

Projet Logiciel Transversal

Dragons & Licornes

Kevin TOOMEY – Chloé LODOLO



Table des matières

| | | |
|-------|---|----|
| 1 | Présentation générale | 4 |
| 1.1 | Archétype | 4 |
| 1.2 | Principe du jeu | 4 |
| 1.3 | Ressources | 4 |
| 2 | Description et conception des états | 5 |
| 2.1 | Description des états | 5 |
| 2.1.1 | Eléments fixes | 5 |
| 2.1.2 | Eléments variables | 5 |
| 2.1.3 | Etat général | 5 |
| 2.2 | Conception logiciel | 6 |
| 2.2.1 | Les éléments | 6 |
| 2.2.2 | Les conteneurs d'éléments | 6 |
| 3 | Rendu : Stratégie et Conception | 8 |
| 3.1 | Stratégie de rendu d'un état | 8 |
| 3.2 | Conception logiciel | 8 |
| 3.3 | Exemple de rendu | 9 |
| 4 | Règles de changement d'états et moteur de jeu | 11 |
| 4.1 | Horloge globale | 11 |
| 4.2 | Changements extérieurs | 11 |
| 4.3 | Changements autonomes | 11 |
| 4.4 | Conception logiciel | 12 |
| 5 | Intelligence Artificielle | 14 |
| 5.1 | Stratégies | 14 |
| 5.1.1 | Intelligence minimale - aléatoire | 14 |
| 5.1.2 | Intelligence basée sur des heuristiques | 14 |
| 5.1.3 | Intelligence basée sur les arbres de recherche | 14 |
| 5.2 | Conception logiciel | 14 |
| 5.3 | Conception logiciel : extension pour l'IA composée | 14 |
| 5.4 | Conception logiciel : extension pour IA avancée | 14 |
| 5.5 | Conception logiciel : extension pour la parallélisation | 14 |
| | Illustration 4 : Diagramme des classes pour l'intelligence artificielle | 15 |
| 6 | Modularisation | 16 |
| 6.1 | Organisation des modules | 16 |
| 6.1.1 | Répartition sur différents threads | 16 |
| 6.1.2 | Répartition sur différentes machines | 16 |

| | | |
|---------------------|--|----|
| 6.2 | Conception logiciel | 16 |
| 6.3 | Conception logiciel : extension réseau | 16 |
| 6.4 | Conception logiciel : client Android | 16 |

1 Présentation générale

Archétype

L'objectif de ce projet est de réaliser un jeu que l'on peut trouver sur internet sous le nom de World Wars, basé sur l'archétype du Risk, mais en s'éloignant de l'approche militaire du jeu et en rendant le thème plus léger. Ainsi, au lieu d'une bataille entre missiles et tanks, notre jeu mettra en scène le combat entre une armée de licornes et une armée de dragons.

Principe du jeu

Chaque joueur est le dirigeant d'une armée : l'un possède une armée de licornes, l'autre une armée de dragons. Le but du jeu est de conquérir tous les territoires ennemis.

Ressources

Pour réaliser ce projet, nous utiliserons les trois textures suivantes :



Figure 1 - Textures des « soldats »



Figure 2 - Textures des bulles indiquant le nombre de soldats sur un territoire



Figure 3 -

Textures des différents territoires : occupé par les licornes, inaccessible, occupé par les dragons

Description et conception des états

Description des états

Un état du jeu est formé d'éléments que l'on a divisé en deux catégories : les éléments fixes et les éléments variables. Les territoires (ou cellules) du plateau sont des éléments fixes tandis que l'armée occupant un territoire et le nombre de soldats présents sur ce territoire sont des éléments variables. Tous les éléments possèdent des coordonnées (x,y) permettant de les repérer sur le plateau et un identifiant qui indique sa nature (fixe ou variable).

Eléments fixes

Le plateau du jeu est formé par un ensemble de cellules de forme hexagonale représentant les territoires pouvant appartenir à l'un ou l'autre des joueurs.

Certains territoires sont inaccessibles : aucun des deux joueurs ne peut envahir ce type de territoire. Ils ont pour but de permettre la modification du plateau à chaque partie différente et de complexifier légèrement le jeu en constituant des obstacles. Ils sont représentés par une cellule avec des rochers.

Le reste des territoires appartient forcément à l'un ou à l'autre des joueurs. Si le territoire appartient aux licornes, il est beige ; s'il appartient aux dragons, il est rouge.

