Projet Logiciel Transversal

Adou DIALLO – Raphaël DUJARDIN

Table des matières

1 Objectif

1.1 Présentation générale

Le but du projet est la réalisation d'un jeu de stratégie tour par tour permettant la conquête de territoires inspiré du jeu RISK avec un nombre moindre de règles beaucoup plus simples.

1.2 Règles du jeu

1.2.1 Aperçu

Le but du jeu est de pouvoir s'emparer des territoires adverses tout en défendant le sien et tous ceux précédemment conquis.

- Univers
  L'univers du jeu est une carte 2D en vue de dessus. La carte est décomposée en territoires et en mers. On dénombre 5 types de territoires : prairie, montagne, forêt, désert, et côtier, repérables par une texture et une légende sur laquelle les joueurs se déplaceront. A cela s'ajoute un type particulier de territoire qui est le territoire neutre. Chaque joueur possède un territoire capitale qui a une grande importance. Sur tous ces territoires excepté le neutre sont présentes des troupes.

- Les joueurs
  Les joueurs peuvent être au maximum au nombre de 6. Ils ont chacun une couleur définie qui servira à différencier les territoires possédés par chacun.

- Les personnages
  Les personnages présents sur la carte qui permettront au joueur de jouer seront les troupes qui serviront à défendre ou conquérir un nouveau territoire. Parmi ces troupes existe un héros qui est un soldat particulier avec des caractéristiques différentes des troupes communes.

- Les items
  Les items apparaissent de manière aléatoire au bout d'un certain nombre de tours. Il y en a de plusieurs sortes avec des effets plus ou moins puissants. Ces objets peuvent autant être des malus que des bonus pour le joueur qui devra dès lors bien réfléchir à leur utilisation.

- Les actions
  Il y a deux types d'actions principales dans le jeu : attaquer et déplacer des troupes. On peut aussi construire des ports. Il y a possibilité de passer son tour mais des mesures adéquates apparaissent à la prise de cette décision.

- Les tours
  Chaque joueur joue chacun son tour, pendant un tour il peut y avoir plusieurs phases qui peuvent être influencées par la présence ou non d'item.

1.2.2 Règles détaillées

- Au début de la partie, tous les territoires sont neutres et chaque joueur choisit à tour de rôle un territoire qui sera sa capitale. S'il la perd, il meurt et celui qui a conquis sa capitale conquiert automatiquement tous ses autres territoires.
- Chaque joueur joue chacun son tour. Chaque joueur peut à tout moment de son tour décider de passer la main au joueur suivant sans terminer son tour. Chaque joueur peut à tout moment de son tour décider d'abandonner, dans ce cas il est éliminé de la partie et tous ses territoires redeviennent neutres avec les soldats présents dessus, et peuvent donc être capturés par un autre joueur.
- Pendant un tour, plusieurs phases successives :
  - Renforts : le joueur choisit un de ses territoires sur lequel il reçoit ses renforts (3 soldats par territoire contrôlé)
  - Action (capturer / attaquer / construire un port) : le joueur choisit un de ses territoires comme territoire de départ, et un territoire comme territoire d'arrivée. Si le territoire d'arrivée est le même que le territoire de départ, un port y est construit. Si le territoire d'arrivée est un territoire neutre (contrôlé par aucun autre joueur), il est capturé ainsi que les soldats présents dessus. Si le territoire d'arrivée appartient à un autre joueur, une bataille est livrée. Un port ne peut être construit que si le territoire n'en est pas déjà doté. Un territoire adverse ne peut être attaqué que si la puissance d'attaque totale est supérieure à la puissance de défense totale. La capture de territoire neutre et l'attaque de territoire adverse ne peut se faire que sur un territoire adjacent ou accessible par la mer si le territoire de départ est doté d'un port.
  - Déplacement : le joueur choisit un de ses territoires comme territoire de départ et un autre de ses territoires comme territoire d'arrivée, une partie (au choix du joueur : 1/4, 1/2, 3/4 ou Tout) des soldats présents sur le territoire de départ est alors déplacée sur le territoire d'arrivée. Là encore soit les deux territoires doivent être adjacents soit ils sont séparés par une mer et le premier doit être doté d'un port.

On peut attaquer plusieurs pays dans le même tour mais les dégâts sont d'autant moindres. Pendant la phase d'action, cliquer sur un de ses pays puis sur un pays adverse affiche une flèche montrant que l'attaque s'apprête à être effectuée. Cliquer sur la flèche annule cette décision. L'attaque n'est pas exécutée de suite, il faut attendre que la planification des attaques soit terminée : le joueur peut prévoir d'autres attaques, puis cliquer sur le bouton « Lancer attaques » apparu lors de la planification de la première attaque. Cela n'est possible que pour les attaques sur un territoire adverse, si la première action effectuée est une capture ou une construction de port, elle est exécutée de suite, le bouton « Lancer attaques » n'apparaît pas et on passe immédiatement à la phase suivante de déplacement de troupes. Si on lance n attaques, la puissance d'attaque pour chacune est divisée par n : plus on lance d'attaques dans le même tour, moins elles sont puissantes. La puissance d'une attaque est encore divisée par deux si elle est effectuée via la mer.

Les attaques sont exécutées dans l'ordre où elles ont été planifiées. Si une attaque n'est plus valide pour une raison quelconque, elle est simplement ignorée. Par exemple si le joueur attaque deux fois le même territoire dans le même tour, depuis deux territoires de départ différents, si la première attaque résulte dans la conquête du territoire cible, la deuxième attaque est annulée.

Lorsqu'une attaque est livrée, le nombre de troupes sur le territoire victime est diminué du nombre de dégâts de l'attaque, qui est calculé ainsi : (attaque totale – défense totale) * facteur de chance. L'attaque totale est le nombre de troupes sur le territoire attaquant (divisé par n si plusieurs attaques

dans le même tour, et encore par deux si attaque par la mer, éventuellement altéré par l'utilisation d'un item), la défense totale est la somme des nombres de troupes présentes sur le territoire victime et sur les territoires adjacents (éventuellement via la mer auquel cas l'importance est divisée par deux) appartenant aussi au joueur victime. Le facteur de chance est tiré aléatoirement entre 0,8 et 1,2 ; il influe sur les dégâts mais n'influe pas sur la possibilité de planifier l'attaque (en effet on ne peut attaquer un territoire que si l'attaque totale est strictement supérieure à la défense totale, même si grâce à un facteur chance supérieur à 1 il serait possible sans remplir cette condition de conquérir le territoire, on ne peut pas planifier l'attaque si cette condition n'est pas remplie). Si les dégâts sont supérieurs ou égaux au nombre de troupes présents sur le territoire victime, le nombre de troupes tombe à zéro et le territoire est conquis par l'attaquant.

Si le territoire conquis était la capitale du joueur victime, ce dernier est éliminé, et l'attaquant capture automatiquement tous les autres territoires de ce joueur. Si le territoire conquis est doté de ports, ceux-ci ne sont pas détruits : un territoire doté de ports l'est définitivement.

Chaque territoire a un type (prairie / montagne / forêt / désert / côtier). Lorsqu'un joueur possède des territoires d'au moins 4 types différents, il débloque le héros qui est une unité très puissante. Si le joueur perd des territoires et n'en possède plus de 4 types différents, il conserve son héros malgré tout. Le héros apparaît sur le territoire capitale. Il meurt si le joueur perd le territoire sur lequel son héros est stationné, une fois mort il ne peut plus réapparaître. Le héros compte dans les puissances de défense et d'attaque comme l'équivalent de 20 % de toutes les troupes possédées par le joueur sur toute la carte, sa puissance peut donc varier dans le temps. Si le héros meurt, le joueur voit ses troupes divisées par deux sur tous ses territoires, la mort du héros constitue donc un double malus conséquent.

