Projet Logiciel Transversal

Luigi CAPO-CHICHI – Yassert BOINALI

Table des matières

1 Objectif

1.1 Présentation générale

Notre projet consiste à réaliser une version simplifiée du jeu de stratégie au tour par tour « Civilization ».

1.2 Règles du jeu

Le joueur doit développer son empire en compétition avec une ou plusieurs autres civilisations dirigées par l'ordinateur (IA). Le but du jeu est d'avoir la plus importante civilisation quand le jeu s'arrête. A son tour, le joueur peut déplacer ses pièces, attaquer, échanger, découvrir de nouvelles technologies et construire de nouvelles unités militaires, colons et colonies.

Le jeu se déroule sur trois époques, la plus ancienne étant l'antiquité, suivi du moyen-âge et enfin de l'époque moderne. Chaque époque a ses propres forces militaires, améliorations de villes et technologies, et chacune est supérieure à celle de l'ère précédente.

Le jeu débute à l'ère antique. Une époque se finit quand :

- un joueur achète la 3e technologie de l'ère actuelle, ou
- un joueur achète la dernière technologie restante de l'ère actuelle.

En combinant habilement le développement économique, la force militaire, la diplomatie et les échanges profitables, le joueur peut créer une plus grande civilisation et gagner la partie.

Fin du jeu

Le jeu se termine à la fin du tour où un joueur possède 3 technologies de l'ère moderne. Quand tous les joueurs ont fini leurs achats, on compte les points de victoire. Le joueur avec le plus de points de victoire a gagné.

1.3 Ressources

L'affichage repose sur plusieurs textures qu'on va présenter ci-dessous.

