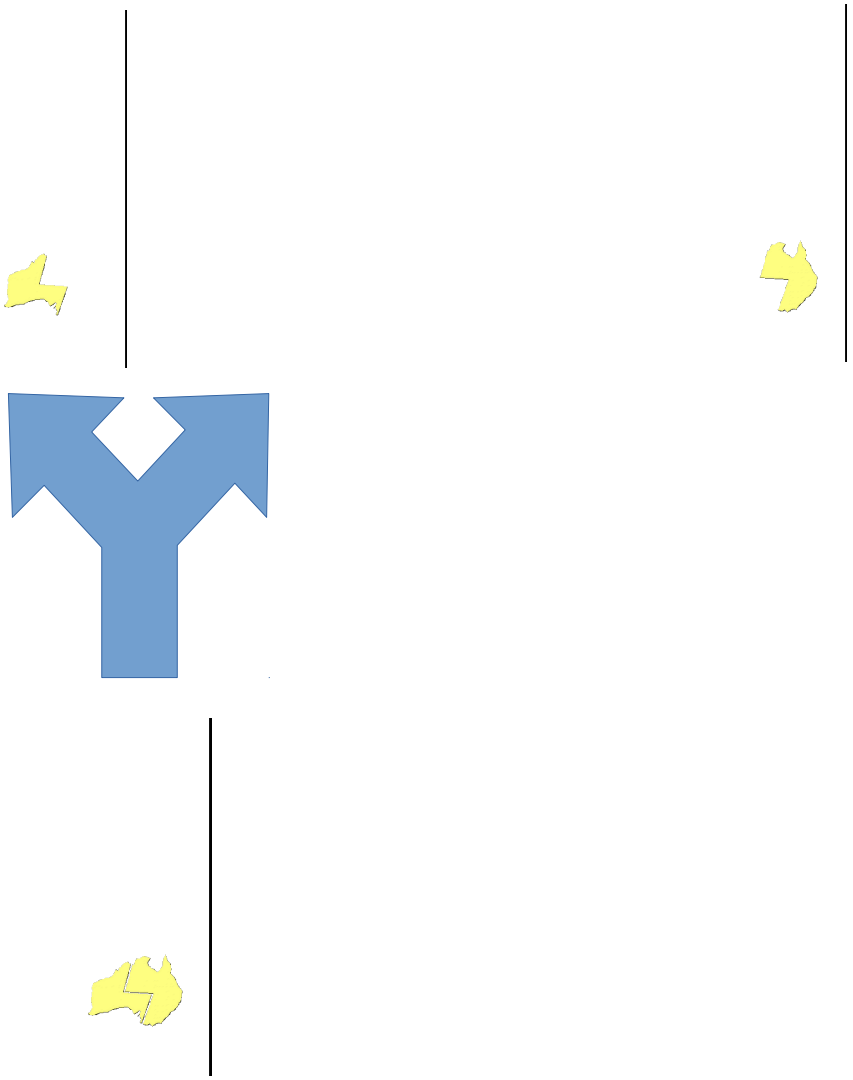
Les méthodes des classes de pays sont utilisées pour gérer les unités situées sur le territoire. Soit ajouter des unités en cas de renforcement. Eliminer les unités en cas de perte en attaque ou en défense. Afin d'initialiser la carte et la liste des voisins, une méthode est utilisée pour ajouter des territoires voisins.

La classe Unité et Type pour gérer les différents types d'unités que le joueur peut choisir pour développer une stratégie.

La Classe d'Action vous permet de gérer toutes les actions / escalade du jeu, c'est à dire les attaques, les mouvements, les renforts..... Lors d'une attaque, par exemple, les troupes des deux territoires impliqués sont temporairement retirées des deux territoires pour être traitées dans la classe d'action. S'ils sont victorieux, ils seront ainsi placés à la fin dans le territoire où ils sont censés aller. S'ils sont détruits, une méthode les éliminera. Ainsi, la classe Action vous permet de gérer quel type d'action sera utilisé pour l'état du jeu. Les autres classes comme Attaque, Mouvement, Renforcement et Initialisation hériteront de la classe Action.

2.3 Conception logiciel : extension pour le rendu

Pour le rendu, la classe de pays est définie par un identifiant associé à chaque territoire. Le rendu général sera défini par un superposition des différents territoires. Pour un territoire donné, il est également nécessaire d'établir un code de couleur en fonction du propriétaire du territoire. Ainsi, l'utilisation d'une superposition de masque indépendante permet de donner plus facilement une couleur à chaque territoire. Voir l'exemple de sprite dans le dossier ressources. Nous avons ici l'exemple de l'Océanie qui est divisée en deux régions terrifiantes. Leurs positions respectives sur la carte.png permettent donc une superposition. Ensuite, en appliquant un filtre de couleur grâce au rendu, nous pouvons avoir l'interface.



2.4 Conception logiciel : extension pour le moteur de jeu

Les classes ont été établies pour faciliter l'extension du moteur de jeu. La définition d'un territoire

par un identifiant et une liste de pays voisins permet au moteur d'avoir facilement accès à quel territoire un joueur peut attaquer. De plus, le fait d'avoir un accès rapide pour chaque territoire au nombre d'unités qui s'y trouvent et au propriétaire permettra d'avoir accès à plus d'information à laquelle l'IA aura accès plus tard dans la programmation de l'IA.

