Projet Logiciel Transversal

Chihab BOUREZGUI– Nicolas MORANDEAU

Objectif

Présentation générale

L'objectif de ce projet est de réaliser un jeu en se servant de l'archétype Risk. Notre archétype est un jeu géopolitique dont le but est de « dominer » le monde, par le biais d'armée.

A partir d'une répartition de ses territoires et de ses armées, chaque joueur a pour mission de répondre à l'objectif qui lui est attribué. Pour cela, il va chercher à conquérir ses pays frontaliers par le biais de ses armées où le résultat du combat est décidé grâce à un jeu de dés. En fonction de ses territoires, le joueur recevra des pièces qui lui permettront de se procurer des aides temporaires. La partie se termine lorsqu'un joueur a rempli son objectif ou que les autres ont abandonné.

Règles du jeu

Notre jeu est composé :

- **d'un univers:** Il s'agit du monde en 2D vu du haut. Le monde est divisé en différents territoires, regroupés en continent.
- **de plusieurs joueurs:** Il faut 3 joueurs au minimum pour lancer une partie. Ils représentent les chefs des différentes armées.
- **de plusieurs armées:** Ce sont des pions qui représentent l'effectif de chaque joueur sur les différents territoires.
- **de dés :** Ils permettent de décider de l'issue des combats.
- **de cartes:** Il existe plusieurs type de carte. Des cartes objectifs afin de fixer la mission du joueur, des cartes territoires pour répartir les territoires entre les joueurs, et des cartes aides.
- **des pièces:** Elles permettent de nous renforcer en armées et en attributs.

Ce jeu est un jeu tour par tour. A chaque tour, un joueur peut effectuer les activités suivantes:

- **se renforcer:** Le joueur, en fonction du nombre de ses territoires, reçoit des armées supplémentaires à répartir sur la carte. En fonction de ses victoires, le joueur dispose de pièces lui permettant d'acheter des attributs temporaires de défense ou d'attaque.

- **déplacer:** Il peut déplacer une partie de ses armées vers différents territoires (les siens ou ceux des adversaires). Il doit laisser au minimum une armée sur chacun de ses territoires.

- **attaquer:** Lorsqu'un joueur déplace ses troupes vers des territoires adverses, un combat est engagé à la frontière. Le territoire est conquis lorsqu'il n'y a plus d'armées pour défendre le territoire.

Conception Logiciel

Présenter ici les packages de votre solution, ainsi que leurs dépendances.

Description et conception des états

L'objectif de cette section est une description très fine des états dans le projet. Plusieurs niveaux de descriptions sont attendus. Le premier doit être général, afin que le lecteur puisse comprendre les éléments et principes en jeux. Le niveau suivant est celui de la conception logiciel. Pour ce faire, on présente à la fois un diagramme des classes, ainsi qu'un commentaire détaillé de ce diagramme. Indiquer l'utilisation de patron de conception sera très appriécé. Notez bien que les règles de changement d'état ne sont pas attendues dans cette section, même s'il n'est pas interdit d'illustrer de temps à autre des états par leur possibles changements.

Description des états

Dans cette partie, nous allons lister les différents éléments d'un état de jeu.

Élément joueur: Cet élément donne toutes les informations du joueur (nom, objectif, couleur, statut, nombre de pièce en acquisition). Il propose d'effectuer différentes actions tel que le déplacement de troupe, l'attaque d'un ou plusieurs territoires ainsi que l'achat d'aide.
Il possède plusieurs statuts:

- «en jeu» : cas le plus courant, où le joueur possède des pions et n'a pas encore atteint son objectif.
- «perdant» : lorsque le joueur ne possède plus de territoire sur la carte. Il est alors éliminé et ne peut plus jouer.
- «vainqueur» : lorsque le joueur remplie son objectif en premier. Il est alors déclaré vainqueur et la partie s'arrête.

Élément territoire: Cet élément donne toutes les informations relatives à un territoire qui sont son nom, son occupant et le nombre de pions placés dessus.

Élément pion: Cet élément donne les informations sur le pion. On y retrouve son nom, son affiliation à un joueur (couleur), sa force d'attaque ainsi que son type (en prévision d'amélioration plusieurs types d'armées pourront peut-être être mises en place et donc avec une puissance d'action différente).

Élément frontière: Cet élément va définir les liens entre les différents territoires (terrestre, naval ou nul). C'est grâce à ces liens que les territoires seront déterminés voisins ou non et donc potentiellement attaquable ou pas. C'est donc grâce à cet élément que les combat seront orchestré. Il possède également un attribut l'identifiant au continent auquel il correspond.