Lorsqu'on déplace des troupes, en choisissant le ratio (1/4, 1/2, 3/4 ou Tout), on peut également préciser si l'on veut aussi déplacer le héros ou non, via une case à cocher. Le ratio et la case à cocher pour le héros prennent pour valeur par défaut la dernière valeur choisie, si c'est le premier déplacement de la partie la valeur par défaut est 1/2 sans le héros.

Quelques territoires sur la carte sont dotés d'une forteresse. Les items n'apparaissent que sur des territoires neutres ou dotés d'une forteresse, mais ils ont plus de chance d'apparaître sur les territoires forteresse. Ainsi quand il n'y a plus aucun territoire neutre, c'est-à-dire qui n'a pas encore été capturé par un joueur, les items n'apparaissent plus que sur les forteresses ; elles constituent donc un enjeu stratégique.

Au début de chaque tour, un tirage d'item est effectué. La probabilité qu'un item apparaisse dans le tour est 12 %. Le lieu d'apparition est tiré au sort parmi les lieux possibles (territoires neutres ou forteresses sur lesquels il n'y a pas déjà un item), avec un poids de 2 pour les territoires forteresse et un poids de 1 pour les territoires neutres. Si aucun lieu n'est possible, le tirage est annulé et aucun item n'apparaît dans le tour. Si le territoire d'apparition est contrôlé par un joueur, il récolte automatiquement l'item, sinon l'item reste sur le territoire jusqu'à ce qu'un joueur le conquiert et récolte donc l'item. L'item reste sur le territoire jusqu'à 4 tours maximum, ensuite il disparaît s'il n'est pas récolté.

Le type de l'item est également choisi aléatoirement : un premier tirage décide de la catégorie de l'item (85 % de chances que ce soit un bonus simple, 10% que ce soit un malus, 5 % que ce soit le fat bonus), un deuxième tirage équiprobable décide du type d'item exact dans la catégorie choisie.

Certains items sont exécutés immédiatement lorsqu'ils sont récoltés, d'autres sont placés dans

l'inventaire du joueur et il peut les utiliser au début du tour qu'il veut. Si le joueur a déjà un item en inventaire et qu'il en récolte un nouveau, il écrase l'ancien. Plusieurs items peuvent être récoltés et/ou exécutés dans le même tour, puisqu'on peut conquérir plusieurs territoires en un tour, dans ce cas l'item effectivement placé en inventaire est le dernier (les attaques sont exécutées dans l'ordre où elles ont été planifiées).

Liste des items, s'il n'est pas précisé « exécuté tout de suite » c'est un item qui va dans l'inventaire :
- attaquer un territoire distant : permet de faire une attaque sur un territoire qui n'est pas forcément adjacent ou accessible par la mer (sauf les capitales)
- attaquer plusieurs pays à pleine puissance : permet de faire plusieurs attaques dans un même tour sans que leurs puissances soient divisée par le nombre d'attaques
- attaquer et construire un port dans le même tour
- jouer 2 tours d'affilée
- bonus de troupes (10% sur chaque territoire) (exécuté tout de suite)
- malus de troupes (10 % sur chaque territoire) (exécuté tout de suite)
- perte d'un territoire (tiré au hasard, sauf la capitale ou le pays où le joueur a le plus de troupes) (exécuté tout de suite)
- fat bonus : on acquiert le plus gros territoire en nombre de troupes des 2 joueurs qui possèdent le plus de troupes au total, hormis le joueur qui a récolté le bonus s'il fait partie des deux concernés. Si le territoire concerné est la capitale du joueur victime, le deuxième plus gros est choisi à la place. Si le héros du joueur victime est présent sur le territoire concerné, il perd son héros (avec toutes les conséquences que ça implique). (exécuté tout de suite).

A tout moment d'un tour, le joueur peut décider d'utiliser l'item qu'il a en inventaire. L'item est alors valable pour le reste du tour. Attention, si l'item est utilisé après l'action qu'il concerne, il ne sera pas effectivement utilisé et sera perdu : par exemple, il faut utiliser l'item « attaquer un territoire distant » avant de planifier ses attaques. L'item reste en inventaire indéfiniment tant qu'il n'est pas utilisé et qu'aucun autre item n'est récolté. A part pour le fat bonus qui a une apparence spécifique, les joueurs ne peuvent pas savoir le type d'un item avant de le récolter, et ils ne peuvent pas non plus voir l'inventaire des autres joueurs. Ils ne peuvent pas savoir le type d'un item récolté par un autre joueur, à part éventuellement via les effets de cet item.

1.3 Conception Logiciel

Packages :
- state
- render
- engine

Dépendances :
- SFML 2

2 Description et conception des états

2.1 Description des états

L'état du jeu est caractérisé par :
- le joueur dont c'est actuellement le tour
- la phase actuelle au sein du tour (placement des renforts / action / déplacement)
- l'item qui a été activé pendant le tour, soit automatiquement soit sur action du joueur, le cas échéant
- l'état de chaque joueur et de chaque territoire

L'état d'un joueur est caractérisé par :
- s'il est en vie ou pas
- le territoire qui est sa capitale
- le territoire sur lequel est positionné son héros
- l'item qu'il a en inventaire

L'état d'un territoire est caractérisé par :
- la liste des territoires adjacents
- ses propriétés géométriques (position et forme), utiles pour le rendu
- son type (prairie / forêt / montagne / désert / côte)
- s'il est doté d'une forteresse ou non
- le joueur qui le possède le cas échéant
- l'item présent dessus et le nombre de tours restant avant sa disparition (le tour en cours inclus, ce compteur est mis à jour en tout début de tour)
- s'il possède des ports ou non
- le nombre de troupes présentes dessus

2.2 Conception logiciel

L'état du jeu sera représenté par trois classes :
- Game pour l'état du jeu global
- Player pour les joueurs
- Land pour les territoires

Le cycle de vie des objets Player et Land dépend de la classe Game. La classe Game est par ailleurs un singleton car elle représente l'état global de la partie.

Les items ne seront pas représentés par une classe car chaque item a un comportement très différent qui peut intervenir à beaucoup de niveaux différents (dans beaucoup de méthodes différentes), le plus simple, y compris pour ajouter de nouveaux items, est de tester l'item et de faire son action directement dans les méthodes concernées.

A ces trois classes s'ajoutent quelques énumérations (type de liaison entre deux territoires, phase de tour, type d'item et type de territoire), et une structure Cell représentant une case élémentaire de la

carte, cases dont sont composées les différents territoires.

Voir diagramme des classes

2.3 Conception logiciel : extension pour le rendu

La classe Land possède un attribut *geometry* de type liste de structures *Cell*. Chaque objet *Cell* possède une position x, y dans la grille des cellules de la carte (64x64 cellules) et un pointeur sur le territoire auquel il appartient. Les cellules auront une taille à l'écran de 9x9 pixels et les bordures feront 3 pixels d'épaisseur.

