

Projet Logiciel Transversal

Antoine Delavoy pierre – Nicolas Leteinturier

Table des matières

1 Objectif.....	3
1.1 Présentation générale.....	3
1.2 Règles du jeu.....	3
1.3 Conception Logiciel.....	3
2 Description et conception des états.....	4
2.1 Description des états.....	4
2.2 Conception logiciel.....	4
2.3 Conception logiciel : extension pour le rendu.....	4
2.4 Conception logiciel : extension pour le moteur de jeu.....	4
2.5 Ressources.....	4
3 Rendu : Stratégie et Conception.....	6
3.1 Stratégie de rendu d'un état.....	6
3.2 Conception logiciel.....	6
3.3 Conception logiciel : extension pour les animations.....	6
3.4 Ressources.....	6
3.5 Exemple de rendu.....	6
4 Règles de changement d'états et moteur de jeu.....	8
4.1 Horloge globale.....	8
4.2 Changements extérieurs.....	8
4.3 Changements autonomes.....	8
4.4 Conception logiciel.....	8
4.5 Conception logiciel : extension pour l'IA.....	8
4.6 Conception logiciel : extension pour la parallélisation.....	8
5 Intelligence Artificielle.....	10
5.1 Stratégies.....	10
5.1.1 Intelligence minimale.....	10
5.1.2 Intelligence basée sur des heuristiques.....	10
5.1.3 Intelligence basée sur les arbres de recherche.....	10
5.2 Conception logiciel.....	10
5.3 Conception logiciel : extension pour l'IA composée.....	10
5.4 Conception logiciel : extension pour IA avancée.....	10
5.5 Conception logiciel : extension pour la parallélisation.....	10
6 Modularisation.....	11
6.1 Organisation des modules.....	11
6.1.1 Répartition sur différents threads.....	11
6.1.2 Répartition sur différentes machines.....	11
6.2 Conception logiciel.....	11
6.3 Conception logiciel : extension réseau.....	11
6.4 Conception logiciel : client Android.....	11

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.

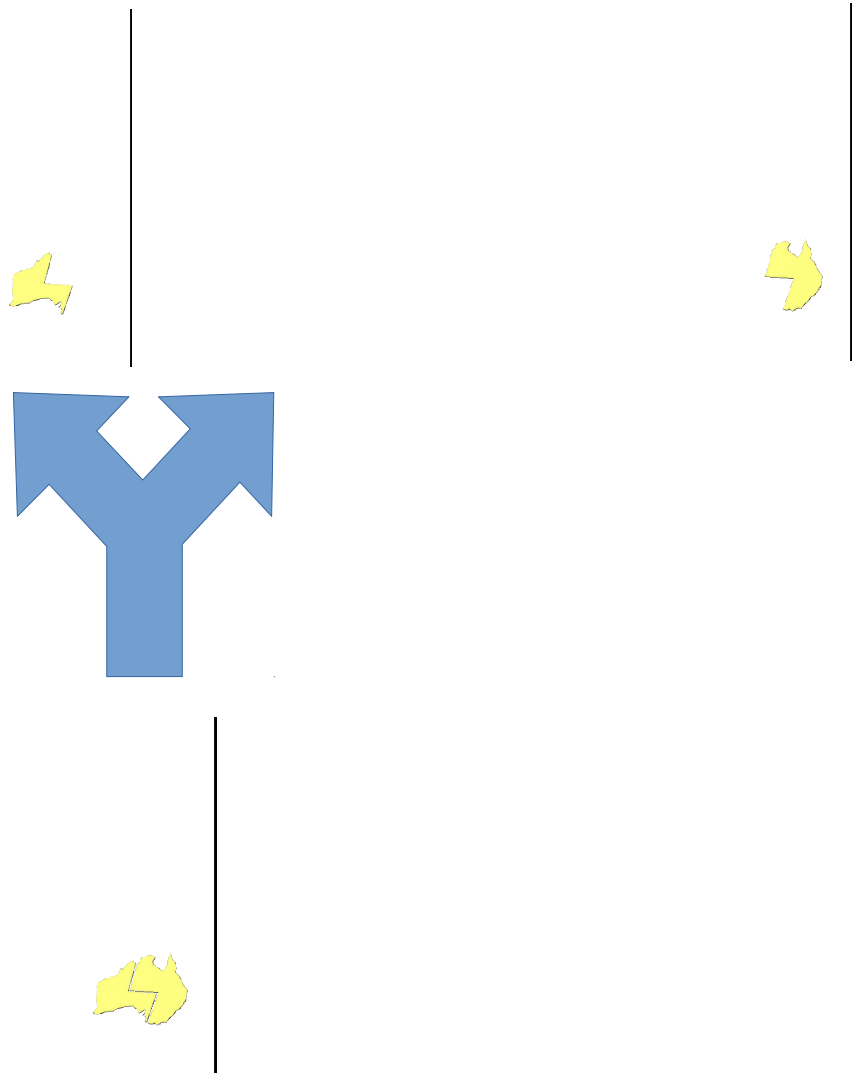
Les méthodes des classes de pays sont utilisées pour gérer les unités situées sur le territoire. Soit ajouter des unités en cas de renforcement. Eliminer les unités en cas de perte en attaque ou en défense. Afin d'initialiser la carte et la liste des voisins, une méthode est utilisée pour ajouter des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