Graphe carte: Ceci va correspondre au graphe représentant la carte. On aura 2 listes, l'une avec les sommets (territoires) et l'autre avec les arrêtes (liens entre territoire). Ce graphe permettra ensuite de savoir quel territoire est attaquable et par quel moyen.

Élément Etat: Cet élément réunit toutes les données du jeu dans un tour tel que le nombre de joueur encore en jeu, la durée de la partie.

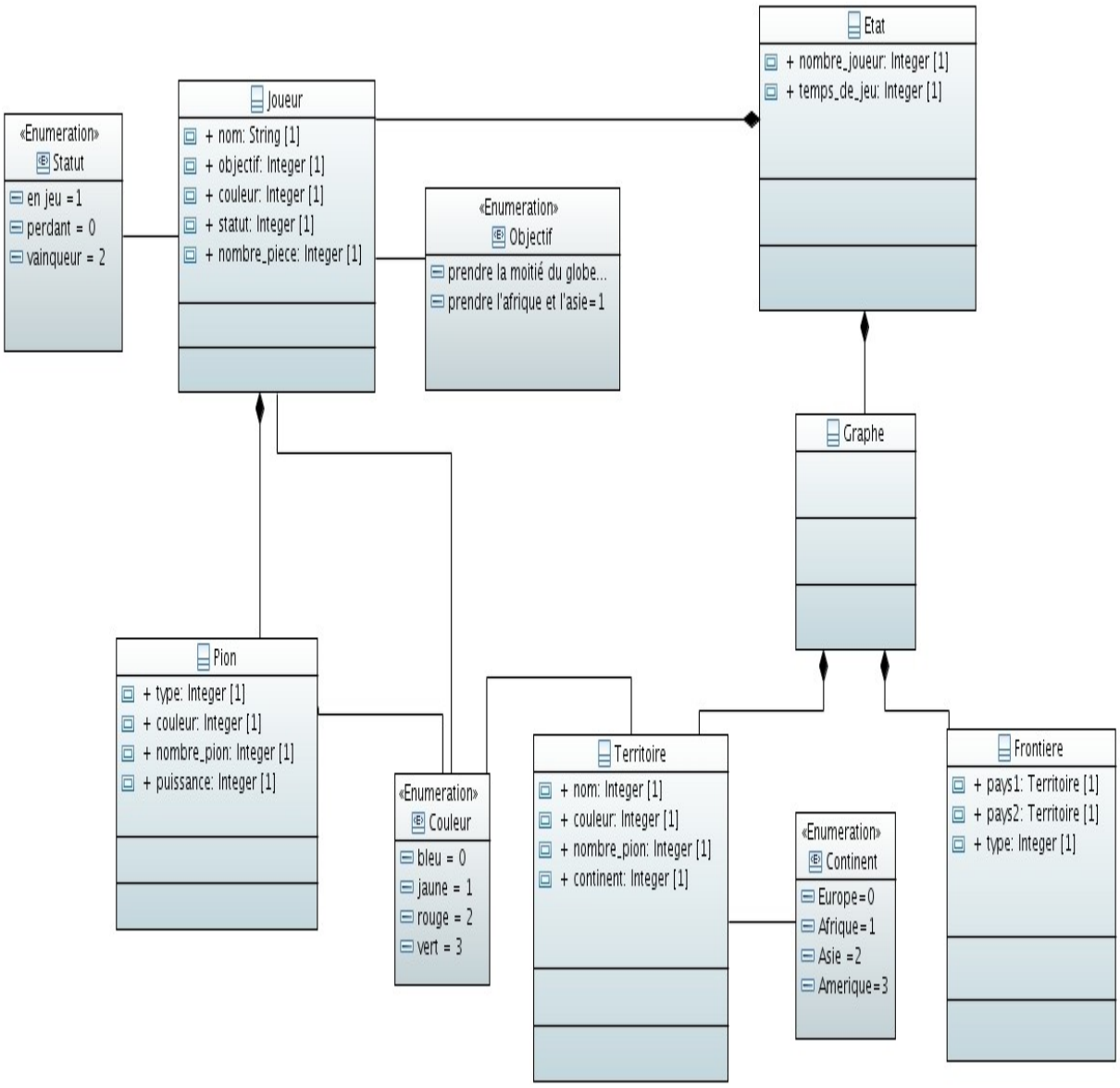


Figure 1: Diagramme de classe de l'état de jeu

Le soucis est que cette conception manque de robustesse. En effet, notre projet a pour vocation de pouvoir être jouer en ligne. Or, en gardant ce diagramme UML comme base de notre projet, nous alons nous heurter à une difficulté assez conséquante lors de la création d'un serveur. Il est temps de repprendre le diagramme afin d'améliorer de manière non négligeable notre conception.