Figure 1 : Textures pour les bâtiments antiques

Figure 2 : Textures pour les tuiles bâtiments du moyen-âge

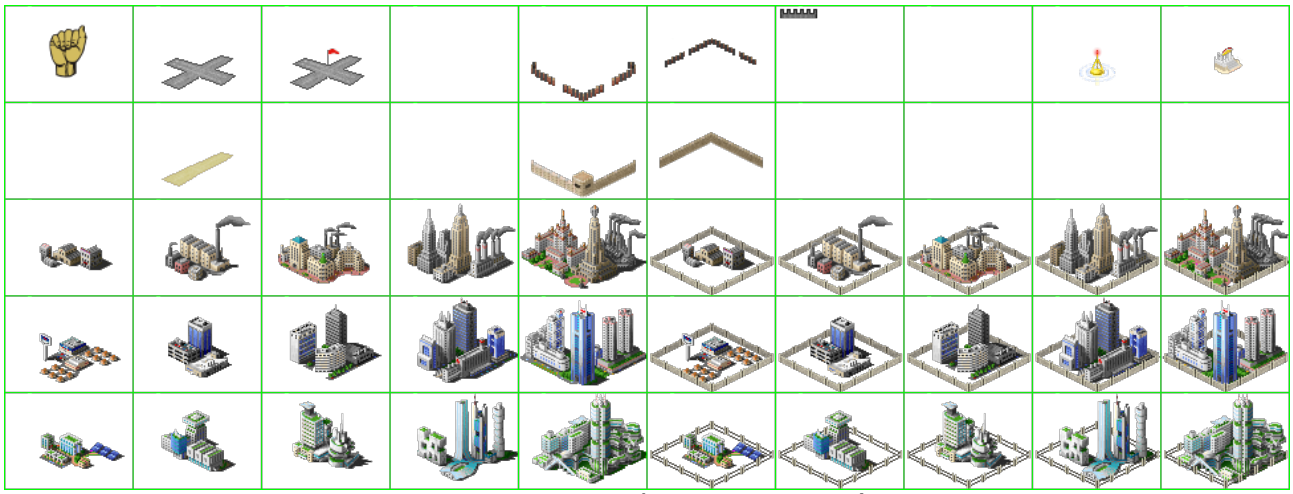


Figure 3 : Textures des bâtiments de l'ère moderne

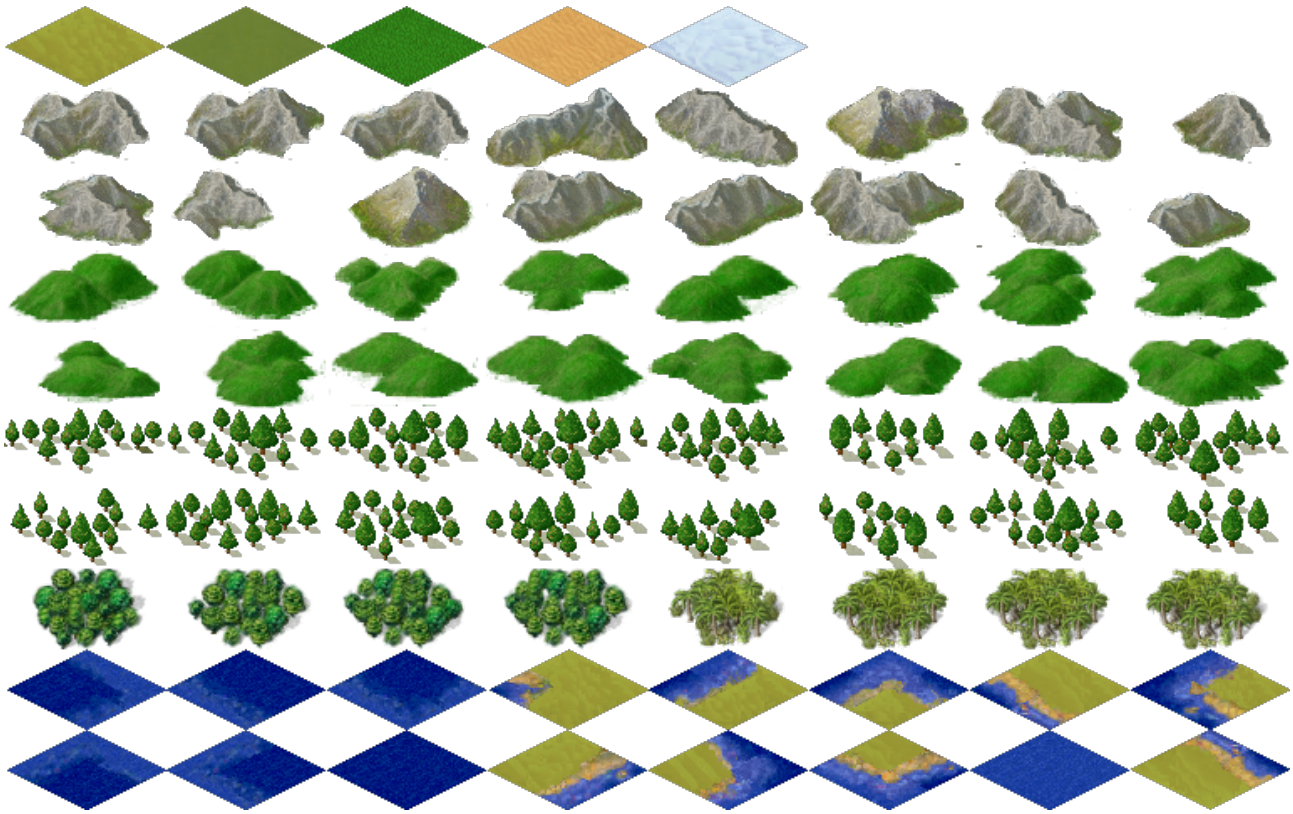


Figure 5 : Textures pour le terrain

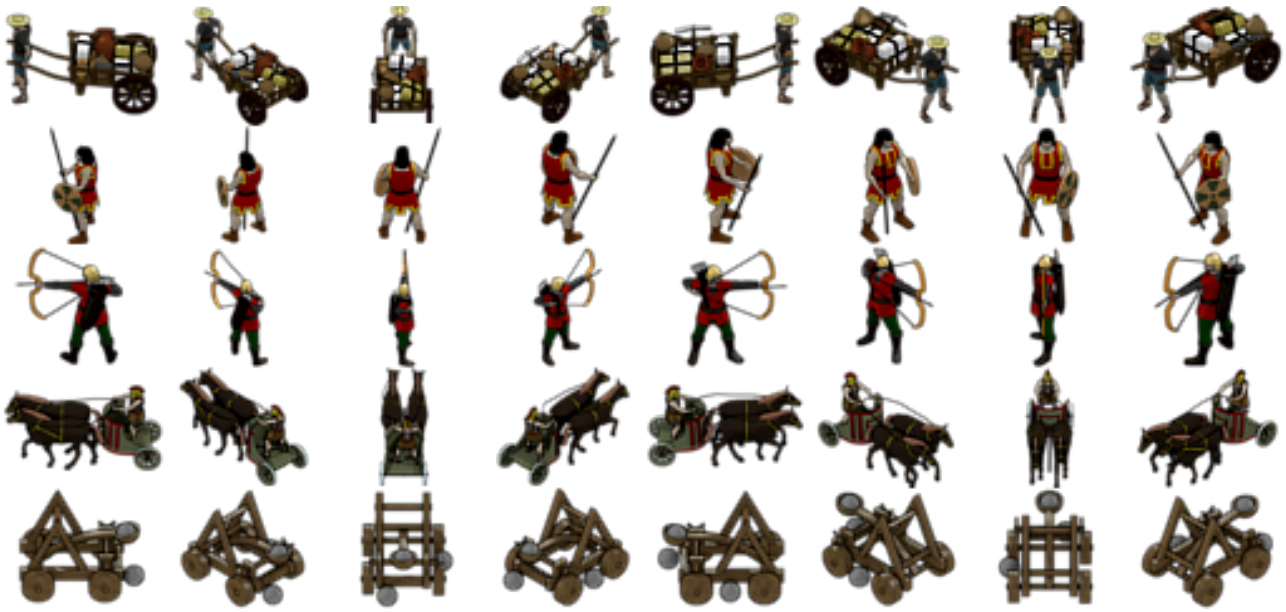


Figure 7 : Textures pour les unités

Figure 9 : Textures pour divers batiments

2 Description et conception des états

2.1 Description des états

Un état du jeu est formée par un ensemble d'éléments fixes (plateau, ressources, colonies, murs) et un ensemble d'éléments mobiles (colons, militaires, catapulte). Tous les éléments possèdent les propriétés suivantes :

- Coordonnées (x,y) dans la grille

- Identifiant du type d'élément : ce nombre indique la nature de élément (c'est à dire de la classe).

2.1.1 Etat des éléments fixes

Le plateau est formé par une grille d'éléments nommé « cases ». La taille de cette grille est fixe. Les types de cases sont :

Cases « Mur » : les cases « mur » sont des éléments infranchissables pour les éléments mobiles.

Cases « Plateau » : les cases « Plateau » vont servir à définir la texture du terrain (désert, plaine, montagne, océan).

Cases « Ressource » : elles contiennent un type de ressource.

Cases « Colonie » : ces cases vont contenir les différents types de bâtiments.

2.1.2 Etat des éléments mobiles

Les éléments mobiles possèdent une direction (aucune, gauche, droite, haut ou bas) et un coût. Chaque élément mobile peut se déplacer d'un certain nombre de cases prédéfini (PM).

Element mobile « Colon » : cet élément est dirigé par le joueur, qui commande la propriété de direction. Les colons sont les seules pièces qui peuvent explorer les régions terrestres et bâtir des colonies.

Elements mobiles « Militaire » et « Catapulte » : ces éléments sont également commandés par la propriété de direction, qu'elle proviennent d'un humain ou d'une IA. Ces éléments disposent d'un point de vie prédéterminé qui évolue lors des combats.

Unité	Epéiste	Mitrailleur	Mousquetaire	Catapulte
Points de vie	100	100	100	100
Combat	20	25	30	35