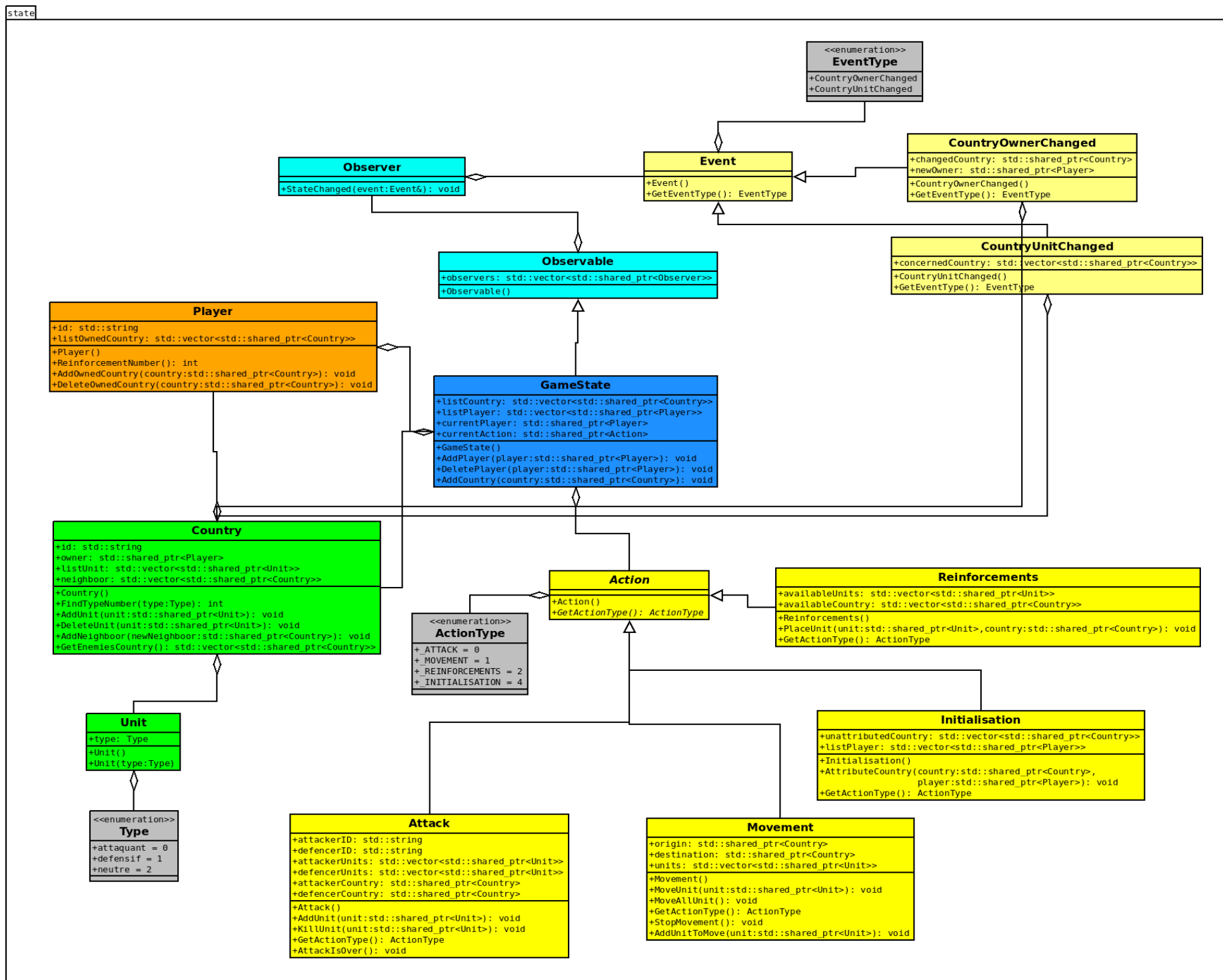


Illustration 1: Diagramme des classes d'état

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 dernière classe GraphicElement est une classe pour traiter chaque élément de Layer indépendamment. Ainsi on peut pour chaque pays modifier de façon indépendante des autres sa couleur en fonction du propriétaire grâce à une simple méthode.