Pour cela, nous allons nous servir d'un pattern, le pattern MVC. Ce pattern a pour vocation de mieux structurer notre code. Il est composé en trois parties : le modèle, la vue, et le contrôleur. Le modèle regroupe toutes les données utiles pour le projet. La vue sert de IHM (interface homme machine), entre le code et l'utilisateur. Elle représente ce que l'utilisateur voit. Quant au contrôleur, il permet de faire le lien entre le la vue et le modèle. Il correspondra, au niveau de notre projet, à notre moteur. Voici un schéma récapitulatif :

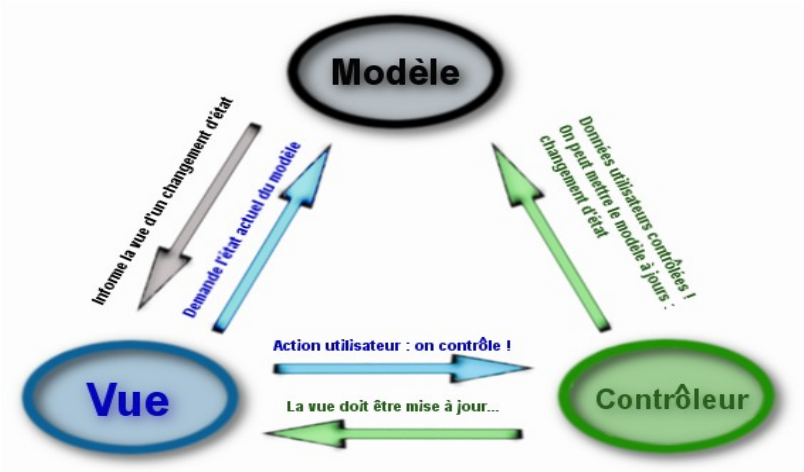


Figure 1: Pattern MVC

Nous allons maintenant lister les différents élements composant notre projet.

Element pays : Cette classe va représenter les différents pays de la carte. Il pourront être attribué aux différent joueurs présent sur la partie.

Element plateau : Cette classe va réunir tous les pays dans une liste, par soucis de simplicité.

Element etape : Cette classe va nous permettre de nous situer dans le jeu, d'un point de vue temporelle. Elle va rescenser l'etape et la sous étape du jeu en cours.

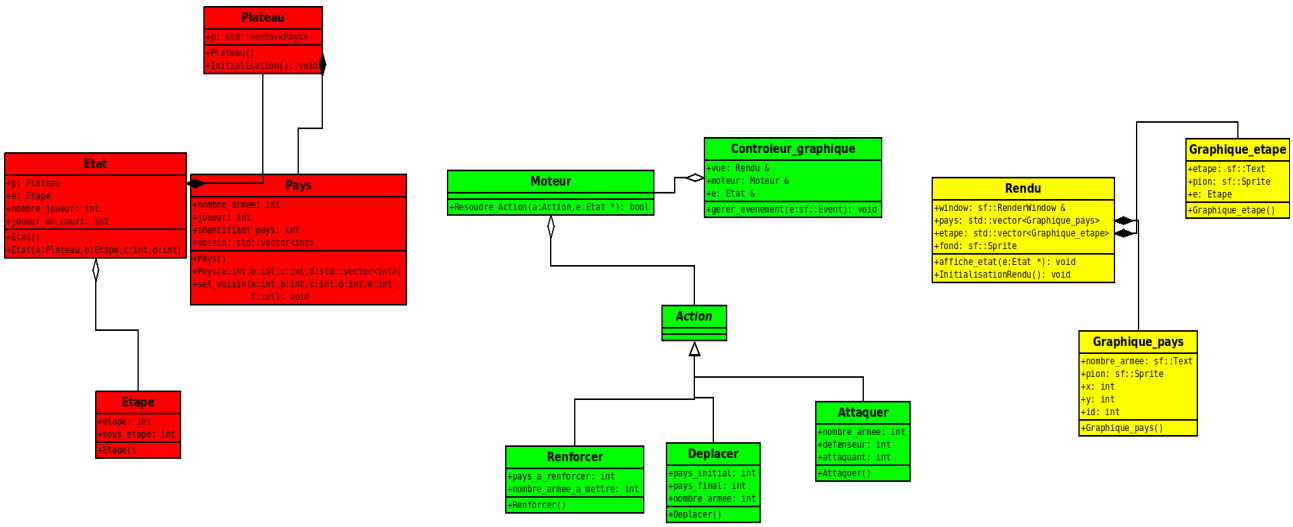
Element etat : Cette classe représente notre jeu, à un instant t. Elle regroupe toutes les données du jeu, du npombre de joueur au plateau.