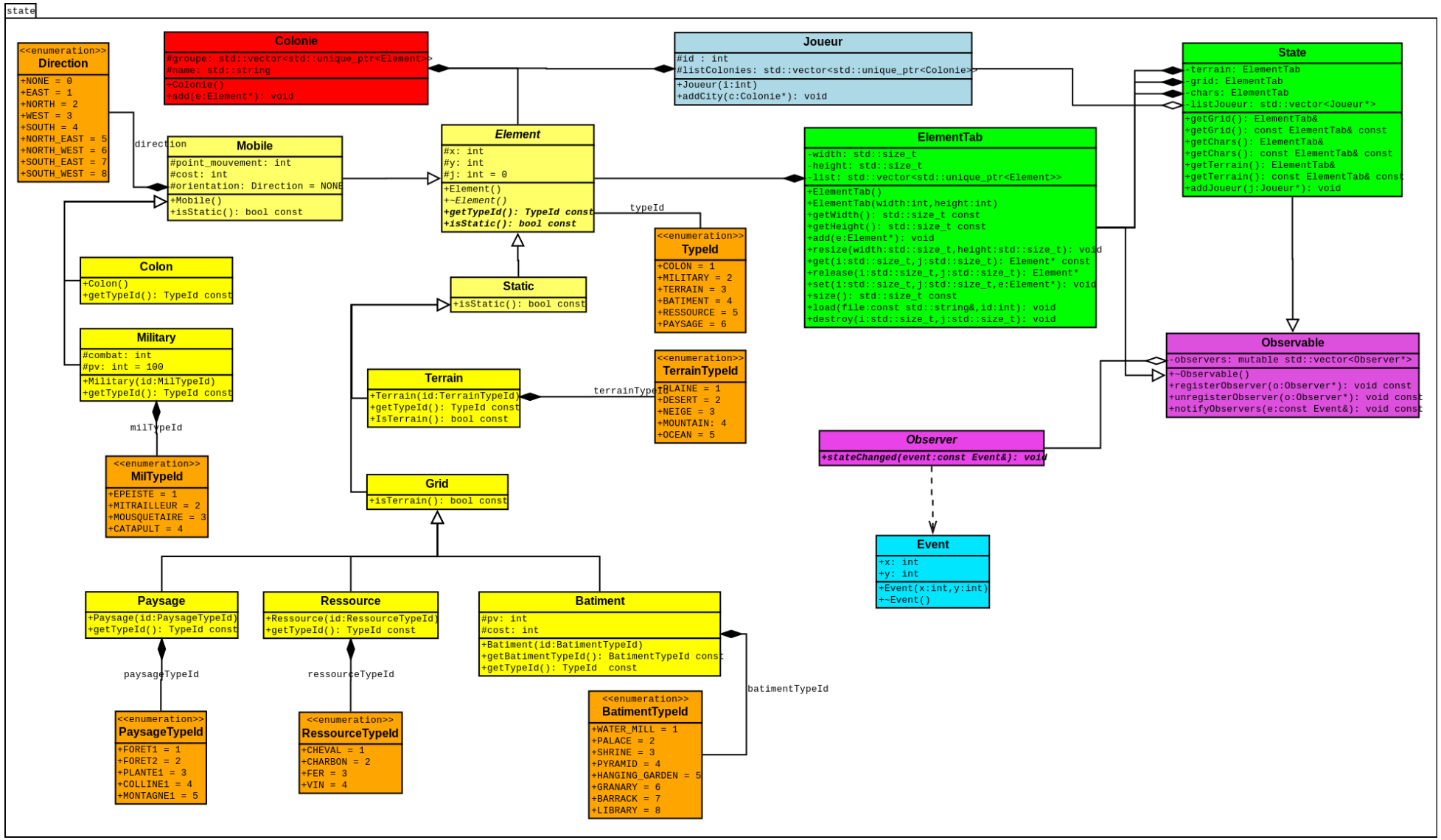
2.2 Conception logiciel

Le diagramme des classes pour les états est présenté sur la figure ci-dessous dont nous pouvons mettre en évidence les groupes de classes suivants :

Classes Element : on dispose de toute une hiérarchie de classes filles qui héritent de la classe « Element » (en jaune). Ces classes permettent de représenter les différents catégories et types d'élément.

Conteneurs d'élément : viennent ensuite les classes State et ElementTab qui permettent de contenir des ensembles d'éléments. ElementTab est un tableau en deux dimension d'éléments, par exemple pour contenir la grille des éléments. Il peut également être considéré comme un tableau à une dimension dans le cas où la hauteur (height) est égale à 1 (utile pour la liste des personnages « chars »). Enfin, la classe State est le conteneur principal, à partir duquel on peut accéder à toutes les données de l'état.

Figure 10: Diagramme des classes d'état



3 Rendu : Stratégie et Conception

3.1 Stratégie de rendu d'un état

Afin de pouvoir réaliser un rendu imagé de notre jeu Civilization, il nous faut tout d'abord réaliser un rendu des différents éléments qui composent notre jeu. Pour cela, on adopte une stratégie assez bas niveau et relativement proche du fonctionnement des unités graphiques.
Plus précisément, nous découperons la scène à rendre en layers(plans). On aura ainsi un plan pour les éléments statiques (Wall, Colonie, Bâtiment...), un autre pour les éléments mobiles tels que les unités militaires et les colons puis enfin un plan pour le score et l'arbre des technologies. Chaque plan contiendra deux informations de bas niveau à savoir une unique texture contenant les tuiles et une unique matrice avec la position des éléments et les coordonnées dans la texture. Ainsi, seuls les éléments dont les tuiles sont associés à la texture du plan seront réalisés dans notre rendu.