Eléments variables

Selon l'évolution du jeu, un territoire - à partir du moment où il est accessible - peut appartenir soit à l'armée des licornes, soit à l'armée des dragons. Les données concernant l'armée qui occupe un territoire et le nombre de soldats formant cette armée présents sont variables.

En effet le nombre de soldats évolue forcément lors d'une attaque et peut aller de 1 à 8 soldats, et après un combat un territoire des licornes peut devenir propriété des dragons et inversement.

Etat général

L'état général est décrit par deux tableaux, l'un contenant les éléments fixes (les territoires) et l'autre les éléments variables (le type de l'armée et son nombre). Ces deux tableaux permettent de décrire l'état du jeu à un instant donné.

Conception logiciel

Le diagramme des états est présenté en Illustration 1. On distingue deux types de classes : les éléments et les conteneurs d'éléments.

La sémantique des couleurs utilisées est la suivante :

- le rose symbolise les classes d'éléments. Le rose le plus foncé correspond à la classe mère, abstraite ; le rose plus clair correspond aux deux classes filles
- le vert a quant à lui été utilisé pour les classes de conteneurs d'éléments

Les éléments

Nous avons trois classes : la classe Element, la classe Territory et la classe Team.

La classe Element est la classe mère des deux autres classes. Elle contient deux attributs qui sont les coordonnées x et y de chaque élément dans la grille du plateau de jeu, un constructeur et un destructeur, et une méthode nommée isStatic(). Cette dernière méthode est abstraite, et nous permet de savoir si un élément est variable ou fixe ; nous avons créé cette méthode car il n'y a pas de possibilité d'introspection d'une instance en C++.

La classe Territory a été créée pour représenter les cellules constituant les territoires du jeu. Elle possède un unique attribut, territoryStatus, dont la vocation est d'indiquer si la cellule peut être occupée par une armée et, si c'est le cas, par laquelle elle est occupée : l'attribut peut valoir impossible, dragons ou unicorns ; ainsi qu'un constructeur, un getter pour obtenir la valeur de l'attribut territoryStatus et la redéfinition de la méthode isStatic().

La classe Team sert à représenter l'armée. Elle a donc deux attributs. L'un indique de quelle armée il s'agit : c'est l'attribut teamStatus, dont les deux valeurs possibles dragons et unicorns ; l'autre, nommé nbCreatures, donne le nombre de soldats occupant la case : un entier entre 1 et 8. Elle possède également quelques méthodes : un constructeur, les getters et setters de teamStatus et nbCreatures et la redéfinition de la méthode isStatic().