Element moteur : Cette classe va permettre de faire varier notre etat de jeu. Elle consistera a traiter les données en fonction des actions.

Element Contrôle_graphique : Cette classe va nous permettre de gérer les évenements arrivant. Que ce soit la gestion de la souris ou du clavier, c'est ici que tous sera gérer.

Element Action : C'est une classe abstraite. Cette classe va être heriter par Deplacer, Renforcer et Attaquer pour être utilisé dans le moteur

Element rendu : Cette classe regoupe les classe graphique_etape et graphique_pays. Elle va nous permettre d'afficher une carte de fond, les différent pays avec une appartenance pour chaque étape.



Conception logiciel : extension pour le rendu

Conception logiciel : extension pour le moteur de jeu

Ressources

Rendu : Stratégie et Conception

Présentez ici la stratégie générale que vous comptez suivre pour rendre un état. Cela doit tenir compte des problématiques de synchronisation entre les changements d'états et la vitesse d'affichage à l'écran. Puis, lorsque vous serez rendu à la partie client/serveur, expliquez comment vous aller gérer les problèmes liés à la latence. Après cette description, présentez la conception logicielle. Pour celle-ci, il est fortement recommandé de former une première partie indépendante de toute librairie graphique, puis de présenter d'autres parties qui l'implémente pour une librairie particulière. Enfin, toutes les classes de la première partie doivent avoir pour unique dépendance les classes d'état de la section précédente.

Stratégie de rendu d'un état

Pour l'affichage d'un état de jeu la scène va être découpée en deux plans et un fond d'écran. Le fond d'écran comportera le plateau de jeu (carte du monde) sur lequel on retrouvera sur un premier plan les pions des joueurs (représentant les possessions et l'état de force de chacun). Ce plan contiendra deux informations qui seront transmises à la carte graphique :

- une unique texture contenant les tuiles
- une unique matrice contenant la position des éléments et leurs coordonnées dans la texture.

Pour la mise à jour de l'affichage on va chercher à observer l'état à rendre et réagir lorsqu'une modification se produit. Nous allons pouvoir diviser un «macro» tour en un ensemble de «petits» tours afin de plus facilement observer les changements. De plus notre jeu n'étant pas en temps réel mais dépendant uniquement des modifications provoquées par un tour de jeu il n'y aura pas besoin de synchroniser avec une horloge mais juste en fonction des tours. Une horloge pourra cependant être mise en place quand même pour indiquer par exemple le temps de jeu (de la partie et restant à un joueur pour effectuer son tour).

Conception logiciel

Nous n'avons pas réalisé de diagramme des classes pour le rendu d'état car tout est réalisé dans une unique classe. En effet, n'ayant qu'un seul plan à fournir nous définissons notre ensemble de tuiles ainsi que leur texture dans la méthode «load» de notre classe état Etat_jeu. Cette fonction est paramétrée par les différentes dimensions nécessaires ainsi que le modèle du niveau. Étant fixe et non modifiable, le fond d'écran est réalisé dans le main. La deuxième méthode «draw» permet ensuite d'afficher notre plan de tuiles sur le fond d'écran.

Notre rendu n'est pour l'instant pas optimisé car il ne s'adapte qu'à un fond d'écran unique mais pourra être ensuite amélioré dans des versions ultérieurs si nous en avons le temps.