Pour la réalisation de ces différentes informations, la première idée est d'observer l'état à rendre et de réagir, lorsqu'une modification se produit. Si on observe un changement permanent dans le rendu, on met à jour le morceau de la matrice du plan correspondant. Pour les changements non permanents comme les éléments mobiles, il faudra modifier la matrice du plan automatiquement à chaque rendu d'une nouvelle frame.

3.2 Conception logiciel

Le diagramme d'état pour le rendu général est présenté en figure 11.

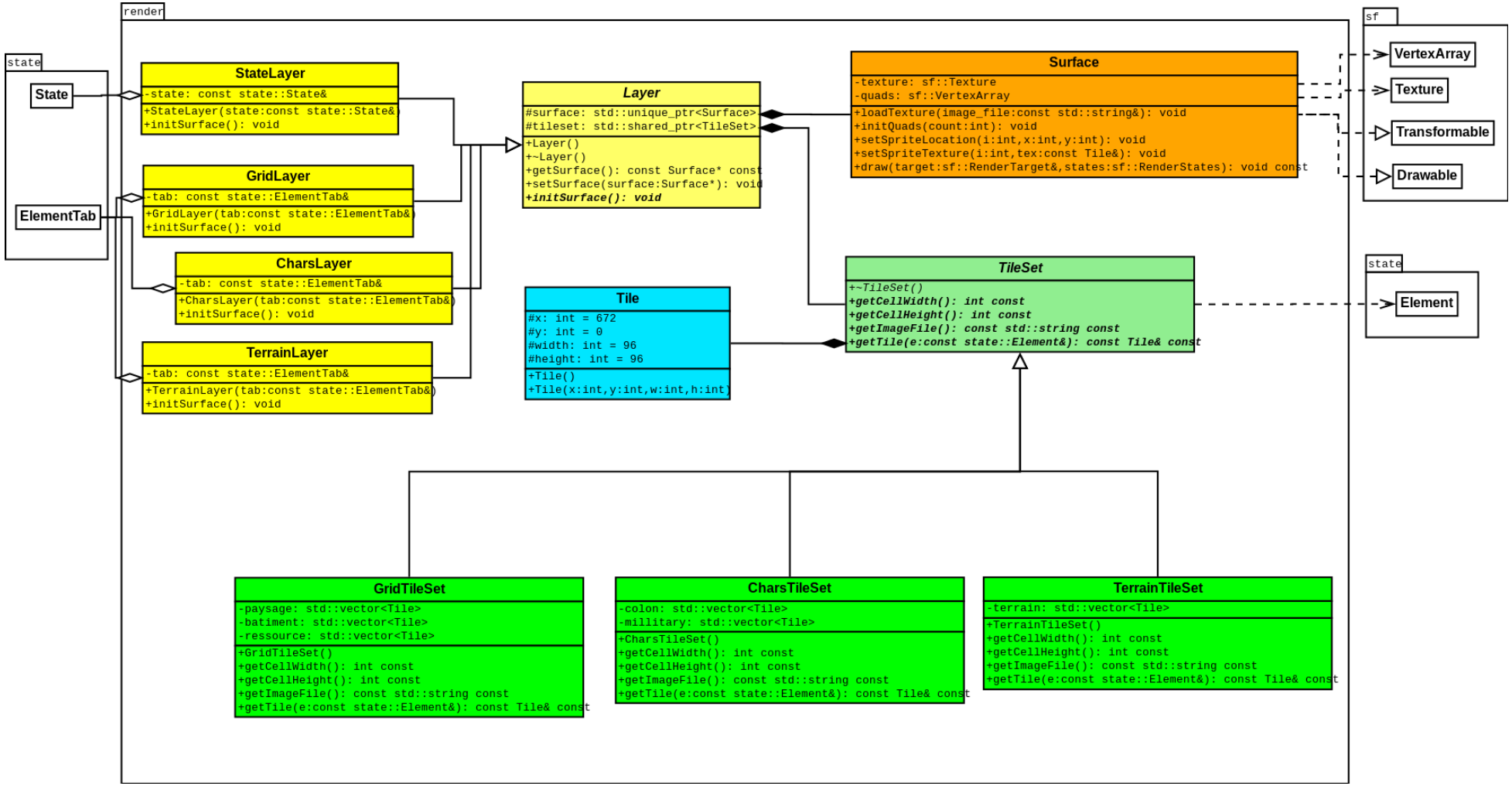
Layer: Cette classe est le moteur du rendu qu'on veut réaliser.Le principales but des objets de cette classe est de former des éléments basiques qu'on pourra transmettre à notre carte graphique par l'intermédiaire d'une classe surface.Les classes filles de cette classe permettent de définir le s différents plans de notre jeu.Ainsi la classe StateLayer permettra de réaliser le plan contenant les différentes informations, et la classe ElementTabLayer constituera par exemple un plan pour les différents personnages de notre jeu.