2.5 Ressources

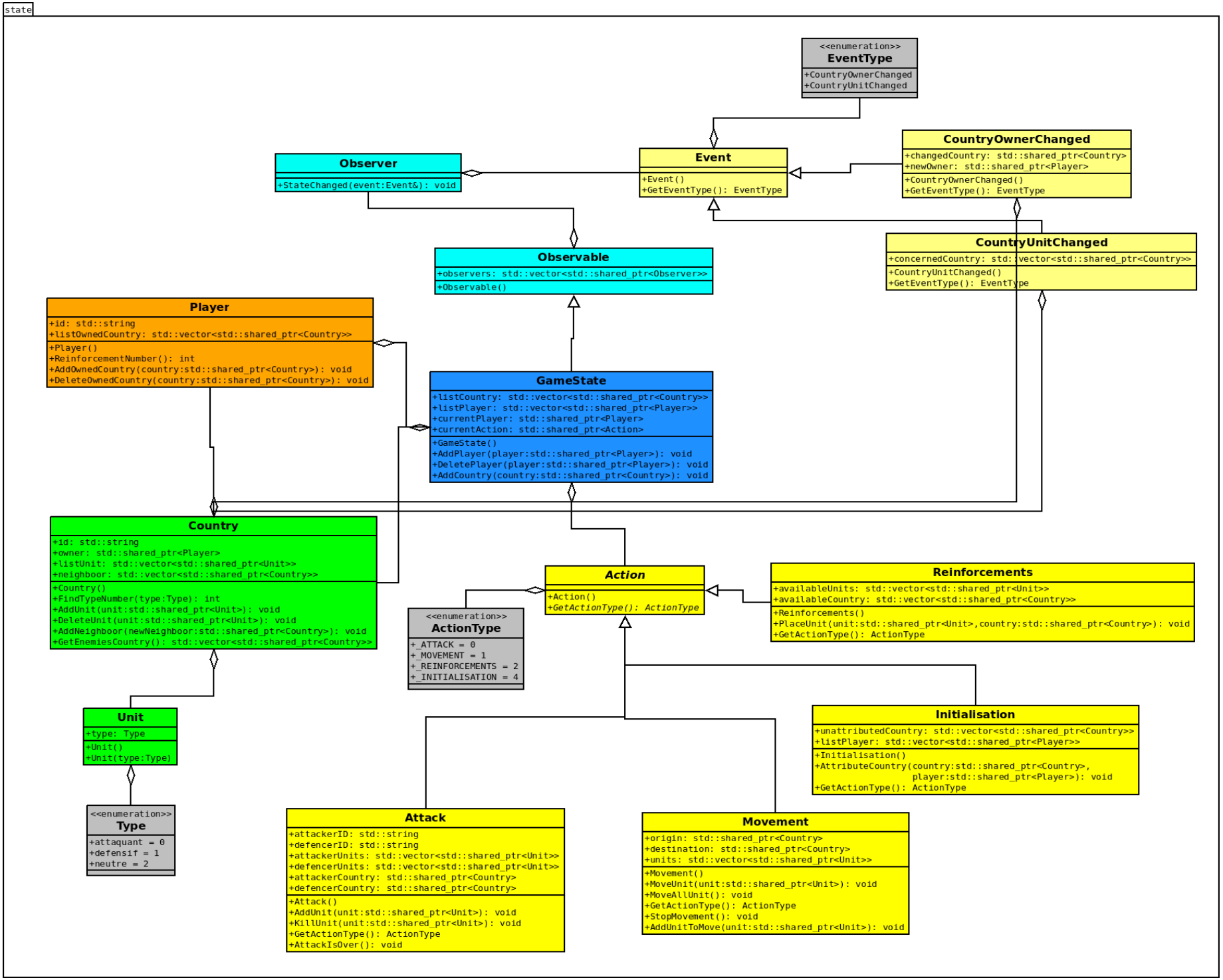


Illustration 1: Diagramme des classes d'état

3 Rendu : Stratégie et Conception

Afin d'obtenir un rendu, nous avons décidé d'organiser notre diagramme de classes pour le rendu comme suit :

Une classe Scène gère toutes les données et la synchronisation en ayant accès à l'état du jeu et aux autres classes. Scène pourra choisir quelle partie du rendu nécessite une mise à jour.

Les deux classes qu'ils peuvent utiliser sont WorldMap et UnitRepresentation.

La classe WorldMap vous permet de gérer tout ce qui concerne les pays. C'est-à-dire la représentation des différents territoires. La classe importe donc en premier lieu tous les sprites de pays, ce qui permet de construire le monde. Une fois le monde créé, cette classe vous permet également de colorier chaque pays individuellement afin de visualiser quel joueur possède le territoire.

La classe UnitRepresentation de sa classification est responsable d'extraire du jeu le nombre d'unités dans chaque territoire afin de le représenter sur la carte. Pour ce faire, les valeurs des numéros d'unité doivent être placées au bon endroit.

Toutes les positions et adresses des sprites sont stockées dans un fichier "define.h" afin de ne pas avoir un code trop lourd avec des tables au milieu du code.

La classe Layer permet de stocker tous les sprites et textures des deux classes précédentes.

3.1 Stratégie de rendu d'un état

Les classes décrites ci-dessus permettent de traiter de manière indépendante le rendu des territoires et le rendu des unités, ceci est important car le jeu est divisé en plusieurs phases. Y compris les phases de renforcement, lorsque les territoires ne changent pas. Seul le nombre d'unités change. Ainsi, en faisant cela, nous ne pouvons mettre à jour que la partie du script qui nous intéresse. Cela nous permet d'imiter sa complexité en termes de temps de calcul et de flexibilité.

3.2 Conception logiciel

Afin de donner accès aux informations nécessaires aux bonnes classes les deux classes WorldMap et UnitRepresentation héritent toutes les deux de Layer. Layer ayant accès à certaines données de l’état de jeu. Ceci permet de pouvoir représenter toutes les informations nécessaires à l’utilisateur pour une prise de décision. La derniere classe GraphicElement est une classe pour traiter chaque element de Layer independement. Ainsi on peut pour chaque pays modifier de facon independante des autres sa couleur en fonction du propriétaire grace à un simple methode.
En parcourant la liste des pays issue de Scène la classe UnitRepresentation est capable de retrouver et representer le nombre de chaque type d’unités sur chaque territoire.