En parcourant la liste des pays issue de Scène la classe UnitRepresentation est capable de retrouver et représenter le nombre de chaque type d'unités sur chaque territoire.

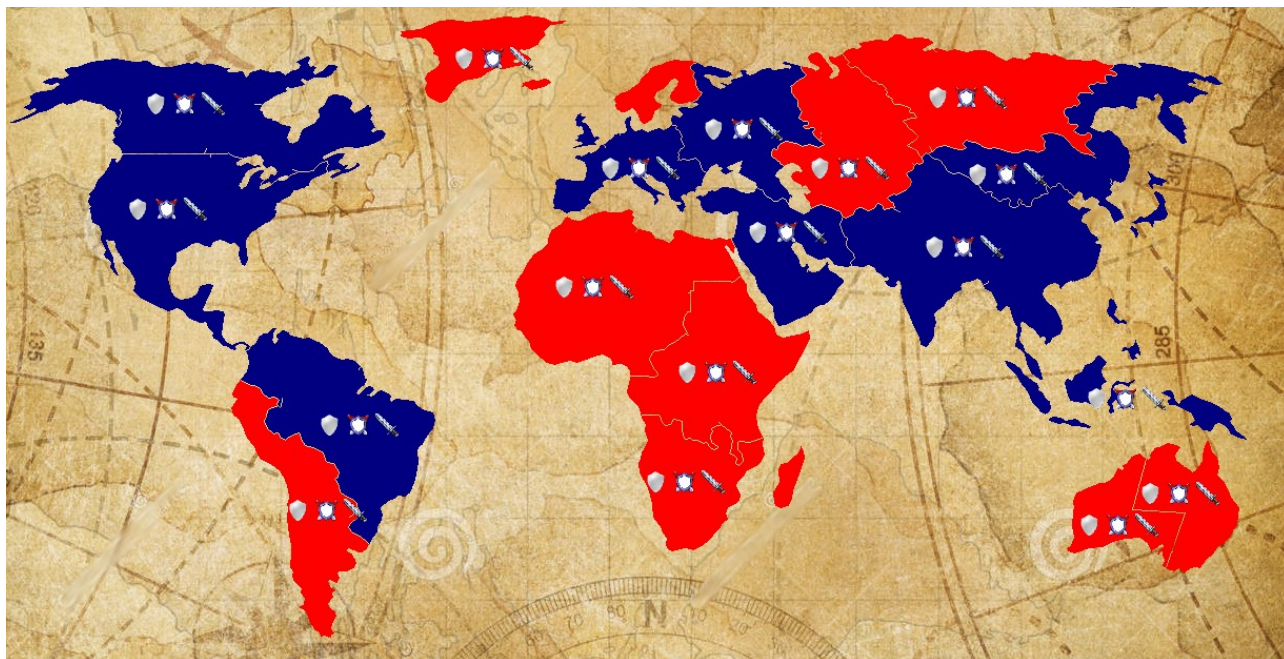
3.3 Conception logiciel : extension pour les animations