Conception logiciel : extension pour les animations

Ressources

Exemple de rendu

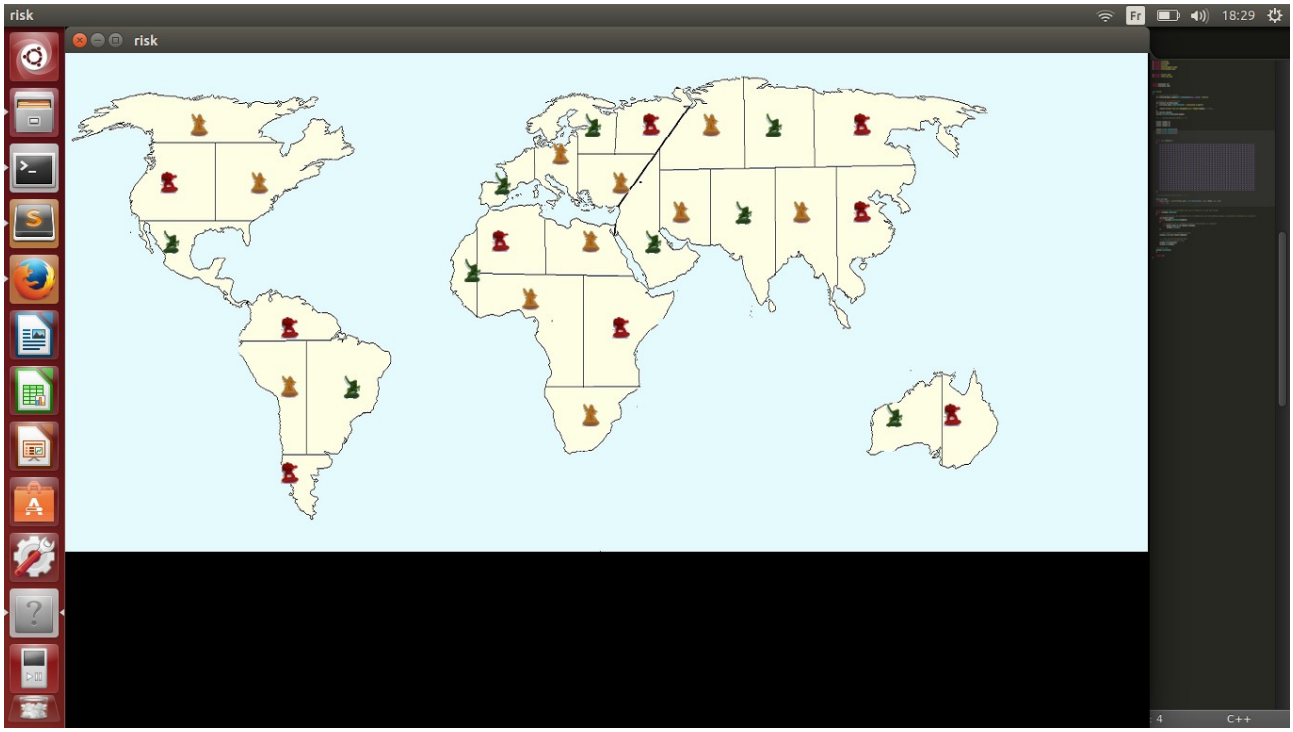


Illustration 2: Diagramme de classes pour le rendu

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.

Description générale

Notre jeu étant sous le format tour par tour aucune horloge n'est nécessaire pour définir un changement d'état. Les modifications se feront à l'issue des différentes étapes d'un tour. On va dans notre cas diviser un tour de jeu en macro-tour et micro-tour, chaque micro-tour contenant une liste d'étapes. Au départ le jeu est initialement créé avec une distribution aléatoire d'un nombre égal de territoires pour chaque joueur contenant chacun le même nombre de pions.

Macro-tour

Un macro-tour correspond à la réalisation de 3 micro-tours effectués (un pour chacun des 3 joueurs de la partie).
Un **changement autonome** est effectué à la fin de chaque macro-tour incrémentant la variable décrivant le nombre de «tour de table» effectué.

Micro-tour

Un micro-tour correspond aux différentes étapes qu'un joueur peut effectuer lors de son tour de jeu.
Lorsque les 3 étapes d'un macro-tour sont effectués un **changement autonome** est effectué modifiant le joueur actif et relançant le cycle de 3 étapes pour celui-ci.

Etape 1 : Renforcement

La première étape d'un micro-tour est le renforcement. Il se déroule de la manière suivante :

1. **Changement autonome** : réception d'un nombre de pions disponibles en fonction du nombre de territoire occupé.
2. **Changement extérieur** : la commande «renforcer A» provoque la demande d'un territoire et la commande «nombre de pions» détermine le nombre X de pions souhaitant être ajouté sur le territoire A.
3. **Changement autonome** : incrémente de X le nombre de pions présent sur le territoire A et décrémente de X le nombre de pion disponibles pour le joueur.
4. **Changement autonome** : fin de l'étape Renforcement et passage à l'étape Déplacement.

Le cycle d'étape 2-3 est réalisé jusqu'à ce que le nombre de pion disponible du joueur soit 0.

Etape 2 : Déplacement

La deuxième étape d'un micro-tour est le déplacement et se déroule de la manière suivante :

1. **Changement extérieur :** la commande «déplacer pions de A vers B» provoque la demande d'un déplacement et la commande «nombre de pions» indique le nombre X de pions souhaitant être déplacés.
2. **Changement autonome :** incrémente de X le nombre de pions présent sur le territoire B et décrémente de X le nombre de pions présent sur le territoire A.