2.4 Conception logiciel : extension pour le moteur de jeu

La classe Game sera dotée de quelques getters utilitaires, qui permettent de faire certains calculs fréquents pouvant être requis à plusieurs endroits différents (validation des actions utilisateur, rendu, IA…). Notamment :

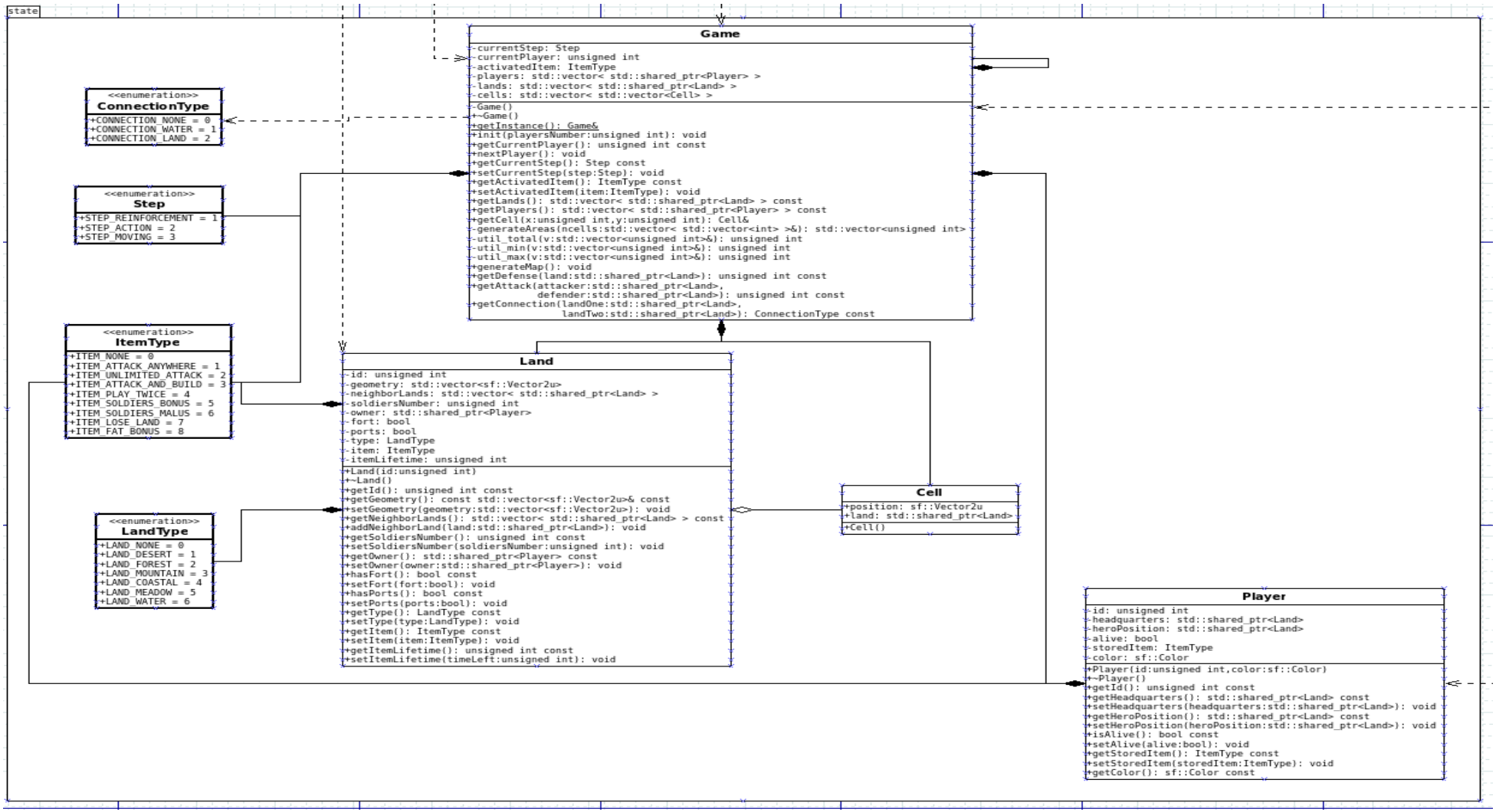
- getConnection : fonction permettant de déterminer si deux territoires sont reliés et comment (par la terre ou par la mer), renvoie un objet énumération ConnectionType
- getDefense : fonction permettant de déterminer la défense totale présente sur un territoire : établit la liste des territoires adjacents ou reliés par la mer et somme leurs nombres de troupes pondérés par 0,5 si par la mer, le tout en tenant compte de la présence éventuelle de héros
- getAttack : fonction permettant de déterminer l'attaque totale d'un territoire sur un autre : nombre de troupes du territoire attaquant, pondéré par 0,5 si par la mer, en tenant compte de la présence éventuelle de héros

Ces méthodes ne sont pas indispensables car elles sont reproductibles en utilisant uniquement des getters des différentes classes d'état, mais cela permet d'éviter la duplication de code ; elles ont leur place dans le package state puisqu'elles décrivent l'état du jeu et ne font appel qu'à d'autres getters du package state.

2.5 Ressources

La carte est générée aléatoirement à chaque partie, lors de la création de l'objet Game. Aucune ressource n'est donc nécessaire.

Illustration 1: Diagramme des classes d'état



3 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.

3.1 Stratégie de rendu d'un état

Tout l'état du jeu est accessible via le singleton Game, qui expose toutes ses données via des accesseurs. On peut donc construire une classe externe pour s'occuper du rendu.

Le rendu est effectué à 30 FPS, il n'a pas besoin d'être synchronisé avec le moteur de jeu puisque ce taux de rafraîchissement est largement supérieur à la vitesse de changement de l'état du jeu, sans consommer de ressources excessives.

Effectuer le rendu avec une classe externe à Game permet de séparer l'état de jeu de son affichage et de la gestion des ressources liées à cet affichage. Cette séparation sera bienvenue quand le jeu sera doté d'une architecture client / serveur.

3.2 Conception logiciel

Le package render est composé d'une unique classe Renderer, un singleton. Celui-ci fait appel au singleton Game du package state pour accéder aux données de l'état du jeu, et s'occupe de l'affichage ainsi que de la gestion des ressources (textures, polices, …).

La boucle événementielle est externe à cette classe Renderer. La classe Renderer a pour seul rôle de dessiner l'état du jeu. La gestion des événements ne lui reviendra pas, elle se fait donc pour le moment dans la fonction *main*. Renderer expose une méthode **void *render* (sf::RenderWindow&)** appelée dans la boucle événementielle pour le dessin du jeu.

3.3 Conception logiciel : extension pour les animations

3.4 Ressources

Numéro du joueur	**Couleur distinctive**	**Son distinctif**
1	Rouge (#ff0000)	res/sounds/player1.wav
2	Cyan (#00ffff)	res/sounds/player2.wav
3	Jaune (#ffff00)	res/sounds/player3.wav
4	Violet (#8000ff)	res/sounds/player4.wav

5	Noir (#222222)	res/sounds/player5.wav
6	Rose (#ff00ff)	res/sounds/player6.wav

Type de territoire	**Texture**
Prairie	Pas de texture
Forêt	res/textures/forest.jpg
Montagne	res/textures/mountain.jpg
Désert	res/textures/desert.jpg
Côtier	res/textures/coastal.jpg

Chaque territoire sera rendu avec une texture selon son type, filtrée par une couleur selon le joueur qui possède le territoire. Les territoires neutres auront la couleur blanche.

Les étendues d'eau n'auront pas de texture mais seront de couleur bleu (#0000ff).