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



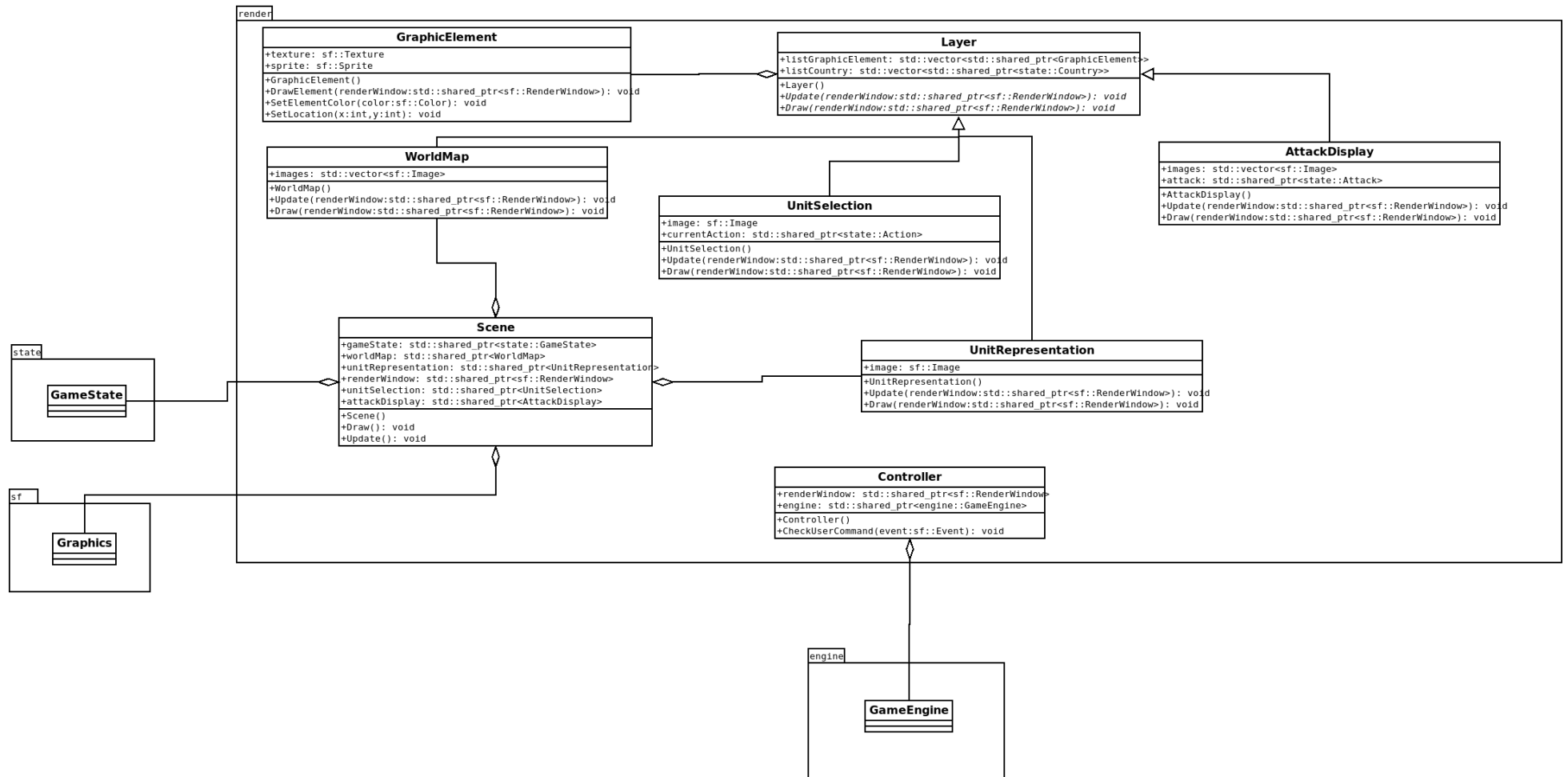


Illustration 2: Diagramme de classes pour le rendu

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 liés 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 phases de jeu qui se suivent de façon séquentielles : pour chaque joueur nous avons les phases suivantes :

- 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 tour 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 a passé chaque phase le moteur change de joueur. Pour cela il a accès à l'état de jeu et peut changer le currentPlayer

4.4 Conception logiciel

Afin d'interagir avec l'utilisateur le moteur doit être capable de prendre des commandes souris ainsi que des commandes venant du clavier (comme appuyer sur la barre espace pour confirmer quelque chose). Nous avons donc implémenté une 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 informations, mais également faire les changements nécessaires :