Un **changement extérieur** grâce à la commande «finir Déplacement» provoque alors un **changement autonome** mettant fin à l'étape Déplacement et provoquant le passage à l'étape Attaque.

Etape 3 : Attaque

La dernière étape d'un micro-tour est l'Attaque et se déroule de la manière suivante :

1. **Changement extérieur :** la commande «attaquer A avec B» provoque la demande d'une attaque et la commande «nombre de pions» indique le nombre X de pions du territoire B souhaitant attaquer A.
2. Après simulation du combat un **changement autonome** est réalisé décrémentant le nombre de pions par territoire en fonction des pertes (Xa et Xb perte sur le territoire A et sur le territoire B).
   Si le territoire A ne contient plus de pion alors il devient la propriété du joueur attaquant et le nombre de pions présent sur ce territoire devient (X-Xa) et on retire (X-Xa) au nombre de pions sur le territoire B.
3. **Changement autonome :** une variable est incrémenté déterminant le nombre d'attaque effectuées.

Cette étape s'achève lorsque la variable précédemment incrémentée devient égale à 5 ou lorsque le joueur provoque un **changement extérieur** grâce à la commande «finir Attaque». Cela provoque alors un **changement autonome** déclarant la fin de l'étape Attaque et donc d'un micro-tour.

Conception logiciel

Conception logiciel : extension pour l'IA

Conception logiciel : extension pour la parallélisation

Illustration 3: Diagrammes des classes pour le moteur de jeu

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.

Stratégies

L'ensemble de notre stratégie d'intelligence artificielle repose sur le principe des poupées russes. L'ensemble est décomposé en différents niveaux d'intelligence, de la plus sommaire à la plus avancée. Les niveaux supérieurs font appel aux niveaux inférieurs pour réduire les possibilités à étudier, en éliminant les comportements absurdes ou «dangereux».

Intelligence minimale

Dans un premier temps notre IA va se contenter de faire des actions réalisables pour chaque action qu'elle a a effectuer :

- **Renforcement :** l'IA choisi aléatoirement un territoire parmi ceux lui appartenant et y ajoute les pions supplémentaire dont elle dispose à l'entame de son tour (via l'action AjouterXpions).
- **Déplacement :** l'IA choisi aléatoirement un territoire parmi ceux lui appartenant, elle cherche ensuite aléatoirement si ce territoire possède un voisin appartenant au même joueur. Si non elle recommence cette opération avec un nouveau territoire de départ, et si oui elle fait passer un nombre aléatoire de pions (mais en accord avec les règles du jeu) du territoire de départ à son voisin (via l'action DeplacerXPionsDeAversB).
- **Attaque :** l'IA choisi aléatoirement un territoire parmi ceux lui appartenant, elle cherche ensuite aléatoirement si ce territoire possède un voisin appartenant à un autre joueur. Si non elle recommence cette opération avec un nouveau territoire de départ, et si oui elle simule un un combat avec ce territoire en attaquant avec un nombre aléatoire de pions (mais en accord avec les règles du jeu) et en modifie les effectifs en fonction du résultat (via l'action AttaquerBavecXpionsDeA).

Intelligence moyenne

Après avoir réalisé une intelligence minimaliste, nous souhaitons mettre en place une IA plus développée. Le concept de stratégie commence alors à s'implémenter.

- **Renforcement :** Le renforcement se fait de manière plus ou moins aléatoire. L'IA va déterminer sur quelle territoire du joueur se trouve le moins de troupe, et va lui donner des pions supplémentaires à l'entame de son tour. On peut observer ici un début de stratégie, mais on ne prend pas encore en compte complétement les actions des autres joueurs, c'est une stratégie défensive.

- **Déplacement :** nous n'avons pas encore pris en compte le déplacement dans notre IA moyenne, nous continuons donc pour l'instant à nous servir de notre IA minimaliste.

- **Attaque :** L'IA aura comme argument une «cartographie» du jeu. Cela consiste en une liste de territoires, pour lequel chaque territoire est affilié à un joueur, avec le nombre de pion basé sur celui ci.
  Dans un premier temps, il faut que l'IA détermine quelle pays il doit attaquer. Pour ce faire, nous avons implémenter un algorithme, qui détermine quelle est le continent qu'il doit attaquer en fonction du nombre de territoires conquis. Il attaquera un continent où il y a le plus de territoires en sa possession, afin d'y assoir sa suprématie.