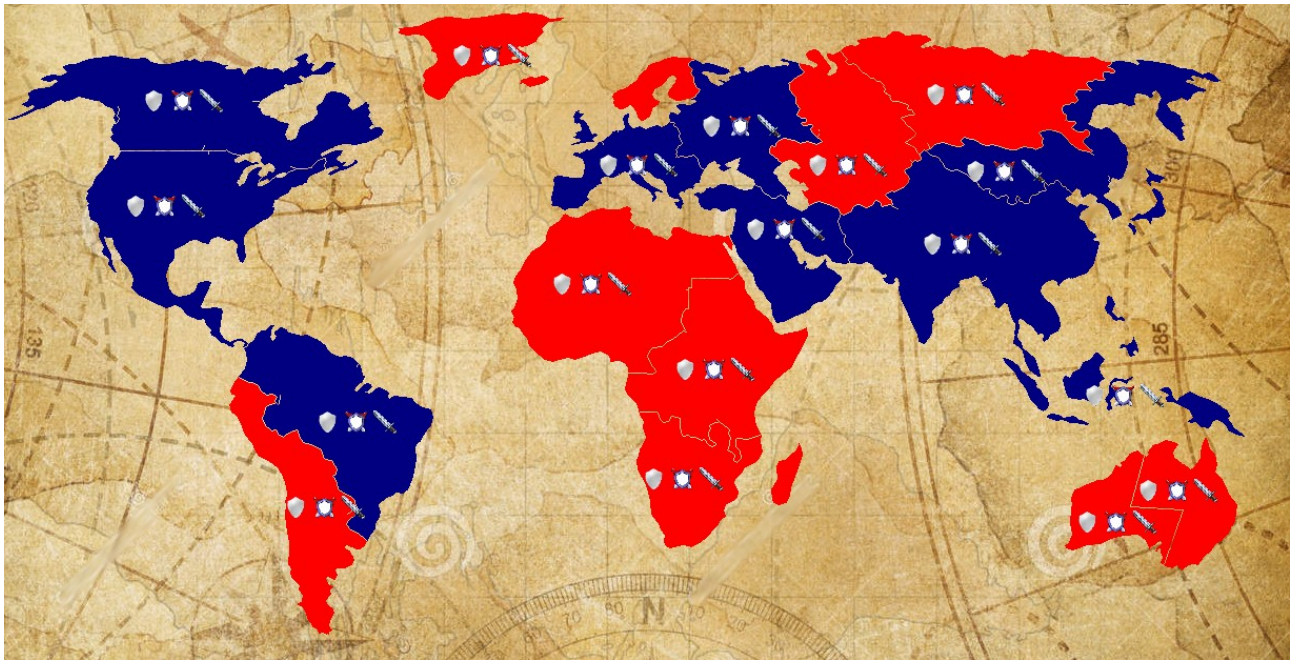
3.3 Conception logiciel : extension pour les animations

3.4 Ressources

Pour cette partie, nous avons ajouté de nouvelles ressources pour optimiser le rendu. Dans une première partie, nous avons ajouté une représentation pour les différents types d'unités. Qui est représenté au centre de chaque pays. Nous avons donc dû déterminer la position de chaque pays afin de positionner ce rendu. Les positions sont enregistrées dans le fichier define.h. Une police de caractères a été ajoutée pour un look plus esthétique. Pour les différents territoires, nous avons

utilisé les territoires décrits dans la section précédente. Avec chaque territoire dans un fichier png. La carte est donc une suppression de tous les fichiers traités individuellement par le moteur de rendu.