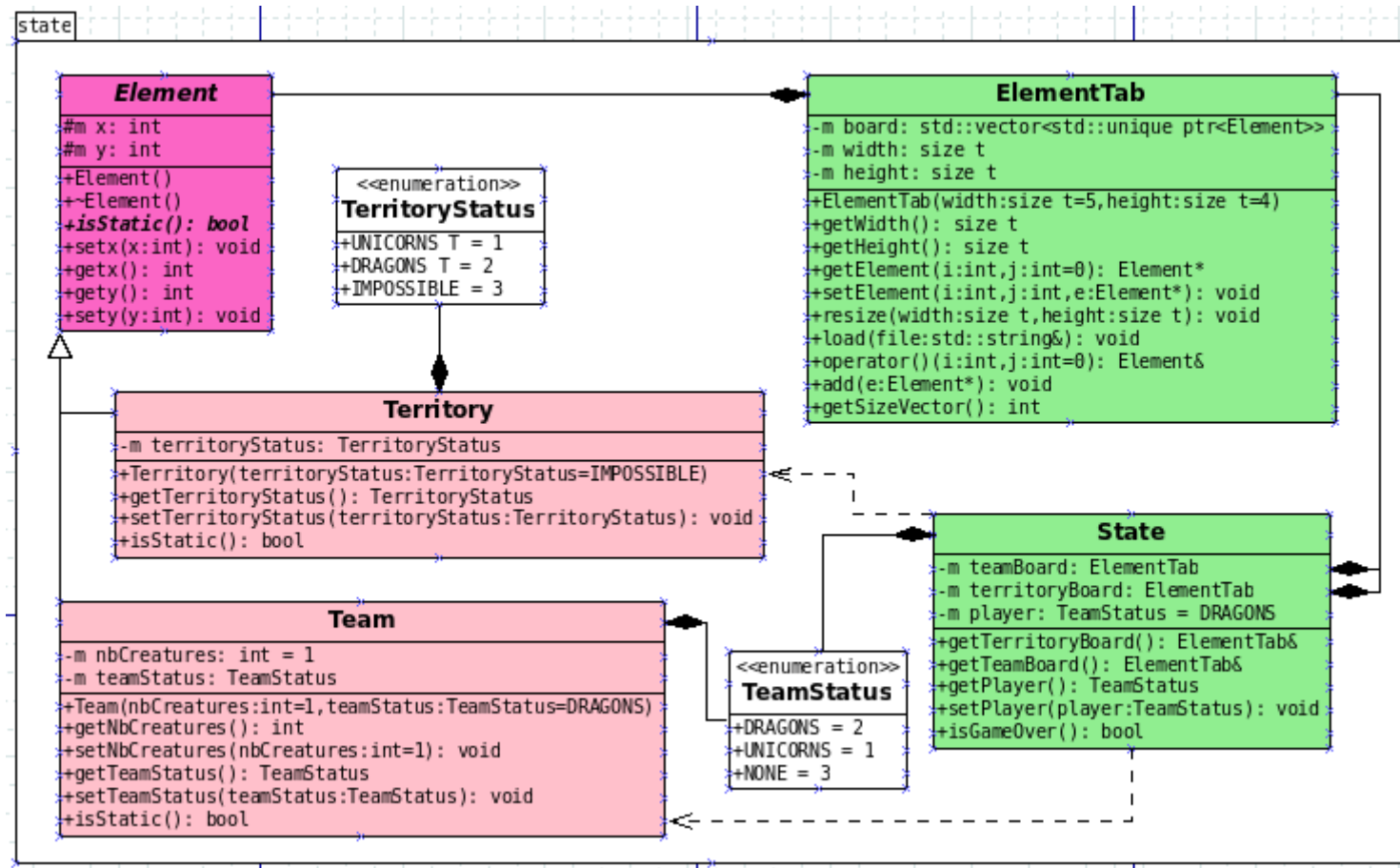
Les conteneurs d'éléments

On distingue la classe ElementTab de la classe State.

La première nous permet de construire des tableaux à deux dimensions contenant des objets de type Element, offrant la possibilité de créer notre plateau de jeu à base des cellules de type Territory.

La classe State permet quant à elle d'accéder à la totalité des données de l'état.

Illustration 1 : diagramme des classes de l'état



Rendu : Stratégie et Conception

Le diagramme du rendu est présenté en Illustration 2. La sémantique des couleurs utilisées est la suivante :

- le bleu symbolise les classes Layers. Le bleu le plus foncé correspond à la classe mère, abstraite ; le bleu plus clair correspond aux deux classes filles
- le vert a quant à lui été utilisé pour la classe Surface
- le violet représente la classe Tile
- le rose symbolise les classes TileSet. Le rose le plus foncé correspond à la classe mère, abstraite ; le rose plus clair correspond aux trois classes filles

Stratégie de rendu d'un état

Pour effectuer le rendu, nous avons choisi de découper la scène à rendre en 3 plans. Le premier plan permet d'afficher tous les territoires : les hexagones rouges, beiges et rocheux. Le second plan est constitué des icônes des dragons et des licornes à placer sur les tuiles correspondantes. Le dernier plan a pour fonction d'afficher le nombre de créatures présentes sur chaque tuile.

Chaque plan contient deux informations qui sont transmises à la carte graphique : une unique texture et une unique matrice contenant la position des éléments dans la fenêtre et les coordonnées de l'image correspondant dans la texture. En conséquence, chaque plan ne pourra rendre que les éléments dont les tuiles sont présentes dans la texture associée.

Conception logiciel

Layers. Le cœur du rendu réside dans le groupe (en bleu) autour de la classe Layer. Le principal objectif des instances de Layer est de donner les informations à transmettre à la carte graphique. Une instance de la classe Surface donne les coordonnées de chacun des éléments dans la grille et dans la texture correspondante, et une instance d'une classe fille de TileSet permet d'exploiter la texture associée à chaque plan.

Les classes filles de la classe Layer se spécialisent pour l'un des plans à afficher. La classe ElementTabLayer permet l'affichage des territoires et des soldats tandis que la classe StateLayer affiche le nombre de soldats.