Les ports seront matérialisés par une série de carrés noirs (représentant des pontons) disposés le long du littoral du territoire concerné, côté eau.

3.5 Exemple de rendu

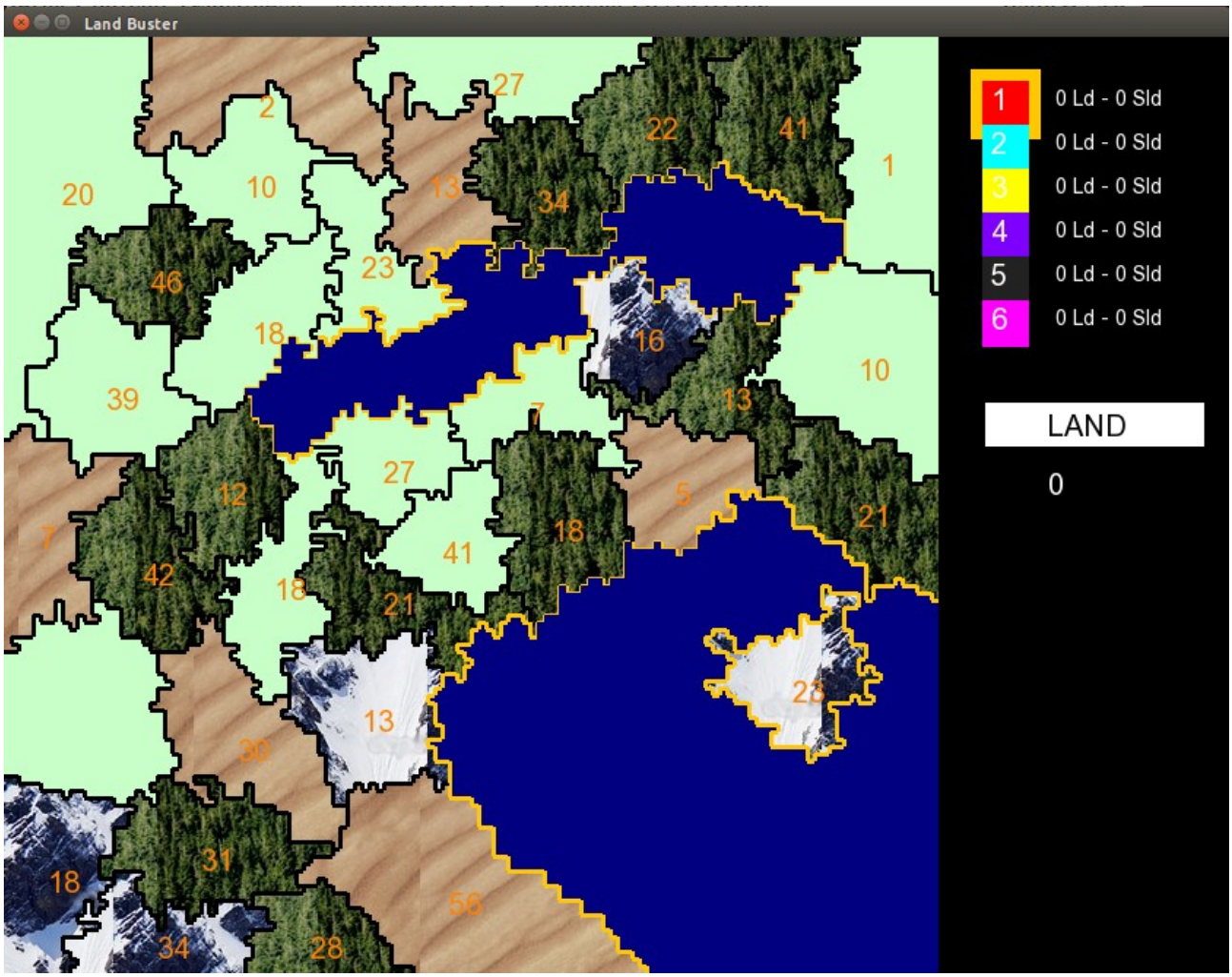
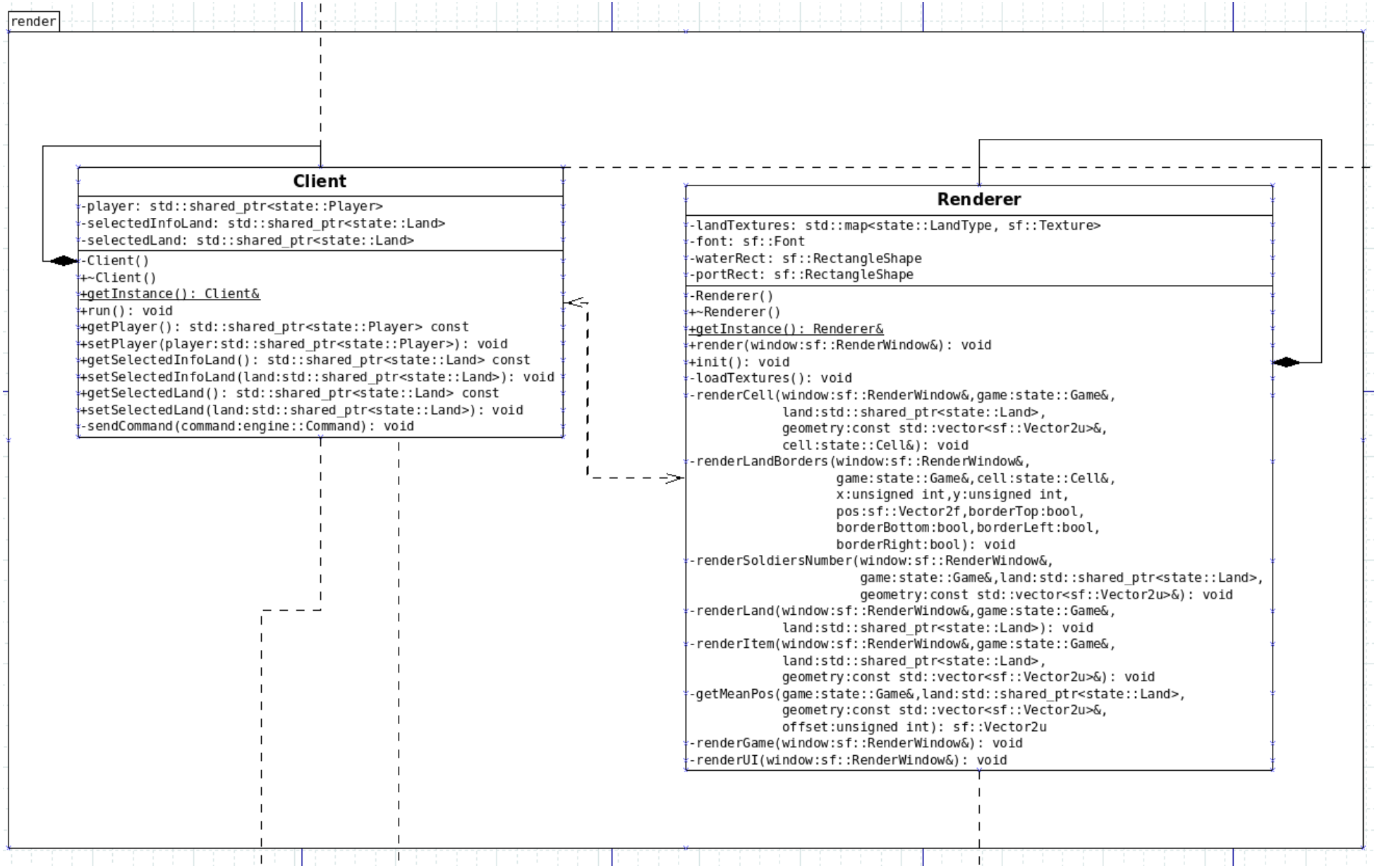

Land Buster
1
0 Ld - 0 Sld
2
0 Ld - 0 Sld
3
0 Ld - 0 Sld
4
0 Ld - 0 Sld
5
0 Ld - 0 Sld
6
0 Ld - 0 Sld
LAND
0

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 jeu est purement tour par tour, son état ne dépend absolument pas du temps, il n'y a donc aucune horloge.