3.5 Exemple de rendu



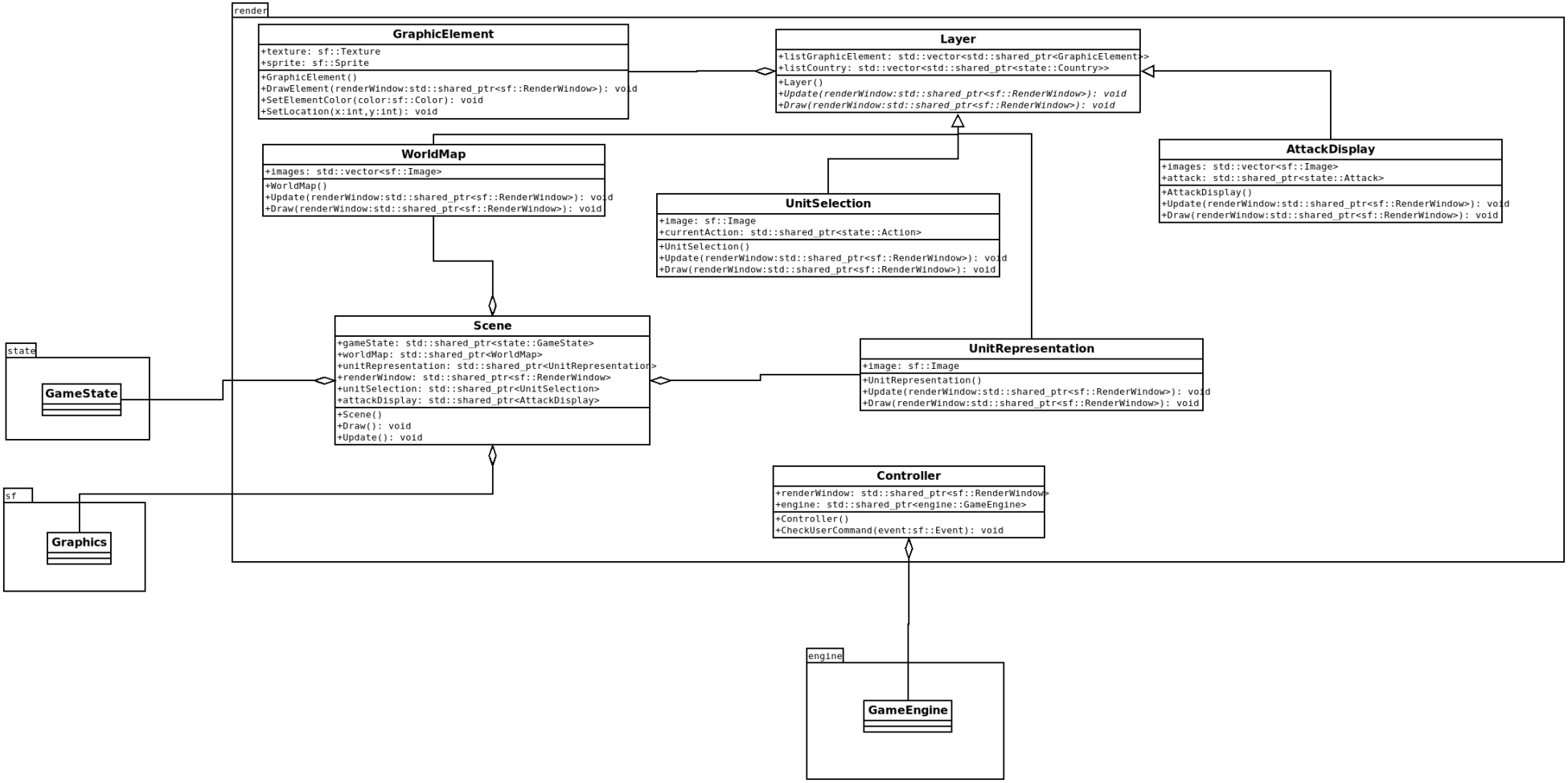


Illustration 2: Diagramme de classes pour le rendu

4 Règles de changement d'états et moteur de jeu

Dans cette section, il faut présenter les événements qui peuvent faire passer d'un état à un autre. Il faut également décrire les aspects lié au temps, comme la chronologie des événements et les aspects de synchronisation. Une fois ceci présenté, on propose une conception logiciel pour pouvoir mettre en œuvre ces règles, autrement dit le moteur de jeu.

4.1 Horloge globale

Le moteur de jeu ajustera les états de jeu : Renforts, Attaque, Mouvement par une gestion en cliquant sur le rendu graphique ou par un tour en appuyant sur la barre d'espace.
Ainsi le jeu passera séquentiellement chaque phase de jeu pour chaque joueur.

4.2 Changements extérieurs

Afin de définir chaque phase de jeu, le moteur de jeu traitera chaque phase de jeu dans un module qui aura accès à l'état du jeu. Afin de gérer chaque phase de jeu indépendamment des interfaces spécifiques à chaque phase de jeu ont été ajoutées au rendu graphique pour choisir sur quel pays l'action doit être effectuée, puis le nombre d'unités impliquées. Les interactions avec l'utilisateur se font via ces interfaces spécialisé pour chaque phase de jeu.

4.3 Changements autonomes

Comme le jeu est rythmé par des phase de jeu qui se suive de façon séquentielles :
pour chaque joueur nous avons les phase suivante :

-Renforts
-Attaques
-Mouvement

Ainsi il est possible pour le moteur de jeu de changer de façon autonome les phases de jeu une fois que le jour à fini son action. De même le joueur peut décider de passer son tour s'il décide de ne pas vouloir attaquer par exemple. Pour cela il suffit au joueur de simplement appuyer sur la barre espace. Une fois que le joueur à passé chaque phase le moteur change de joueur. Pour cela il à accès à l'état de jeu et peut changer le currentPlayer

4.4 Conception logiciel

Afin d'interagir avec l'utilisateur le moteur dois etre capable de prendre des commandes souris ainsi que des commandes venu du clavier (comme appuyer sur la barre espace pour confirmer quelque chose). Nous avons donc implémenté un classe « commande » pour gérer ces entrées venant de l'utilisateur.
La classe « moteur » est la classe principale qui gère chaque phase de jeu pour ce faire nous avons décidé d'écrire une méthode pour chaque phase. Ainsi le moteur peut appeler l'état de jeu pour avoir accès aux information, mais egalement faire les changement necessaires :
- changer le propriétaire d'un pays par exemple.
- changer de joueur
-ajouter des unités...(par exemple)
Chaque fonction de phase de jeu interagit également avec le rendu graphique, car nous avons décidé

d’ajouter des sprites pour chaque phase avec un menu pop up pour la sélection des unités.
Avec cette architecture et le traitement de chaque phase dans une méthode il est plus facile d’interagir avec le moteur de jeu qui n’a que à parcourir les actions à faire.

4.5 Conception logiciel : extension pour l'IA

Afin de pouvoir jouer contre l’ordinateur nous avons généré un nouveau diagramme de classe pour l’intelligence artificiel. Pour commencer nous avons programmé un bot contre qui jouer qui fait ses actions de façon totalement aléatoire parmi ses actions possible.
Pour ce faire il doit connaître à tout moment les pays qu’il possède, les unités sur chaque pays : c’est dire toute les information qu’un joueur humain verrait sur le rendu graphique. Puis il fera ces actions de façon aléatoire : il choisira donc quel pays attaquer dans la liste de ses voisins, il choisira son nombre de troupe de façon aléatoire pour une attaque et pour un mouvement.