Les classes ElementTabLayer et StateLayer s'articulent autour de la méthode initSurface(). Celle-ci fabrique une nouvelle surface, charge la texture adaptée, puis initialise la liste des sprites. Pour afficher le niveau, elle demande un

nombre de quads/sprites égal aux nombre de cellules dans la grille avec `initHexas()`. Puis, pour chaque cellule du niveau, elle fixe leur position avec `setSpriteLocation()` et leur tuile avec `setSpriteTexture()`.

Surfaces. Chaque surface contient une texture du plan et une liste de paires de quadruplets de vecteurs 2D. L'élément `texCoords` de chaque quadruplet contient les coordonnées des quatre coins de la tuile à sélectionner dans la texture. L'élément `position` de chaque quadruplet contient les coordonnées des quatre coins du carré où doit être dessiné la tuile à l'écran.

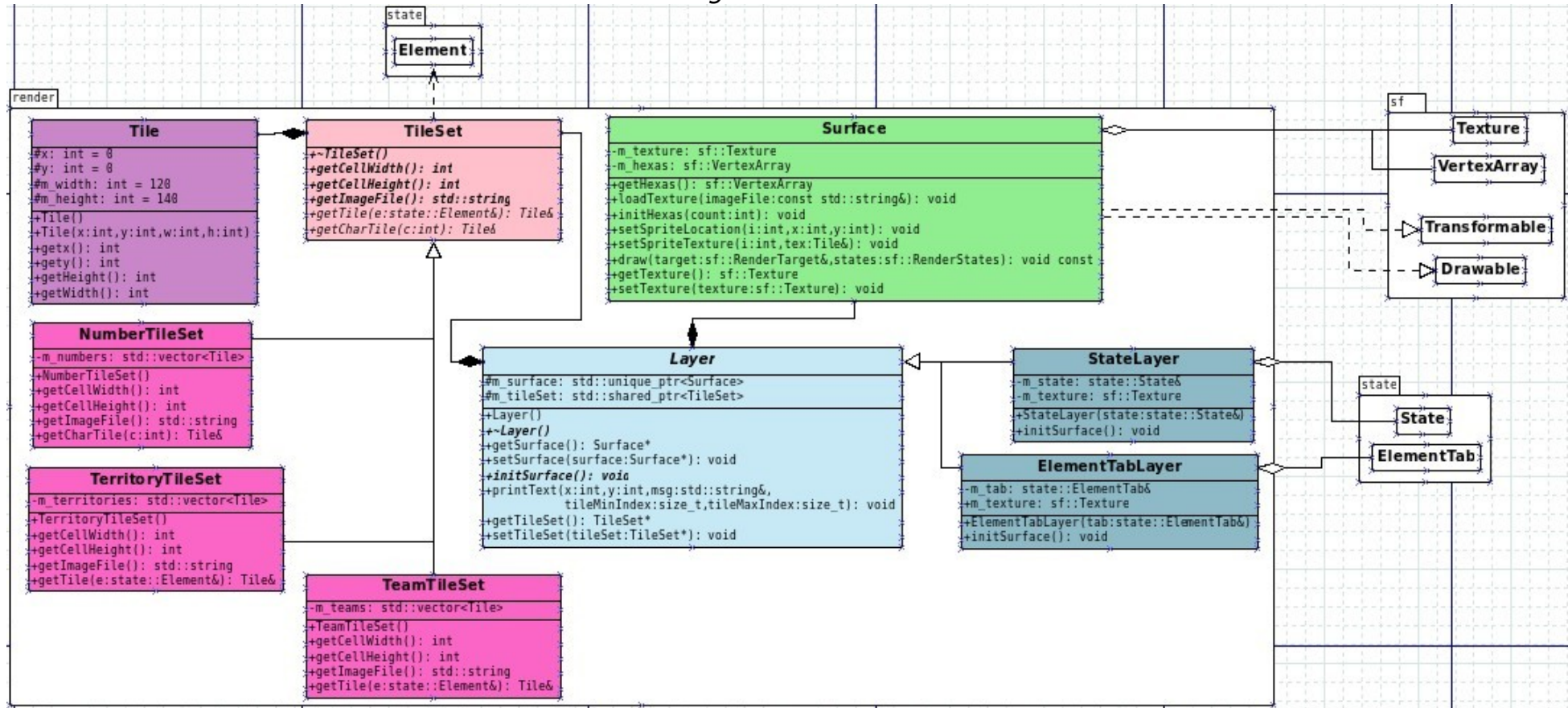
Tuiles. Les classes filles de `TileSet` regroupent toutes les définitions des tuiles d'un même plan. La méthode `getTile()` est celle qui permet d'obtenir la partie de texture que l'on souhaite. En passant en argument de cette méthode une instance d'une classe d'`Element` (`Team` ou `Territory`), celle-ci renvoie une instance de la classe `Tile` contenant la largeur et la hauteur de l'image à sélectionner, ainsi que les coordonnées du point en haut à gauche de cette image : cela permet de sélectionner la partie de la texture contenant l'image de l'élément que l'on souhaite afficher.