  Ensuite, dans le bon continent correspondant, il faut choisir le pays attaquer. Pour ce faire, l'IA choisira de manière assez aléatoire. Enfin, les règles de combat étant déterminé à l'avance, l'IA n'a pas de maîtrise la dessus.

Le fonctionnement de notre IA peut s'observer sur le jeu par l'appuie sur les touches gauche et droite du clavier :

- *la touche de gauche lance un tour complet de l'IA minimaliste (renforcement, déplacement et attaque) puis passe la main au joueur suivant (on ne tourne pour l'instant que sur les 2 joueurs IA, c'est à dire le jaune et le vert).*
- *la touche de droite lance un tour complet de l'IA moyenne (renforcement, déplacement et attaque) puis passe la main au joueur suivant (on ne tourne pour l'instant que sur les 2 joueurs IA, c'est à dire le jaune et le vert).*

La description complète du tour est observable sur le terminal.

Intelligence basée sur les arbres de recherche

Conception logiciel

Conception logiciel : extension pour l'IA composée

Conception logiciel : extension pour IA avancée

Conception logiciel : extension pour la parallélisation

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.

Organisation des modules

Répartition sur différents threads

Répartition sur différentes machines

Conception logiciel

Conception logiciel : extension réseau

Conception logiciel : client Android

Illustration 4: Diagramme de classes pour la modularisation

Modularisation réseau

Principe

Le but de cette partie est de faire fonctionner notre jeu sur plusieurs machines. Pour cela nous allons mettre en place un système Serveur/Client afin de gérer la communication entre les différents utilisateurs.

On peut représenter cela de la manière suivante :

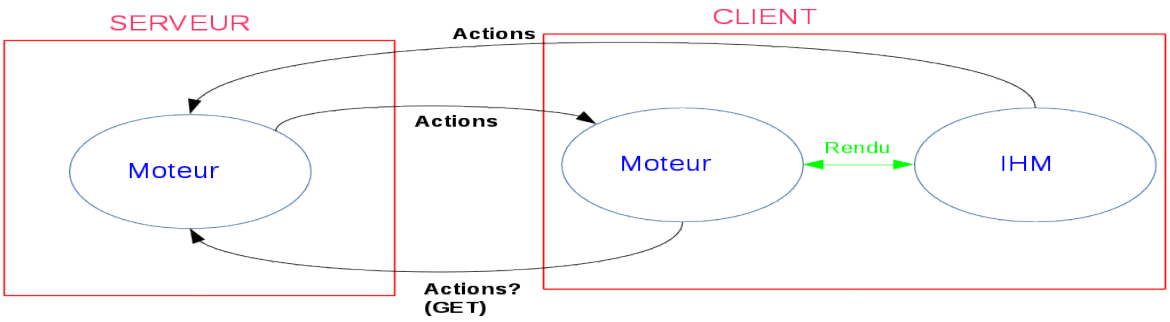


Image 24 : Schéma de principe du fonctionnement

Chaque client possède une IHM et un moteur local, le serveur possède uniquement un moteur.

Le client envoie au serveur les différentes actions produites par l'IHM. Le serveur a ensuite pour rôle de valider ou non la commande et de la mettre en place sur son moteur.

Le moteur local de chaque client interroge le serveur de manière régulière dans le temps afin de savoir si des actions ont été mises en place. Si oui il les applique alors à son tour à son moteur locale. Il n'y a donc plus que le rendu qui relie le moteur local et l'IHM.

Notre jeu étant un tour par tour, et donc relativement lent, l'interrogation au serveur peut ne se produire que toutes les 1 ou 2 secondes.

API Web

Dans cette partie nous allons décrire les types d'informations échangées entre le client et le serveur. Le service fournit est un service de commande où vont transiter les commandes des actions à réaliser.

Les opérations qui vont être mises en place pour cela seront :

- GET : pour récupérer les commandes du côté du client.
- PUT : pour transmettre les commandes au client.

Pour cela nous allons sérialiser les différentes commandes à envoyer (Attaquer, Déplacer, Renforcer, Changer de joueur, Changer d'étape de jeu, Oter pion, Supprimer joueur, Changer possesseur…) sous la forme de Json avec attributs.