4.6 Conception logiciel : extension pour la parallélisation

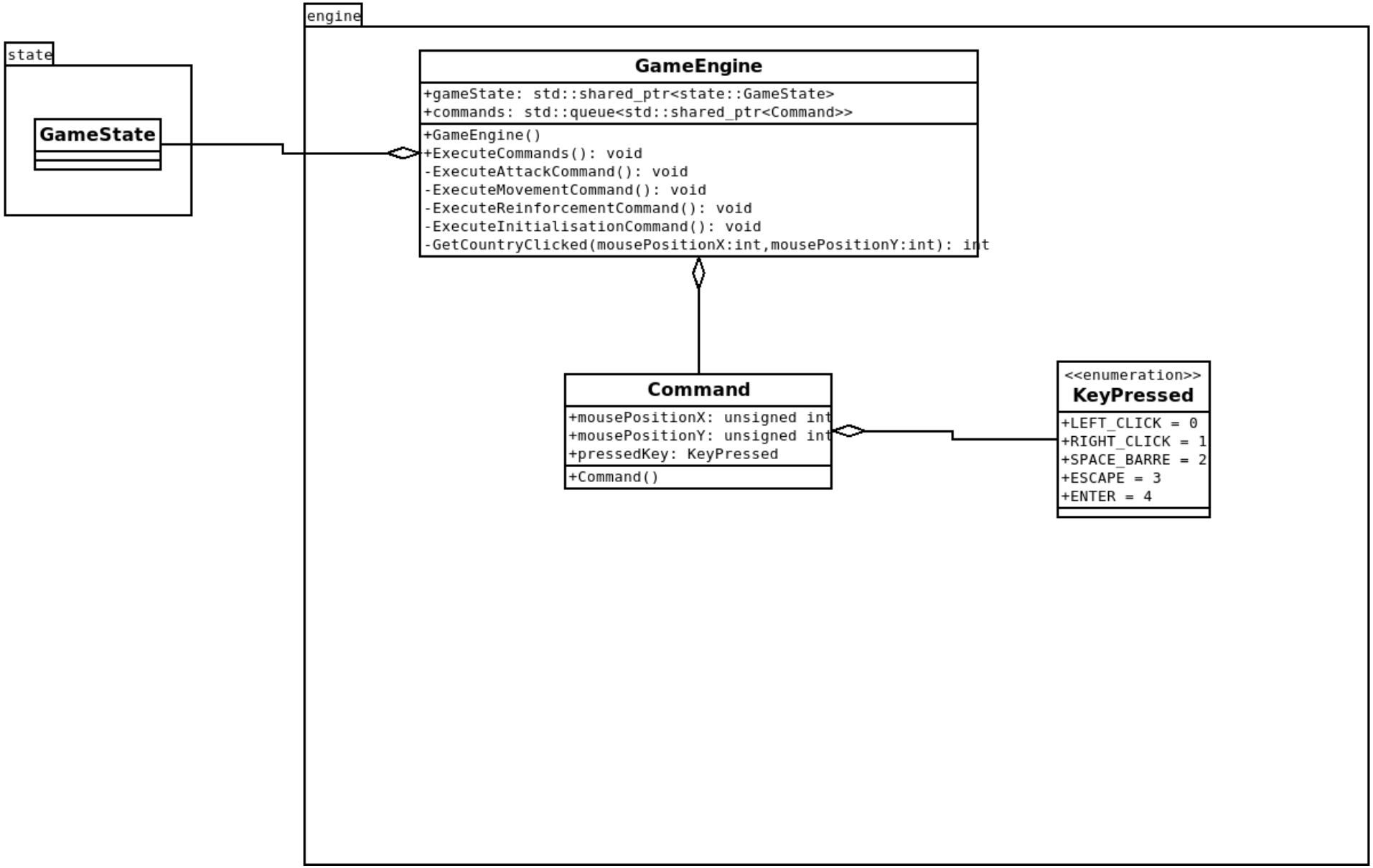


Illustration 3: Diagrammes des classes pour le moteur de jeu

5 Intelligence Artificielle

Cette section est dédiée aux stratégies et outils développés pour créer un joueur artificiel. Ce robot doit utiliser les mêmes commandes qu'un joueur humain, ie utiliser les mêmes actions/ordres que ceux produit par le clavier ou la souris. Le robot ne doit pas avoir accès à plus information qu'un joueur humain. Comme pour les autres sections, commencez par présenter la stratégie, puis la conception logicielle.

5.1 Stratégies

Afin de programmer l'intelligence artificielle à n'importe quel niveau de jeu, l'IA doit être capable d'envoyer des instructions aux moteurs de jeu, comme le ferait un joueur humain. Ensuite, en fonction du niveau de l'IA, elle aura besoin de plus d'informations afin de pouvoir calculer les choix optimaux.

Afin de gérer les IA et les jours dans le GameState, nous ajoutons un booleen dans la classe du joueur qui détermine si le joueur est un IA ou non. S'il s'agit d'un IA, le moteur jouera l'IA en appelant le code AI demandé. Sinon, le moteur attendra les instructions du joueur.

5.1.1 Intelligence minimale

Pour l'intelligence minimale, on a fait une IA aléatoire. Cette IA n'aura donc besoin que d'avoir accès à l'état du jeu pour sélectionner l'action en cours : Renforcement, Attaque, Mouvement.
Ainsi, pour chaque tour, l'IA sait quelles options choisir : par exemple, sélectionner un pays qui lui appartient, de sorte que dans la liste des pays qui lui appartiennent, elle peut choisir au hasard un pays. Ensuite, l'IA envoie une demande de commande au moteur de jeu pour qu'il joue son tour. La commande se fait sous la forme d'un clic avec des coordonnées choisies dans un tableau de "define.h" en fonction du pays choisi.