Exemple de rendu



Exemple de rendu

Illustration 2 : diagramme des classes du rendu



Règles de changement d'états et moteur de jeu

Horloge globale

On passe d'un état à un autre sans transition, il n'y a pas de notion d'état intermédiaire.

Changements extérieurs

Les changements extérieurs sont réalisés par l'utilisateur à l'aide de la souris. Dans une version suivante, il pourra sélectionner la case avec laquelle il veut attaquer puis celle qu'il veut attaquer.

Les règles du jeu sont les suivantes.

Au début du jeu, les armées de deux joueurs sont égales en quantité, mais les territoires possédés par chaque armée et le nombre de soldats présents sur chaque territoire sont placés aléatoirement.

Le joueur actif choisit le territoire qui attaque ainsi que le territoire ennemi qu'il veut attaquer, tout en sachant qu'il ne peut attaquer qu'un territoire frontalier. Un lancer de dés a alors lieu : chacun des deux joueurs a autant de lancers de dés qu'il a de soldats sur son territoire. Les résultats des lancers de chaque joueur sont ensuite sommés.

Si celui qui fait le score le plus haut est l'attaquant, alors celui-ci devient propriétaire du territoire ennemi et tous les soldats du territoire attaquant sauf un migrent vers le territoire nouvellement conquis, offrant un territoire en plus pour l'attaquant et une nouvelle répartition des soldats. En revanche si celui qui fait le score le plus haut est le défenseur alors le territoire attaquant perd tous ses soldats sauf un. En cas d'égalité, la défense l'emporte.

Lorsque le joueur considère qu'il a fini son tour, il passe la main à l'autre joueur et des renforts sont alors envoyés sur certaines cases du joueur dont le tour vient de se finir.

Le jeu est fini lorsque tous les états de la carte sont possédés par la même équipe.

Changements autonomes

Nous avons deux types de changements autonomes : l'initialisation de l'état au tout début du jeu et la gestion des renforts en fin de tour, après les différentes attaques d'une équipe.

Il est à noter que le moteur est fait tel que si l'on tente d'attaquer un territoire que l'on ne peut attaquer (un territoire inaccessible ou un territoire de notre équipe), rien ne se passe et l'état n'est pas modifié.

L'état initial est pour l'instant construit case par case d'une manière bien définie. Dans une version ultérieure, il sera généré aléatoirement. La gestion des renforts se fait pour le moment de façon simple : un soldat est ajouté sur chaque territoire du joueur dont le tour vient de se finir. Cette façon de faire pourra être complexifiée par la suite.

Conception logiciel

Le diagramme des classes pour le moteur du jeu est présenté en Illustration 3. L'ensemble du moteur de jeu repose sur un patron de conception de type Command, et a pour but la mise en œuvre différée de commandes extérieures sur l'état du jeu.

La sémantique des couleurs utilisées est la suivante :

- Le bleu symbolise les classes Command. Le bleu le plus foncé correspond à la classe mère, abstraite ; le bleu plus clair correspond aux deux classes filles
- Le orange symbolise les classes Actions. Le orange le plus foncé correspond à la classe mère, abstraite ; le orange plus clair correspond aux deux classes filles
- Le vert a quant à lui été utilisé pour la classe Engine
- Le blanc représente les énumérations CommandType et ActionType.