4.2 Changements extérieurs

Chaque joueur peut en permanence accéder aux statistiques joueurs (liste des joueurs avec leur numéro, leur couleur, le nombre de territoires qu'ils contrôlent et le nombre de soldats que cela totalise, ainsi que le joueur en train de jouer). Chaque joueur peut également en permanence demander des informations sur un territoire donné : joueur qui possède le territoire le cas échéant, nombre de soldats présents et donc pouvant être capturés dans le cas d'un territoire neutre, sinon défense totale et attaque totale si le territoire appartient à un autre joueur. La défense totale représente la capacité de défense du territoire (nombre de soldats présents sur le territoire et sur les territoires adjacents ou accessibles par la mer appartenant au même joueur), l'attaque totale représente la capacité d'attaque du joueur sur le territoire (nombre de soldats présents sur le territoire d'où il attaque). La requête de ces informations n'est pas un changement d'état, c'est une action qui ne concerne que le joueur lui-même et qui sera gérée uniquement du côté IHM. Toutefois la classe d'état Game exposera les méthodes adéquates pour récupérer ces informations, ce qui permettra par exemple à une IA d'y requérir également.

Les changements extérieurs qui peuvent survenir sont :
- choix d'un territoire capitale (en début de jeu) / paramètres : numéro du joueur, territoire concerné
- choix d'un territoire où apporter les renforts / paramètres : numéro du joueur, territoire concerné
- construction d'un port sur un territoire / paramètres : numéro du joueur, territoire concerné
- passer son tour / paramètres : numéro du joueur
- abandon d'un joueur / paramètres : numéro du joueur
- attaque d'un territoire (si le territoire est neutre il est capturé, sinon une bataille est livrée) / paramètres : numéro du joueur qui attaque, territoire d'où il attaque, territoire qu'il attaque
- déplacement de soldats d'un territoire à l'autre / paramètres : numéro du joueur, territoire d'origine, territoire de destination
- utilisation d'un item en attente / paramètres : numéro du joueur

L'état de jeu contient les informations sur le joueur dont le tour est en cours et la phase de tour dans laquelle il est, ce qui limite les changements licites. Seuls ceux-ci seront exécutés par le moteur de jeu, et celui-ci incrémentera de manière adéquate la phase ou le joueur courant.

4.3 Changements autonomes

Les changements autonomes sont :
- expiration d'un item en attente
- apparition d'un item sur un territoire

En outre, des changements peuvent être conséquence d'un changement extérieur :
- victoire/défaite d'un joueur

4.4 Conception logiciel

Le jeu est divisé en deux grandes parties :

- le client, qui gère la conversion des entrées utilisateur (événements SFML) en commandes pour le moteur de jeu, et qui gère le rendu de l'état de jeu
- le moteur de jeu, qui gère la vérification de la validité des commandes et qui les exécute en modifiant l'état de jeu

Les deux parties ont en commun l'état de jeu, mais seul le moteur agit dessus, le client ne fait que le lire pour effectuer le rendu.

Le gameplay est tel que toute action qu'on puisse faire concerne toujours simplement un à deux territoires. On peut donc décrire une commande par une simple classe composée de deux identifiants de territoires. Comme il est prévu par la suite de placer le moteur de rendu et le moteur de jeu dans deux threads séparés, puis avec chacun sa version de l'état du jeu, nous avons décidé de désigner les éléments de jeu par un identifiant numérique plutôt que par un pointeur direct.

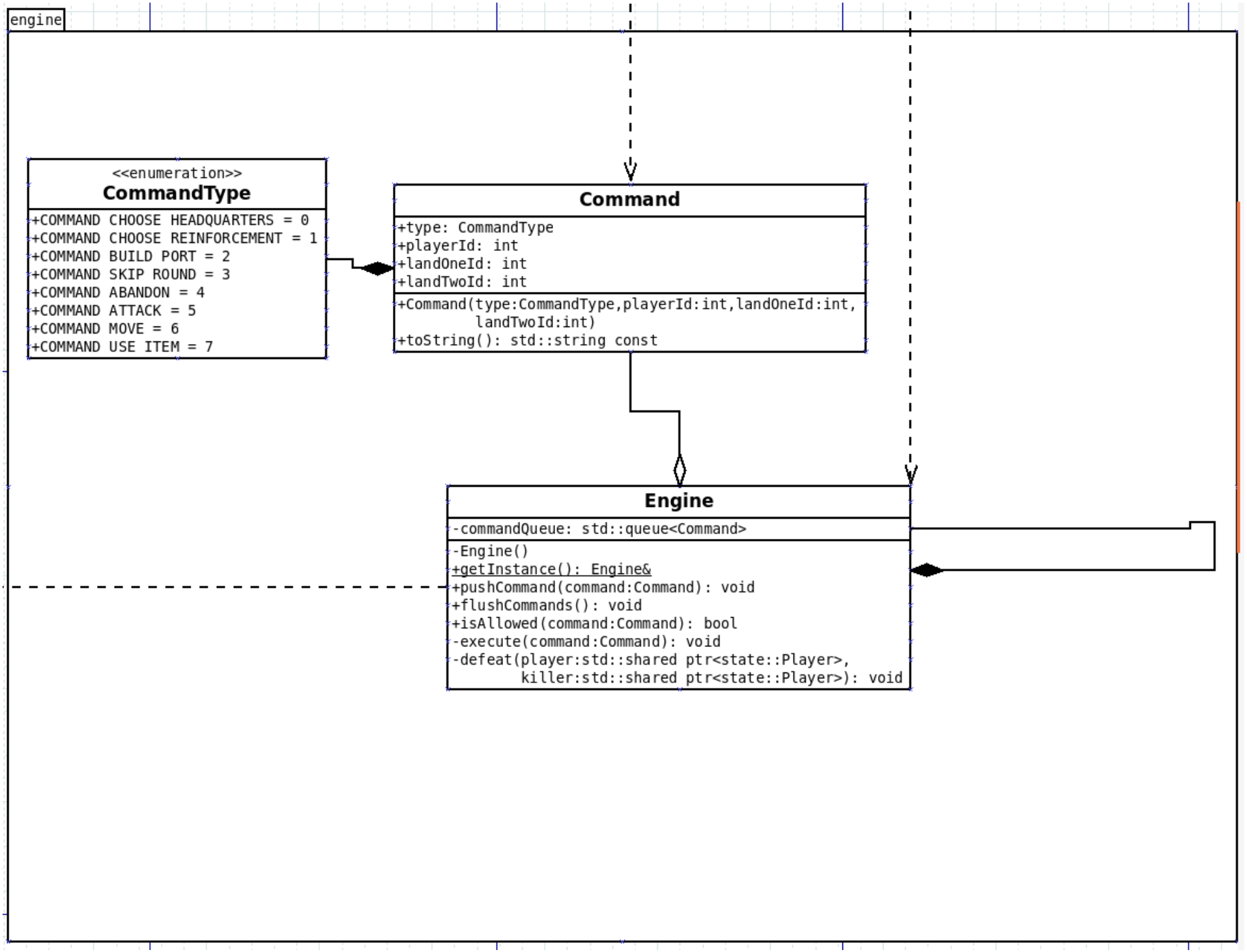
4.5 Conception logiciel : extension pour l'IA

La séparation des événements clavier/souris et des commandes pour le moteur de jeu permettra d'insérer des joueurs IA, qui enverront directement des objets de notre classe Command au moteur de jeu comme le ferait le client d'un utilisateur humain.