5.1.2 Intelligence basée sur des heuristiques

Pour l'intelligence heuristique, nous avons dû mettre en place des procédures que l'ordinateur doit appliquer pour savoir comment agir et avoir de meilleures performances que l'intelligence minimale.
C'est pourquoi nous avons traité chaque phase de manière indépendante :

Pour les renforts, nous avons cherché à renforcer les territoires avec les voisins les plus ennemis. Il s'agit de renforcer les frontières potentiellement hostiles et non d'accumuler des troupes sur un territoire entouré de territoires alliés.

Pour l'attaque, nous avons cherché parmi les territoires de l'attaquant quels territoires ont le plus d'unités offensives (Ainsi il pourra avoir plus d'impact lors d'une attaque sur un de ses voisins). Ainsi, le territoire attaquant est sélectionné et nous pouvons choisir parmi les territoires voisins le pays à attaquer : celui qui possède le moins d'unités défensives.

Pour les mouvements, nous avons décidé de déplacer les unités du pays ayant le moins de voisins ennemis vers le pays ayant le plus de voisins ennemis. Ainsi, le renseignement pourra renforcer les fronts qui risquent le plus d'être attaqués.

5.1.3 Intelligence basée sur les arbres de recherche

Pour les renseignements basés sur un arbre de recherche, nous avons mis en place un système de notation pour chaque situation. Chaque pays a un score de vulnérabilité et un score d'attaque.
Le score de vulnérabilité est déterminé par le nombre de voisins ennemis et le nombre de troupes offensives ennemies aux frontières. Le score d'attaque d'un pays est déterminé par le nombre de troupes offensives sur le pays et le nombre de troupes défensives sur les voisins ennemis qui en ont au moins un. Ainsi, pour chaque phase de jeu, il est possible de tester le score de chaque pays. En additionnant tous les scores, nous pouvons obtenir un score global pour l'intelligence en question. En essayant les mouvements possibles pour chaque cas, il est possible de trouver le mouvement optimal en utilisant un algorithme de maximisation du score.

5.2 Conception logiciel

Pour la conception logicielle des différents IA, nous avons appliqué le diagramme de classes si joint, créer une classe pour chaque type d'IA. Ensuite, en communication avec l'état de la situation, l'IA sait dans quelle phase de jeu elle se trouve et si elle doit agir comme un renforcement, une attaque ou un mouvement.

Nous traitons chaque IA dans la main en testant un élément de l'objet PLAYER : IsAnie
S'il s'agit d'une IA, nous traitons l'IA en question en appelant le code correspondant à son niveau de difficulté.

5.3 Conception logiciel : extension pour l'IA composée

5.4 Conception logiciel : extension pour IA avancée

L'IA avancée utilisera l'arbre de recherche avec le score de chaque action pour minimiser la probabilité de pertes et maximiser la probabilité de gains. Ainsi, l'IA pourra prendre des décisions pour sélectionner la meilleure action possible.

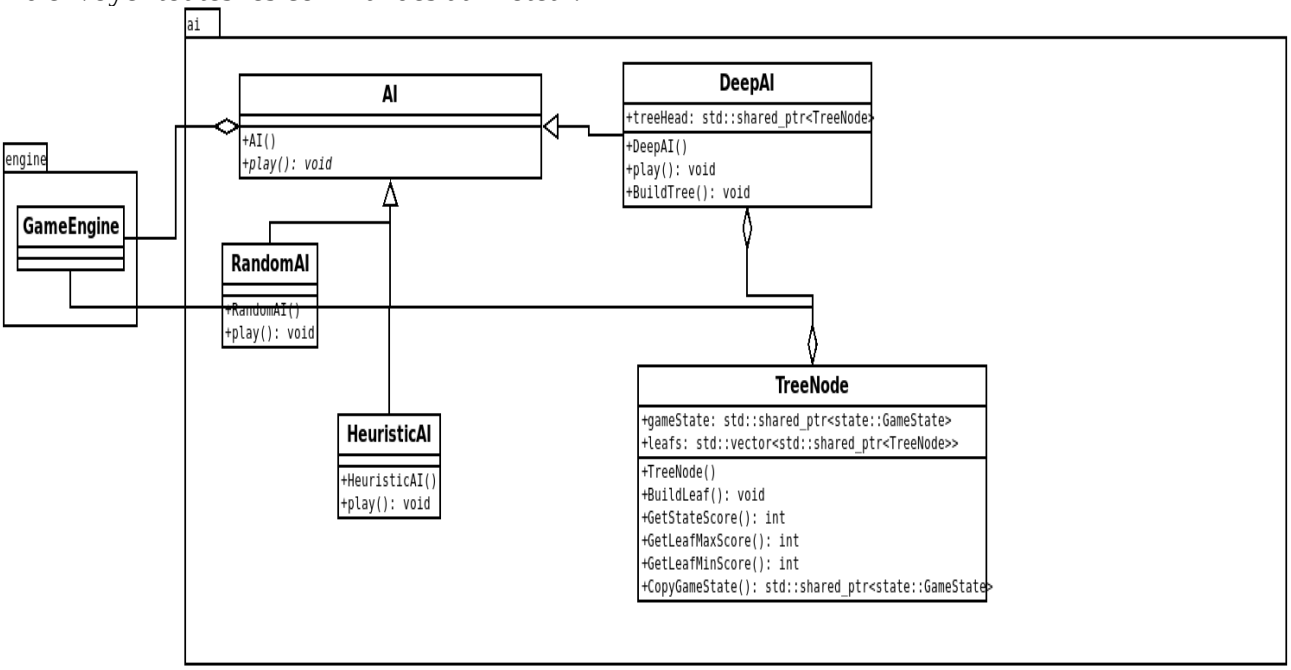
Afin de sélectionner les meilleures actions possibles, l'IA avancée peut également tester les coups. Grâce à la fonction de Rollback, si le coup est mauvais, elle peut remettre le jeu dans un état antérieur. Il est donc même possible de comparer directement deux coups.