Command. Le rôle de ces classes est de représenter une commande, quelle que soit sa source (automatique, clavier, réseau, ...). Ces classes ne gèrent absolument pas l'origine des commandes, ce sont d'autres éléments en dehors du moteur de jeu qui fabriquerons les instances de ces classes. Une énumération a été définie pour différencier les commandes :

- InitBasicState : initialise l'état de départ du jeu
- AttackCommand : permet d'effectuer l'attaque d'une case depuis une autre. Cette commande détermine lequel des deux territoires gagne la bataille
- GestionRenforts : effectue l'attribution de renforts aux territoires de l'équipe qui vient de finir son tour

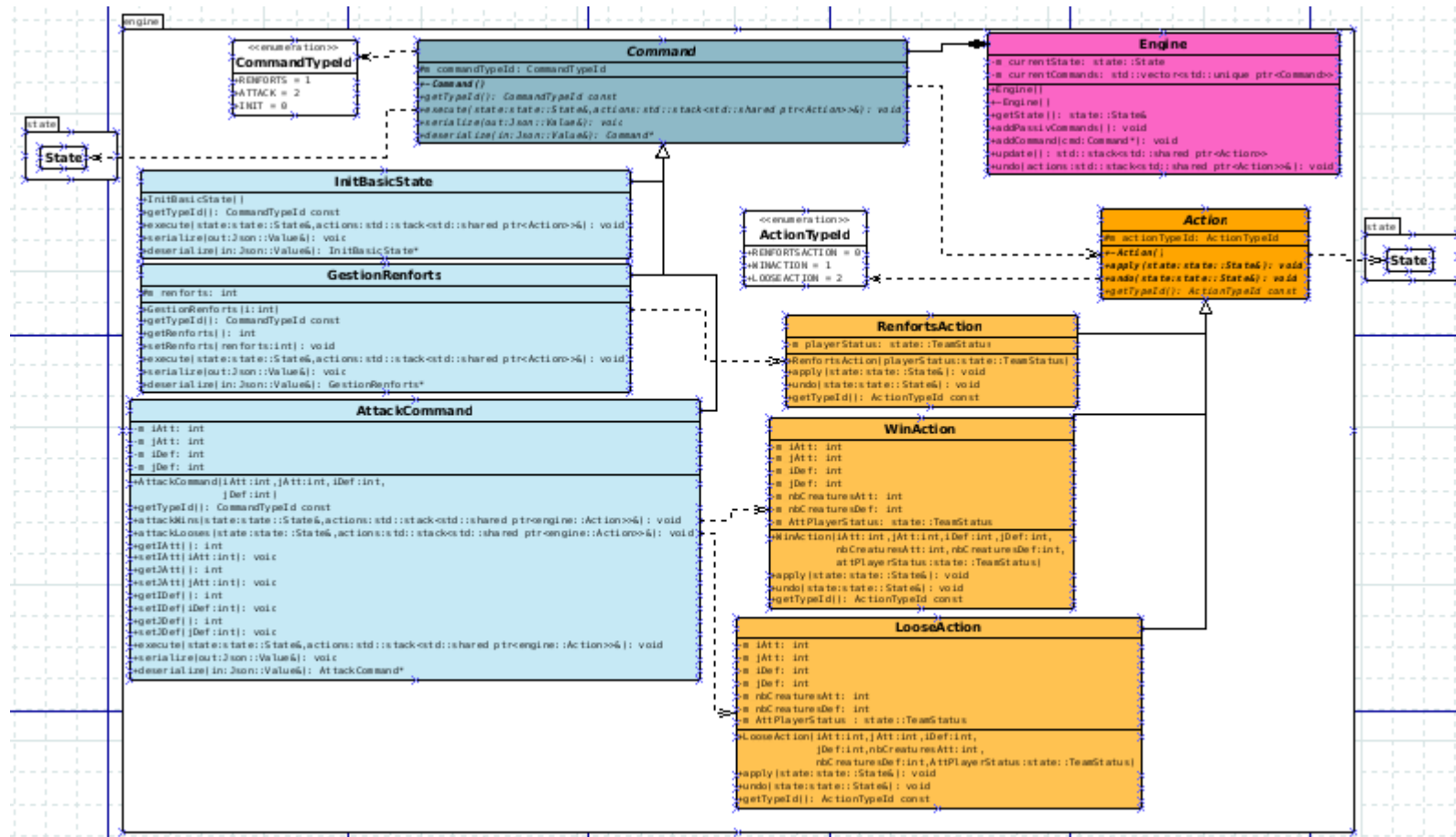
Action. Le rôle de ces classes est de représenter une action qui a été effectué sur l'état. Ces classes permettent de stocker l'information liée à une commande qui modifie l'état. Une Action va nous permettre d'enregistrer une commande qui a été effectué et pour pouvoir revenir en arrière. Le but étant de créer une IA qui parcourt un arbre des possibilités et fait les meilleurs choix. Une énumération a été définie pour différencier les Actions :

- LooseAction : permet d'effectuer/annuler la défaite d'une attaque sur une case depuis une autre.