Nos Json auront pour premier attribut le type de la commande et les différents attributs suivant varieront ensuite en fonction de celui-ci.Voici quelques exemples de Json pour différentes commandes :

Commande Attaquer	Commande OterPion	Commande Changer de joueur
Json::Value jsonObject; jsonObject[«type»]= «Attaquer» jsonObject[«pays1»]= 1 jsonObject[«pays2»]= 2 jsonObject[«armees»]= 3 jsonObject[«joueur»]= 2;	Json::Value jsonObject; jsonObject[«type»]= «Oter» jsonObject[«pays»]= 1 jsonObject[«paysarmees»]= 2;	Json::Value jsonObject; jsonObject[«type»]= «Changement de joueur» jsonObject[«actuel»]= 1 jsonObject[«suivant»]= 2

Les commandes Attaquer, Déplacer et Renforcer se font de la même façon avec le ou les pays concernés, le nombre d'armées impliqué ainsi que le joueur réalisant l'action.

Les commandes Changer de joueur, Changer d'étape se font de la même façon avec le joueur ou l'étape en cours et le suivant. Pour le changement de joueur il est nécessaire d'indiquer le joueur suivant car certains joueurs peuvent être éliminés (pas forcément nécessaire pour les étapes car c'est toujours le même enchainement).

Les commandes envoyer dans un sens ou dans l'autre peuvent différer. En effet lors de l'envoie par le client d'une commande Attaquer, le moteur du serveur réalisera cette action et renverra elle plusieurs commandes au local : OterPion (1 fois si un seul des pays a perdu, 2 fois si les 2 pays ont des pertes) ainsi qu'un potentiel ChangerPossesseur et Renforcer en cas de conquête d'un pays.

Conception logiciel

Notre conception va être divisé en deux sous parties : la partie serveur et la partie client.
Nous allons commencer par expliquer la partie serveur.

Partie serveur :

Dans un premier temps, nous allons utilisé une version assez simpliste d'un serveur, fortement inspiré de ce qu'on a pu voir en TD. Notre partie serveur sera composé de plusieurs classes :

- ActionDB et Action : ces classes permettent de pouvoir stocker en donnée les différentes commandes envoyées par le client ainsi que de réaliser les actions nécessaires. Il y a

nécessité de stocker les différentes commandes car plusieurs peuvent être réalisées avant la requête du client.

- CommandeService : cette classe permet de traiter les commandes envoyées par le client.
- ServiceManager : cette classe permet de sélectionner le bon service ainsi que la bonne opération à exécuter en fonction de l'URL et de la méthode HTTP.
- ServiceException : cette classe permet de jeter une exception à tout moment pour interrompre l'exécution d'un service.
- AbstractService : cette classe est abstraite gère tous les services REST.

Voici le diagramme correspondant :

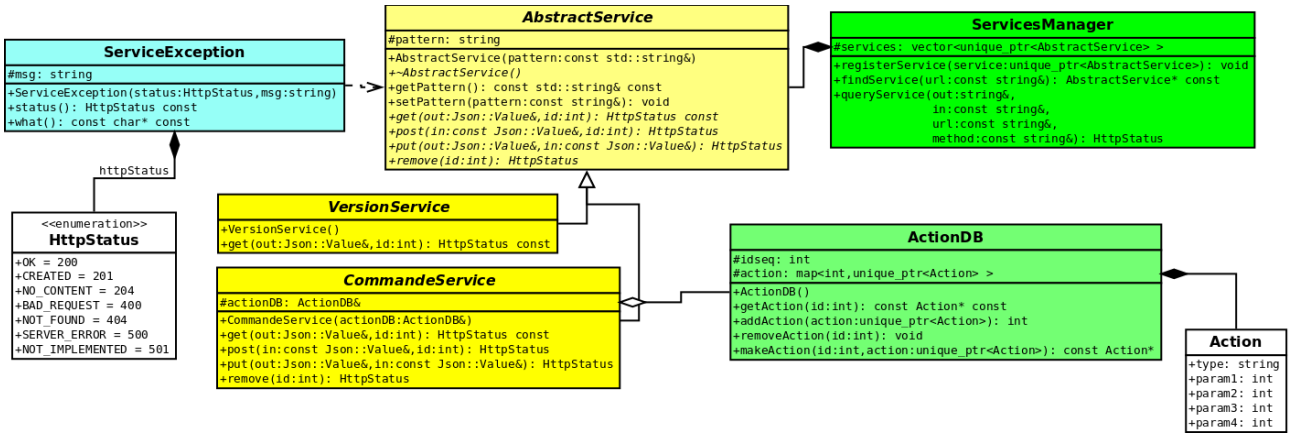


Image 25 : Diagramme des classes côté serveur

Partie client :

Pour la partie client nous utiliserons directement un module fournie par SFML réalisant cette tâche via du Http.