Cette fonction de Rollback est également intéressante pour sauvegarder un état de jeu donné afin de reprendre une partie plus tard. Il est ainsi possible de sauvegarder chaque coup dans un fichier texte pour le réimporter plus tard par une ligne de commande. Nous avons testé dans le script de chaque 100 coups de recharger l'état de jeu précédent et nous pouvons voir que le moteur de jeu supporte cette fonctionnalité.

5.5 Conception logiciel : extension pour la parallélisation

Tel que les IA sont programmer elles vont se décider pour une suite d'action à faire :
-choisir un pays de départ
-Sélectionnez un pays de destination pour une attaque ou un mouvement...

Une fois que toutes les actions ont été sélectionnées et stockées dans une liste, elles seront envoyées au moteur de jeu pour être traitées.
Donc pour implémenter le multi-threading nous allons essayer de paralléliser les différentes commandes. Ainsi, dès qu'un pays d'origine est choisi par exemple, il est possible d'envoyer la commande déjà au moteur de jeu, en parallèle l'IA pourra calculer sa prochaine commande pour gagner du temps. Il ne sera donc pas nécessaire d'attendre que tous les calculs soient terminés avant d'envoyer toutes les commandes au moteur.



6 Modularisation

Cette section se concentre sur la répartition des différents modules du jeu dans différents processus. Deux niveaux doivent être considérés. Le premier est la répartition des modules sur différents threads. Notons bien que ce qui est attendu est un parallélisation maximale des traitements: il faut bien démontrer que l'intersection des processus communs ou bloquant est minimale. Le deuxième niveau est la répartition des modules sur différentes machines, via une interface réseau. Dans tous les cas, motivez vos choix, et indiquez également les latences qui en résulte.

6.1 Organisation des modules

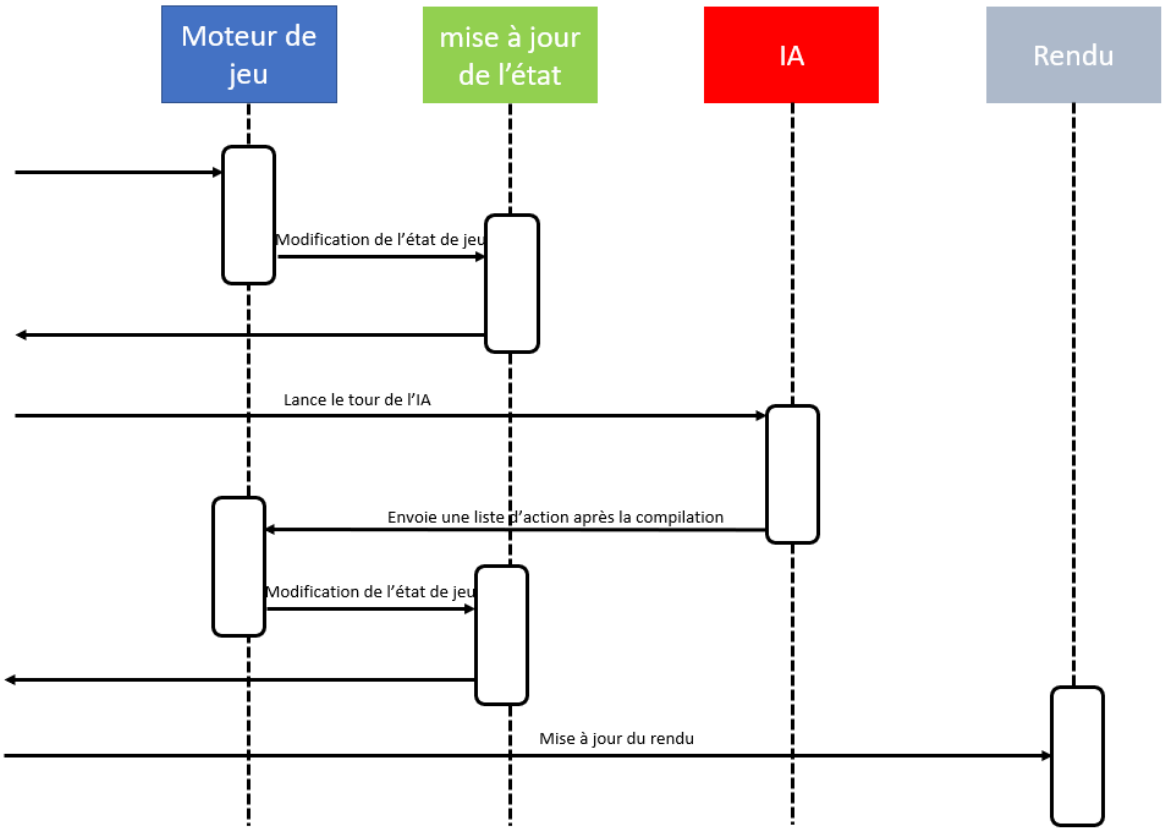
Nous avons organisé les modules comme décrit ci-dessus avec la partie moteur de jeu qui gère tout consiste en les règles du jeu et les tests que le joueur demande, ou l'IA n'est pas en violation des règles. La partie de l'état du jeu qui va modifier et stocker tous les changements dans l'état du jeu. Ensuite l'état du jeu peut être visualisé par le module de rendu graphique qui montrera pour chaque pays son propriétaire et le nombre d'unités qu'il possède.