- WinAction: permet d'effectuer/annuler la victoire d'une attaque sur une case depuis une autre.
- RenfortsActions: permet d'effectuer/annuler l'attribution de renforts aux territoires de l'équipe qui vient de finir son tour.

Engine. C'est le cœur du moteur. La liste des commandes à exécuter est contenue dans un vector. Puis, la méthode update() exécute les commandes contenues dans la liste et la vide ainsi depuis la commande la plus ancienne jusqu'à la plus récente. Cette méthode fait appel aux méthodes execute() de chaque commande afin de modifier l'état, celle-ci font appelle aux méthodes apply() des Actions liées aux commandes. La classe engine contient aussi la méthode undo() qui permet de revenir au tour précédent. Celle-ci fait appelle aux méthodes undo() de chaque action.

Illustration 3 : diagramme des classes du moteur



Intelligence Artificielle

Stratégies

Intelligence minimale - aléatoire

L'IA attaque dans un ordre aléatoire tant qu'elle peut, c'est-à-dire jusqu'à ce qu'elle ne puisse réaliser aucune attaque, soit parce qu'il ne reste qu'une unité sur chacune de ses territoires soit parce que les territoires alentours ne sont pas attaquables (ils contiennent trop de soldats pour être attaqués).

Intelligence basée sur des heuristiques

Afin d'offrir un meilleur comportement que le hasard, nous proposons un ensemble d'heuristiques. Le but de l'IA est ici de conquérir l'ensemble des territoires accessibles de la carte. Ainsi, l'IA doit cette fois réaliser à chaque tour l'attaque la plus sûre, c'est-à-dire celle où l'écart entre le nombre de soldats sur la case attaquante et la case attaquée est le plus grand. Si cet écart est nul, elle tente quand même sa chance.

Intelligence basée sur les arbres de recherche

Nous proposons maintenant de mettre en place une intelligence basée sur les arbres de recherche afin de disposer d'une IA particulièrement efficace.

Le jeu peut alors être représenté par un graphe : les arcs entre les sommets du graphe d'état sont les changements d'états ; chaque sommet représente un état du jeu. Le score d'un sommet correspond au nombre de territoires conquis (avec tout de même le hasard lié aux lancers de dés).

Comme dans le cas de l'IA heuristique, les attaques se feront dans l'ordre le plus judicieux. Si on nomme δ , l'écart entre le nombre de soldats sur la case attaquante et la case attaquée. L'attaque la plus judicieuse est celle qui a le δ le plus grand. Une fois celle-ci réalisée, on regarde les nouvelles cases adverses attaquables et si le δ vers l'une de ces nouvelles cases est négatif alors on remonte l'arbre de recherche, on annule les commandes associées, et notre état retrouve sa forme passée car notre attaque initiale à donner accès à une case ennemie que l'on ne peut pas gérer.

On répète ce procédé à chaque possibilité d'attaque et cela va nous fournir un tour qui conquiert sans se mettre trop en danger.

Conception logiciel

Le diagramme des classes pour l'intelligence artificielle est présenté en Illustration 4. Un seul ensemble de classes est nécessaire. Le bleu le plus foncé correspond à la classe mère, abstraite, tandis que le bleu plus clair correspond aux classes filles.