4.6 Conception logiciel : extension pour la parallélisation

Comme dit précédemment, nous avons désigné les éléments de l'état de jeu par un identifiant numérique, une solution indépendante de la disposition en mémoire des éléments. Cela permettra de manipuler l'état de jeu sur plusieurs machines différentes.

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

5.1.1 Intelligence minimale

Le niveau le plus simple d'intelligence artificielle proposé est comme suit :
- tant qu'il y a un territoire limitrophe on essaye de le récupérer
- lorsqu'il n'y a plus de territoire neutre libre, on attaque les territoires adverses

5.1.2 Intelligence basée sur des heuristiques

5.1.3 Intelligence basée sur les arbres de recherche

L'IA complexe implémente l'algorithme Min Max avec une profondeur de 4 coups.

5.2 Conception logiciel

Toutes les formes d'IA héritent de la classe abstraite *ai::AI* et implémentent notamment la méthode *run* qui retourne une commande à envoyer au moteur et qui est le coup joué par l'IA.

L'IA minimale est implémentée par la classe *ai::DumbAI*.
L'IA complexe, basée sur l'algorithme Min-Max, est implémentée par la classe *ai::MinMaxAI*.

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 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

6.1.1 Répartition sur différents threads

L'objectif ici est de placer plusieurs threads sur le jeu : un thread sur le moteur du jeu en plus du thread principal présent dans le *main*.
Les commandes sont les informations qui transitent le plus d'un module à un autre. Lorsque l'on parle de commande, il s'agit bien évidemment des commandes de jeux. Elles peuvent provenir soit d'une IA soit d'un *event user* (appui sur clavier, clic sur souris) lors d'un tour.
Jusqu'à présent on avait dans le déroulé un envoi de commandes, une mise à jour de l'état et une mise à jour graphique. Avec le multithreading, à la place de la mise à jour de l'état on utilise une fifo qui servira de zone tampon entre le thread principal et le thread moteur. En effet une fois les commandes envoyées, on les enfile / défile dans un fifo pour être utilisées dans le moteur, il est toutefois possible de cloner la fifo pour en avoir deux et être ainsi moins embêté s'il y a besoin de lecture et d'écriture au même moment.
Pour résumer, le thread principal fait tourner le jeu et le rendu et le thread moteur exécute les commandes envoyées dans la fifo par le thread principal.

Dans notre programme, nous avons en fait un thread pour le client (qui gère le rendu et les entrées utilisateur) et un thread pour le moteur qui traite les commandes et met à jour les données de jeu. Les données partagées sont la file de commandes du moteur, et l'état de jeu, qui ont donc chacun un mutex. Le moteur boucle en permanence et s'il y a des nouvelles commandes dans la file, il les évalue. La vérification qu'il y a de nouvelles commandes ne nécessite pas l'utilisation du mutex car c'est une pure lecture. Le client boucle tant qu'on ne ferme pas la fenêtre, c'est la boucle événementielle. Pour l'envoi des commandes il a besoin de pousser des éléments dans la file de commandes du moteur donc il utilise le mutex du moteur. L'état de jeu n'existe qu'en un exemplaire, utilisé par le client pour savoir quelle commande envoyer en fonction des événements clavier / souris et pour le rendu, qui tourne à 30 FPS. Il y a donc aussi un mutex sur l'état de jeu.
Une amélioration possible, comme dit précédemment, est de faire un *double buffering* sur la file de commandes : ainsi quand un thread utilise l'une des deux files, l'autre peut utiliser l'autre file, les deux étant interverties régulièrement.
N'avoir qu'un seul état de jeu provoque des blocages entre les threads car tous deux l'utilisent intensivement, ce qui nous a obligés à introduire des délais dans les boucles. Si nous avions un pattern Observer cela pourrait grandement faciliter les choses. Nous en aurons par ailleurs besoin pour limiter les échanges réseau lorsque le jeu sera sur plusieurs machines et que le client devra synchroniser son propre objet état sur celui du serveur.

6.1.2 Répartition sur différentes machines

On a maintenant une architecture clients-serveur.
Chaque client contient une IHM, un état de jeu et un moteur de jeu. Quand le joueur fait une action, au lieu d'envoyer la commande dans le moteur local, elle est envoyée via l'API HTTP au serveur.
Le serveur stocke un historique de toutes les commandes validées de tous les clients connectés, c'est-à-dire de tous les joueurs dans la partie.
Les clients requêtent à intervalles réguliers le serveur pour récupérer les commandes dans l'historique, que leur moteur local exécute afin de mettre à jour le rendu.
Le serveur contient un état de jeu, ainsi qu'un moteur de jeu pour vérifier la licité des commandes.

A noter qu'il n'y a que des commandes qui transitent sur le réseau, et pas de « notifications de rendu », car chaque client et chaque serveur embarque son propre moteur de jeu, pour chaque client la mise à jour de l'état de jeu et du rendu ne se fait que localement. Donc finalement on peut toujours se passer de pattern Observer.

Prise en charge des joueurs

Le serveur sera mono-partie (comme la majorité des jeux open source : Teeworlds, Urban Terror…) et non pas multi-partie (comme League of Legends).
Le serveur démarre dans un état où il n'y a pas de partie en cours. Un premier client doit envoyer une requête pour créer une partie, dans laquelle il spécifie le nombre de joueurs et les IA.
Le serveur lance alors l'initialisation de la partie (dont la génération de la carte) et attend que les autres joueurs humains rejoignent la partie.
Lorsque tous les joueurs ont rejoint, le jeu peut commencer.

Prise en charge des événements aléatoires
Certains événements ne sont pas la conséquence d'une commande mais surviennent spontanément, par exemple l'apparition d'un item sur un territoire. En réalité ces événements sont pseudo-aléatoires car ils dépendent d'une graine associée à chaque état de jeu et calculée en fonction des données de l'état et de la graine précédente. Chaque client ayant son propre exemplaire de l'état de jeu, synchronisé avec celui du serveur, il est donc via son moteur naturellement en mesure de reproduire ces événements sans devoir être notifié par le serveur.

Prise en charge de la fin de partie
Lorsqu'une commande provoque la fin de partie, le serveur cesse d'accepter les commandes et se replace en état initial (pas de partie en cours). De même que pour les événements aléatoires, chaque client a son propre exemplaire de l'état de jeu et est donc en mesure de savoir que la partie est finie.

Description de l'API HTTP :

Les données envoyées ou reçues sont présentées dans le format d'un schéma de validation JSON.

A chaque fois que le serveur attend du client des données dans sa requête, si celles-ci ne respectent pas le format attendu, le serveur renvoie une erreur 400 (Bad Request).