- 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

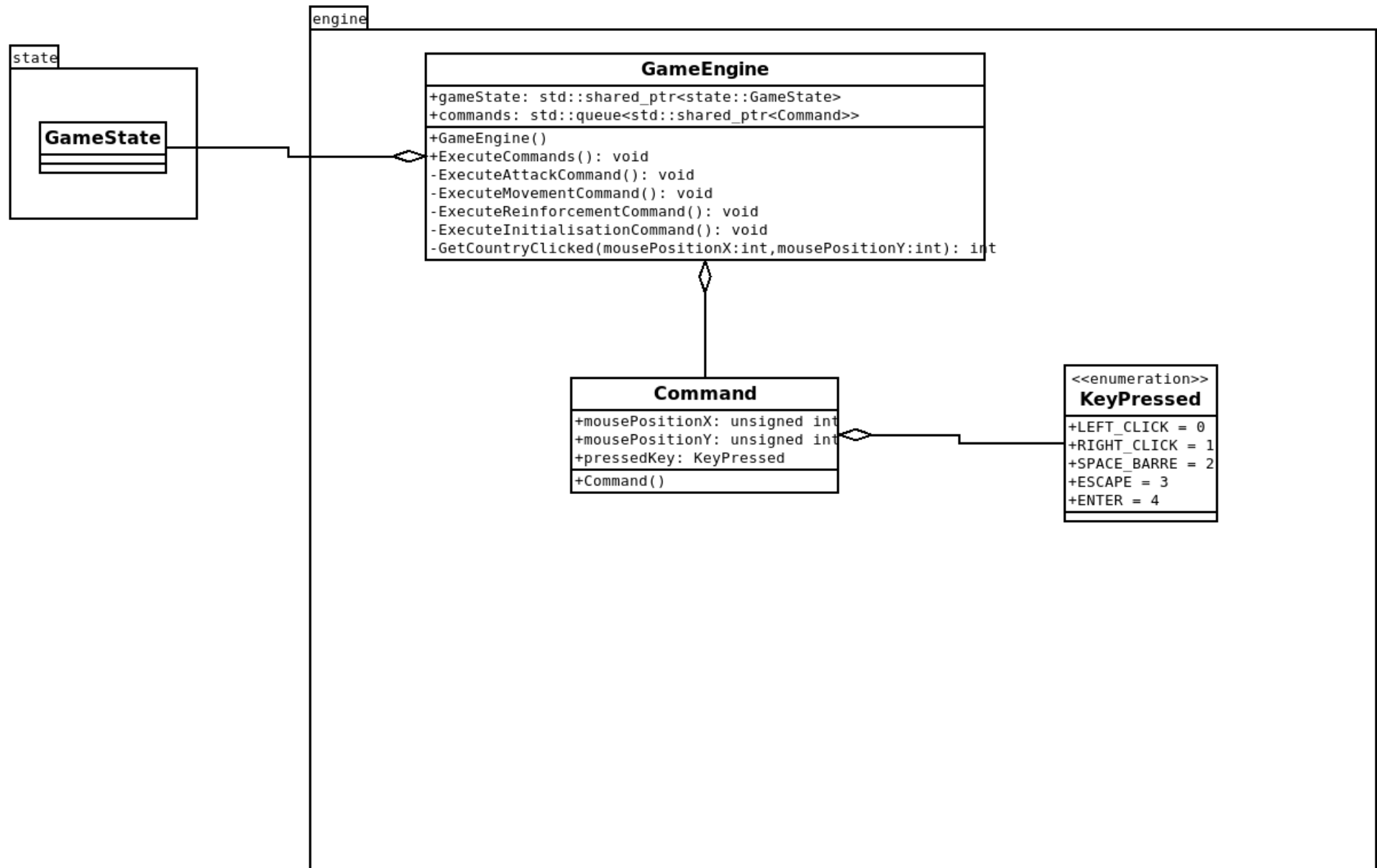


Illustration 3: Diagrammes des classes pour le moteur de jeu

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 une intelligence artificielle peut importe son niveau de jeu, l'IA doit pouvoir envoyer des instructions aux moteur de jeu, comme le ferait un joueur humain. Puis en fonction du niveau de l'IA elle aura besoin de plus d'information afin de pouvoir calculer les choix optimaux.

Afin de gérer les IA et les jours dans la GameState nous ajoutons dans la classe joueur un booléen qui permet détermine si le joueur est ou non un IA. Si il s'agit d'une IA le moteur va faire jouer l'IA en appelant le code de l'IA demandé. Sinon le moteur va attendre les instruction du joueur.

5.1.1 Intelligence minimale

Pour l'intelligence minimale nous avons fait une IA aléatoire. Cette IA va donc juste avoir besoin d'avoir accès à l'état de jeu pour sélectionner l'action en cours : Reinforcement, Attack, Movement. Ainsi pour chaque tour l'IA sait quelles sont ses options a choisir : par exemple sélectionner un pays qui lui appartient, donc parmi la liste des pays qui lui appartient elle peut choisir au hasard une pays. Puis l'IA envoie une requête de commande au moteur de jeu afin de jouer son tour. La commande se fait sous la forme d'un click avec des coordonnées choisi dans un tableau de « define.h » en fonction du pays choisi.

5.1.2 Intelligence basée sur des heuristiques

5.1.3 Intelligence basée sur les arbres de recherche

5.2 Conception logiciel

5.3 Conception logiciel : extension pour l'IA composée

5.4 Conception logiciel : extension pour IA avancée

5.5 Conception logiciel : extension pour la parallélisation

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 une 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

6.1.1 Répartition sur différents threads

6.1.2 Répartition sur différentes machines

6.2 Conception logiciel

6.3 Conception logiciel : extension réseau

6.4 Conception logiciel : client Android

Illustration 4: Diagramme de classes pour la modularisation