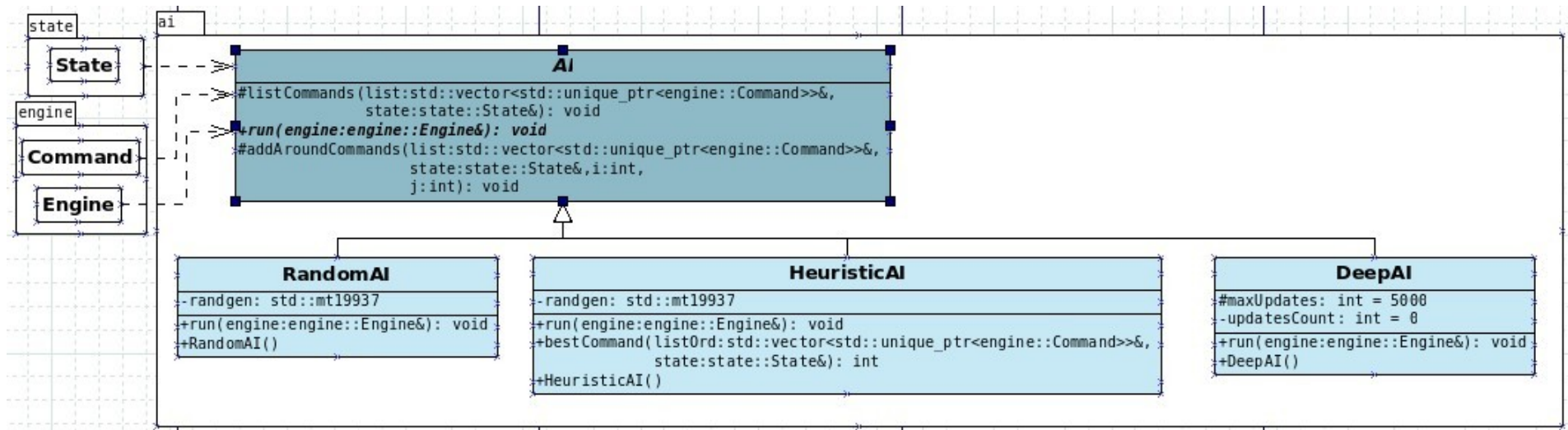
Conception logiciel : extension pour l'IA composée

RandomAI. Cette IA choisit aléatoirement une commande à exécuter.

Conception logiciel : extension pour IA avancée

Conception logiciel : extension pour la parallélisation

Illustration 4 : Diagramme des classes pour l'intelligence artificielle



Modularisation

Organisation des modules

Répartition sur différents threads

Notre objectif ici est de placer le moteur de jeu sur un thread, puis le moteur de rendu sur un autre thread. Le moteur de rendu est nécessairement sur le thread principale (contraire matérielle), et le moteur du jeu est sur un thread secondaire. Nous avons deux type d'information qui transite d'un module à l'autre : les commandes et les notifications de rendu.

Commandes. Les commandes peuvent arriver à l'importe quel moment, y compris lorsque l'état du jeu est mis à jour. Pour résoudre ce problème, nous proposons d'utiliser un double tampon de commandes. L'un contiendra les commandes actuellement traitées par une mise à jour de l'état du jeu, et l'autre accueillera les nouvelles commandes. A chaque nouvelle mise à jour de l'état du jeu, on copie les commandes d'un tampon à l'autre. Le temps de la copie étant négligeable (quelques nanosecondes), le tampon de réception est toujours capable d'être enrichi, et cela sans aucun blocage. Il en résulte une parfaite répartition des traitements, ainsi qu'une latence au plus égale au temps entre deux époques de jeu.

Notifications de rendu. Ce cas est problématique, car il n'est pas raisonnable d'adopter une approche par double tampon. En effet, cela implique un doublement de l'état du jeu (un en cours de rendu, l'autre en cours de mise à jour), ce qui augmente de manière significative l'utilisation mémoire et processeurs, ainsi que la latence du jeu. Nous nous sommes donc tournés donc vers une solution avec recouvrement entre les deux modules, en proposant une approche qui le minimise autant que possible. Pour ce faire, nous ajoutons un cache qui va « absorber » toutes les notifications de changement qu'émet une mise à jour de l'état du jeu. Puis, lorsqu'une mise à jour est terminée, le moteur de jeu envoie un signal au moteur de rendu. Celui-ci, lorsqu'il s'apprête à envoyer ses données à la carte graphique, regarde si ce signal a été émis. Si c'est le cas, il vide le tampon de notification pour modifier ses données graphiques avant d'effectuer ses tâches habituelles. Lors de cette étape, une mise à jour de l'état du jeu ne peut avoir lieu, puisque le moteur de rendu a besoin des données de de l'état pour mettre à jour les siennes. Nous avons donc ici un recouvrement entre les deux processus. Cependant, la quantité de mémoire et de processeur utilisée est très faible devant celles utilisées par la mise à jour de l'état du jeu et par le rendu.

Répartition sur différentes machines

Conception logiciel

Conception logiciel : extension réseau

Conception logiciel : client Android