L'IA est traitée de manière indépendante car elle utilise les données de l'état de jeu pour prendre des décisions. Ainsi les décisions sont prises sous la forme de commandes envoyées au moteur. Chaque IA crée une liste de commandes qu'elle envoie au moteur à la fin du traitement.

Pour améliorer ce processus, nous pouvons utiliser le multithreding pour mieux optimiser l'utilisation du CPU. (décrit dans la partie suivante)

6.1.1 Répartition sur différents threads

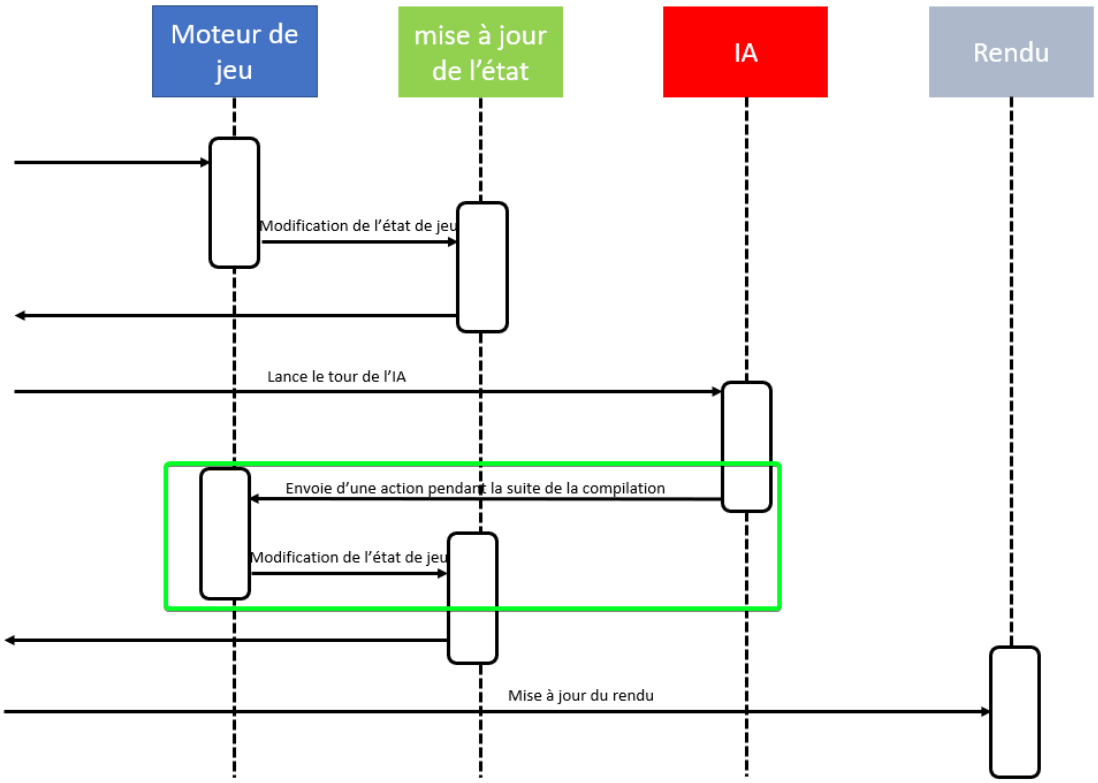
Nous avons essayé de représenter les différentes interactions entre les modules dans un graphique pour savoir comment mettre en œuvre le multithreading de manière efficace. En effet, une approche naïve serait de mettre chaque module dans un thread et de mettre des verrous pour organiser les différentes parties (voir schéma de séquence).



Mais vous pouvez voir sur la figure que pour mettre à jour l'état du jeu, vous devez passer par le moteur de jeu. Et l'IA et le rendu doivent être à jour. Donc une grande partie de l'interaction entre les modules doit être faite de manière séquentielle. Mettre chaque module dans un thread ne fera pas gagner du temps.

Nous avons donc cherché des moyens d'implémenter le multithreading de manière efficace. En regardant le diagramme de séquence, nous pouvons voir qu'il est possible d'implémenter le multithreading dans l'interaction entre l'IA et le moteur de jeu. En effet, en traitant chaque commande indépendamment et en les exécutant dans des threads différents, il est possible de gagner du temps. Pendant que l'IA calcule sa prochaine commande, le moteur peut déjà mettre à jour l'état du jeu.

Il est donc intéressant de mettre le moteur dans un thread isolé pour qu'il puisse toujours mettre à jours les informations de l'état de jeu. Puis en particulier dans l'IA profonde il est important d'implémenter du multithreading car en parcourant l'arbre elle doit faire beaucoup de calculs qui sont parallélisable.



6.1.2 Répartition sur différentes machines

6.2 Conception logiciel

Pour mettre en place une solution logicielle, nous avons donc traité chaque commande dans un thread qui est ensuite envoyé à un thread du moteur de jeu. Ceci est particulièrement important pour l'IA Heuristique et Avancée car elles prennent plus de temps à calculer. Il est donc possible de gagner plus de temps avec cette méthode. Le plus important est donc de traiter le moteur dans un thread dédié car c'est le moteur qui orchestre tous les autres modules.

6.3 Conception logiciel : extension réseau

Afin de préparer l'étape de mise en réseaux nous avons également implémenté un fonction de codage et de décodage des différentes commandes du jeu en format JSON. Ainsi il est possible de stocker toutes les commandes de l'utilisateur ou de l'IA dans un fichier JSON ce qui facilitera la communication pour le jeu en réseau. Les commandes sont donc sauvegardé dans un fichier commandes.json

6.4 Conception logiciel : client Android

Illustration 4: Diagramme de classes pour la modularisation

Voir le diagramme de séquence au dessus