La documentation de l'API figure ci-dessous.

get_status

Requête
Description : permet d'obtenir le statut du serveur pour savoir si une partie est en cours et si l'initialisation de la partie (dont la génération de carte) est terminée ou non, et pour recevoir si elle est terminée le numéro de joueur qui a été attribué au client et l'état du jeu actuel complet. Pour ne pas donner d'avantage à l'hôte de la partie (qui sera probablement le premier à lancer son client), les numéros de joueur ne sont pas distribués dans l'ordre mais tirés aléatoirement.
Méthode : GET
URI : /status/
Données envoyées : rien

Réponse 1
Description : réponse envoyée si aucune partie n'est en cours (état de départ)
Statut : 503 (Service Unavailable)
Données reçues : rien

Réponse 2
Description : réponse envoyée si une partie a été créée mais que l'initialisation n'est pas terminée ou que la partie attend encore des joueurs
Statut : 204 (No Content)
Données reçues : rien

Réponse 3
Description : réponse envoyée si une partie est en cours et que l'initialisation est terminée
Statut : 200 (OK)
Données reçues :

```
{
« $schema » : « http://json-schema.org/draft-04/schema# »,
« type » : « object »,
« properties » : {
        « playerId » : { « type » : « number », « minimum » : 0, « maximum » : 6 },
        « state » : {
                « type » : « object »,
                « properties » : {
                        « currentStep » : { « type » : « number », « minimum » : 0, « maximum » : 2 },
                        « currentPlayer » : { « type » : « number », « minimum » : 0, « maximum » : 6 },
                        « activatedItem » : { « type » : « number », « minimum » : 0, « maximum » : 8 },
                        « players » : {
                                « type » : « array »,
                                « minItems » : 0,
                                « maxItems » : 6,
                                « items » : {
                                        « type » : « object »,
                                        « properties » : {
                                                « id » : { « type » : « number », « minimum » : 0,
```

```
« maximum » : 6 },
                                        « headquarters » :  {  « type » :  « number »,
« minimum » : 0, « maximum » : 9999 },
                                        « heroPosition » :  {  « type » :  « number »,
« minimum » : 0, « maximum » : 9999 },
                                        « deadHero » : { « type » : « boolean » },
                                        « alive » : { « type » : « boolean » },
                                        « storedItem » : { « type » : « number », «minimum » :
0, « maximum » : 8 },
                                        « color » : {
                                                « type » : « object »,
                                                « properties » : {
                                                        « r » :  {  « type » :  « number »,
« minimum » : 0, « maximum » : 255 },
                                                        « g » :  {  « type » :  « number »,
« minimum » : 0, « maximum » : 255 },
                                                        « b » :  {  « type » :  « number »,
« minimum » : 0, « maximum » : 255 }
                                                },
                                                « required » : [« r », « g », « b »]
                                        }
                                },
                                required :  [« id »,  « deadHero »,  « alive »,  « storedItem »,
« color »]
                        }
                },
                « lands » : {
                        « type » : « array »,
                        « minItems » : 0,
                        « maxItems » : 9999,
                        « items » : {
                                « type » : « object »,
                                « properties » : {
                                        « id » :  {  « type » :  « number »,  « minimum » :  0,
« maximum » : 9999 },
                                        « neighborLands » : {
                                                « type » : « array »,
                                                « minItems » : 0,
                                                « maxItems » : 9999,
                                                « items » :  {  « type » :  « number »,
« minimum » : 0, « maximum » : 9999 },
                                        },
                                        « soldiersNumber » :  {  « type » :  « number »,
« minimum » : 0, « maximum » : 99999 },
                                        « owner » : { « type » : « number », « minimum » : 0,
« maximum » : 6 },
                                        « fort » : { « type » : « boolean » },
                                        « ports » : { « type » : « boolean » },
                                        « type » :  {  « type » :  « number »,  « minimum » :  0,
« maximum » : 6 },
```

« item » : { « type » : « number », « minimum » : 0, « maximum » : 8 },
« itemLifeTime » : { « type » : « number », « minimum » ; 0, « maximum » : 99 },
},
« required » : [« id », « neighborLands », « soldiersNumber », « fort », « ports », « type », « item », « itemLifeTime »]
}
},
« cells » : {
« type » : « array »,
« minItems » : 0,
« maxItems » : 99999,
« items » : {
« type » : « object »,
« properties » : {
« x » : { « type » : « number », « minimum » : 0, « maximum » : 999 },
« y » : { « type » : « number », « minimum » : 0, « maximum » : 999 },
« land » : { « type » : « number », « minimum » : 0, « maximum » : 9999 },
},
« required » : [« x », « y », « land »]
}
},
},
},
« required » : [« playerId », « state »],
}

Réponse 4
Description : erreur envoyée si il n'y a plus de place pour un nouveau joueur
Statut : 403 (Forbidden)
Données reçues : rien

create_game

Requête
Description : permet de demander la création d'une nouvelle partie, en spécifiant le nombre de joueurs désirés et les caractéristiques des éventuelles IA
Méthode : PUT
URI : /status/
Données envoyées :

{
« $schema » : « http://json-schema.org/draft-04/schema# »,
« type » : « object »,
« properties » : {
« numPlayers » : { « type » : « number », « minimum » : 2, « maximum » : 6 },

```
		« ais » : {
			« type » : « array »,
			« minItems » : 0,
			« maxItems » : 5,
			« items » : { « type » : « string », « pattern » : « (dumb|heuristic|minmax) » }
		}
	},
	« required » : [« numPlayers »]
}
```

Réponse 1
Description : le serveur n'a pas de partie en cours ou en initialisation et a donc accepté la demande et initié une nouvelle partie
Statut : 200 (OK)
Données reçues : rien

Réponse 2
Description : une partie est en cours ou en initialisation et la demande est donc rejetée
Statut : 403 (Forbidden)
Données reçues : rien

Réponse 3
Description : les données envoyées ne respectent pas le schéma de validation ou sont incorrectes, la requête est rejetée
Statut : 400 (Bad Request)
Données reçues : rien

push_command

Il y a plusieurs requêtes possibles, une par type de commande. Toutes les URI renvoient les mêmes réponses.

Requête 1
Description : permet d'envoyer une nouvelle commande action au serveur
Méthode : PUT
URI : /commands/action/
Données envoyées :

```
{
	« $schema » : « http://json-schema.org/draft-04/schema# »,
	« type » : « object »,
	« properties » : {
		« type » : { « type » : « number », « minimum » : 0, « maximum » : 7 },
		« playerId » : { « type » : « number », « minimum » : 0, « maximum » : 6 },
	},
	« required » : [« type », « playerId]
}
```

Requête 2

Description : permet d'envoyer une nouvelle commande choix au serveur
Méthode : PUT
URI : /commands/choice/
Données envoyées :

```
{
	« $schema » : « http://json-schema.org/draft-04/schema# »,
	« type » : « object »,
	« properties » : {
		« type » : { « type » : « number », « minimum » : 0, « maximum » : 7 },
		« playerId » : { « type » : « number », « minimum » : 0, « maximum » : 6 },
		« landId » : { « type » : « number », « minimum » : 0, « maximum » : 9999 }
	},
	« required » : [« type », « playerId », « landId »]
}
```

Requête 3
Description : permet d'envoyer une nouvelle commande attaque au serveur
Méthode : PUT
URI : /commands/attack/
Données envoyées :

```
{
	« $schema » : « http://json-schema.org/draft-04/schema# »,
	« type » : « object »,
	« properties » : {
		« type » : { « type » : « number », « minimum » : 0, « maximum » : 7 },
		« playerId » : { « type » : « number », « minimum » : 0, « maximum » : 6 },
		« interactions » : {
			« type » : « array »,
			« minItems » : 0,
			« maxItems » : 99999,
			« items » : {
				« type » : « object »,
				« properties » : {
					« landOne » : { « type » : « number », « minimum » : 0,
« maximum » : 9999 },
					« landTwo » : { « type » : « number », « minimum » : 0,
« maximum » : 9999 }
				},
				« required » : [« landOne », « landTwo »]
			}
		}
	},
	« required » : [« type », « playerId », « interactions »]
}
```

Requête 4
Description : permet d'envoyer une nouvelle commande mouvement au serveur
Méthode : PUT

URI : /commands/move/
Données envoyées :

```
{
	« $schema » : « http://json-schema.org/draft-04/schema# »,
	« type » : « object »,
	« properties » : {
		« type » : { « type » : « number », « minimum' : 0, « maximum » : 7 },
		« playerId » : { « type » : « number », « minimum » : 0, « maximum » : 6 },
		« interaction » : {
			« type » : « object »,
			« properties » : {
				« landOne » : { « type » : « number », « minimum » : 0, « maximum » : 9999 },
				« landTwo » : { « type » : « number », « minimum » : 0, « maximum » : 9999 }
			},
			« required » : [« landOne », « landTwo »]
		},
		« ratio » : { « type » : « number », « minimum » : 0.0, « maximum » : 1.0 },
		« hero » : { « type » : « boolean » }
	},
	« required » : [« type », « playerId », « interaction », « ratio », « hero »]
}
```

Réponse 1
Description : réponse envoyée si la commande est jugée licite et acceptée
Statut : 200 (OK)
Données reçues :

Le serveur renvoie l'index de la commande dans l'historique, ce qui permet ensuite au client et au serveur de l'identifier de manière unique.

```
{
	« $schema » : « http://json-schema.org/draft-04/schema# »,
	« type » : « number »,
	« minimum » : 0,
	« maximum » : 99999
}
```

Réponse 2
Description : erreur envoyée si la commande est jugée illicite et rejetée
Statut : 403 (Forbidden)
Données reçues : rien

Réponse 3
Description : erreur envoyée si le serveur n'a pas de partie en cours ou si l'initialisation n'est pas terminée, la requête est donc rejetée
Statut : 503 (Service Unavailable)
Données reçues : rien

Réponse 4
Description : les données envoyées ne respectent pas le schéma de validation ou sont incorrectes, la requête est rejetée
Statut : 400 (Bad Request)
Données reçues : rien

get_commands

Requête
Description : permet de récupérer toutes les commandes à partir d'un certain index dans l'historique, ainsi le client n'est pas obligé de recharger l'historique complet à chaque fois mais ne charger que ce qui lui manque
Méthode : GET
URI : /commands/<index : number, minimum 0>/
Données envoyées : rien

Réponse 1
Description : erreur envoyée si l'index est invalide
Statut : 400 (Bad Request)
Données reçues : rien

Réponse 2
Description : réponse envoyée si l'index est valide, contenant la liste des commandes dans l'historique depuis cet index (exclus). Le serveur envoie toutes les commandes dans l'historique, sauf celles que le client a lui-même envoyées précédemment. Cela permet d'économiser un peu de bande passante, charge au client de stocker l'index de ses commandes acceptées afin de tout exécuter dans l'ordre.
Statut : 200 (OK)
Données reçues :

```
{
	« $schema » : « http://json-schema.org/draft-04/schema# »,
	« type » : « array »,
	« minItems » : 0,
	« maxItems » : 9999,
	« items » : {
		« type » : « object »,
		« properties » : {
			« index » : { « type » : « number », « minimum » : 0, « maximum » : 99999 },
			« type » : { « type » : « number », « minimum » : 0, « maximum » : 7 },
			« playerId » : { « type » : « number », « minimum » : 0, « maximum » : 6 },
			« landId » : { « type » : « number », « minimum » : 0, « maximum » : 9999 },
			« interactions » : {
				« type » : « array »,
				« minItems » : 0,
				« maxItems » : 99999,
				« items » : {
					« type » : « object »,
					« properties » : {
```

« landOne » : { « type » : « number », « minimum » : 0, « maximum » : 9999 },
« landTwo » : { « type » : « number », « minimum » : 0, « maximum » : 9999 }
},
« required » : [« landOne », « landTwo »]
}
},
« ratio » : { « type » : « number », « minimum » : 0.0, « maximum » : 1.0 },
« hero » : { « type » : « boolean » }
},
« required » : [« index », « type », « playerId »]
}
}

Réponse 3
Description : le serveur n'a pas de partie en cours et à l'initialisation finie, la requête est donc rejetée
Statut : 503 (Service Unavailable)
Données reçues : rien

6.2 Conception logiciel

6.3 Conception logiciel : extension réseau

Pour la modularisation réseau, le but est de pouvoir communiquer des données au format JSON via le protocole HTTP, nous utilisons pour cela la librairie jsoncpp et les librairies microhttpd côté serveur et libcurl côté client (SFML Network ne permettant pas d'utiliser toutes les méthodes HTTP dont nous avons besoin pour une API REST).

L'API requiert de pouvoir sérialiser un état complet de jeu en JSON, ainsi que de pouvoir sérialiser n'importe quelle commande. Nous avons pour cela ajouté les méthodes toJSON aux classes du package state et aux classes héritant de engine::Command, ainsi que la méthode fromJSON à la classe state::Game (qui se charge des classes associées Land, Player et Cell) et la méthode commandFromJSON à engine::Engine, qui instancie un objet commande correspondant à un objet JSON donné.

Le package server contient une classe Server qui prend en charge le démon d'écoute HTTP et traite toutes les requêtes dans la méthode queryService. L'objet Server contient un state::Game, et un moteur pour vérifier les commandes et les exécuter sur son instance de state::Game.

Le package client contient une classe Client qui contient un state::Game dont il prend en charge la synchronisation avec le serveur au moyen des méthodes request*, à l'aide de son moteur pour exécuter les commandes reçues du serveur. Il contient également un render::Client pour l'interface graphique.

Nous ne produisons qu'un seul exécutable qui peut aussi bien faire client que serveur. Pour le lancer en client, il faut le lancer sans arguments, pour le lancer en serveur il suffit d'ajouter l'argument

server dans la ligne de commande.

Le serveur écoute sur le port 8080.

Le package server contient également une énumération ServerState pour stocker l'état du serveur (inactif, en initialisation ou actif), et une classe ServiceException pour gérer l'émission d'erreurs HTTP dans la méthode queryService. Nous ne l'avons pas utilisée mais nous l'avons implémentée pour tenir compte d'éventuelles évolutions ultérieures.

Etat actuel : les méthodes de sérialisation/désérialisation JSON ont été implémentées et testées avec succès, le serveur a été implémenté mais pas testé (il faut le client), le client n'a pas été implémenté car la librairie libcurl pose problème.

Illustration 4: Diagramme de classes pour la modularisation

Pour la modularisation multithread, il n'y a pas de modification au diagramme de classes, hormis le simple ajout d'un mutex sur Engine et sur Game.

Modularisation réseau :