La méthode initSurface() de cette classe permettra de créer un surface, de changer sa texture et d'initialiser la liste des sprites.

Tuiles: La classe TileSet permet de définir les tuiles des différents éléments de notre jeu.Ces classes filles regroupent toutes les définitions des tuiles d'un même plan.La classe StateTileSet pour les informations au niveau du jeu, ma classe GridTileSet pour les éléments statiques et la classe CharsTileSet pour les éléments mobiles.

Surface: Chaque surface contient une texture du plan et une liste de quadruplet de vecteurs.

Figure 11: Diagramme de classes pour le rendu



4 Règles de changement d'états et moteur de jeu

4.1 Changements extérieurs

Les changements extérieurs sont provoqués par des commandes extérieurs, comme la pression sur une touche ou un ordre provenant du réseau :
— Commandes principales : « Charger un niveau » : On fabrique un état initial à partir d'un fichier
— Commandes « Orientation personnage », paramètres « personnage », « direction » : Si cela est possible (pas de mur ni océan), un personnage se déplace toujours selon orientation.

4.2 Changements autonomes

- Si un élément mobile est dans une case ressource, on incrémente le compteur de la ressource visée
- Lors d'un changement d'époque, on met à jour le niveau pour correspondre à la nouvelle époque. Une époque se finit quand :
  - un joueur achète la 3e technologie de l'ère actuelle, ou
  - un joueur achète la dernière technologie restante de l'ère actuelle.

La nouvelle époque commence au début du tour suivant. Les technologies et unités militaires ne sont disponibles qu'à leur époque appropriée. Quand une époque se termine, toutes ses technologies et unités militaires non possédées deviennent indisponibles.

- Lorsque deux éléments mobiles sont sur une même case, il ne se passe rien
- Lorsqu'un colon est sur une case vide, il a la possibilité de fonder une ville à cet emplacement
- Si un militaire est à 2 cases d'un ennemi, il a la possibilité de l'attaquer :
  - si c'est un colon, il peut le capturer
  - sinon il peut l'attaquer
- Une catapulte a la possibilté d'attaquer un mur si elle se trouve à proximité (à deux cases)
- Une colonie a la possibilité de porter une attaque à des unités ennemis qui sont dans son térritoire.

4.3 Conception logiciel

Le diagramme des classes pour le moteur du jeu est présenté en figure 12. 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.

Classes Command. Le rôle de ces classes est de réaliser une commande,

quelque soit sa source(automatique, clavier, réseau,...).Notons bien que 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.
A ces classes, on a défini un type de commande avec CommandTypeId pour identifier précisément la classe d'une instance.

- LoadCommand. Charge un niveau depuis un fichier
- MoveCommand. Permet de déplacer un personnage selon une direction.
- AttaqueCommand.Réalise une attaque sur une unité ennemi.
- FondationCommand. Fonde une nouvelle colonie.

Classe Action : Chaque commande lorsqu'elle est réalisée donne lieu à différenes action selon le cas.Par exemple dans otre cas, la commande attaqué permet de réaliser les actions d'attaquer en personne d'un joueur adverse, ainsi qu'à une commande pour le tuer.Cette version avancée du "Pattern Command" permet d'atomiser les opérations effectives, et donc de les inverser grâce à l'appel aux methodes undo().Ce schéma nous permet d'annuler les conséquences de commandes, et ainsi remonter dans l'arbre des états pour l'intelligence artificielle avancée.

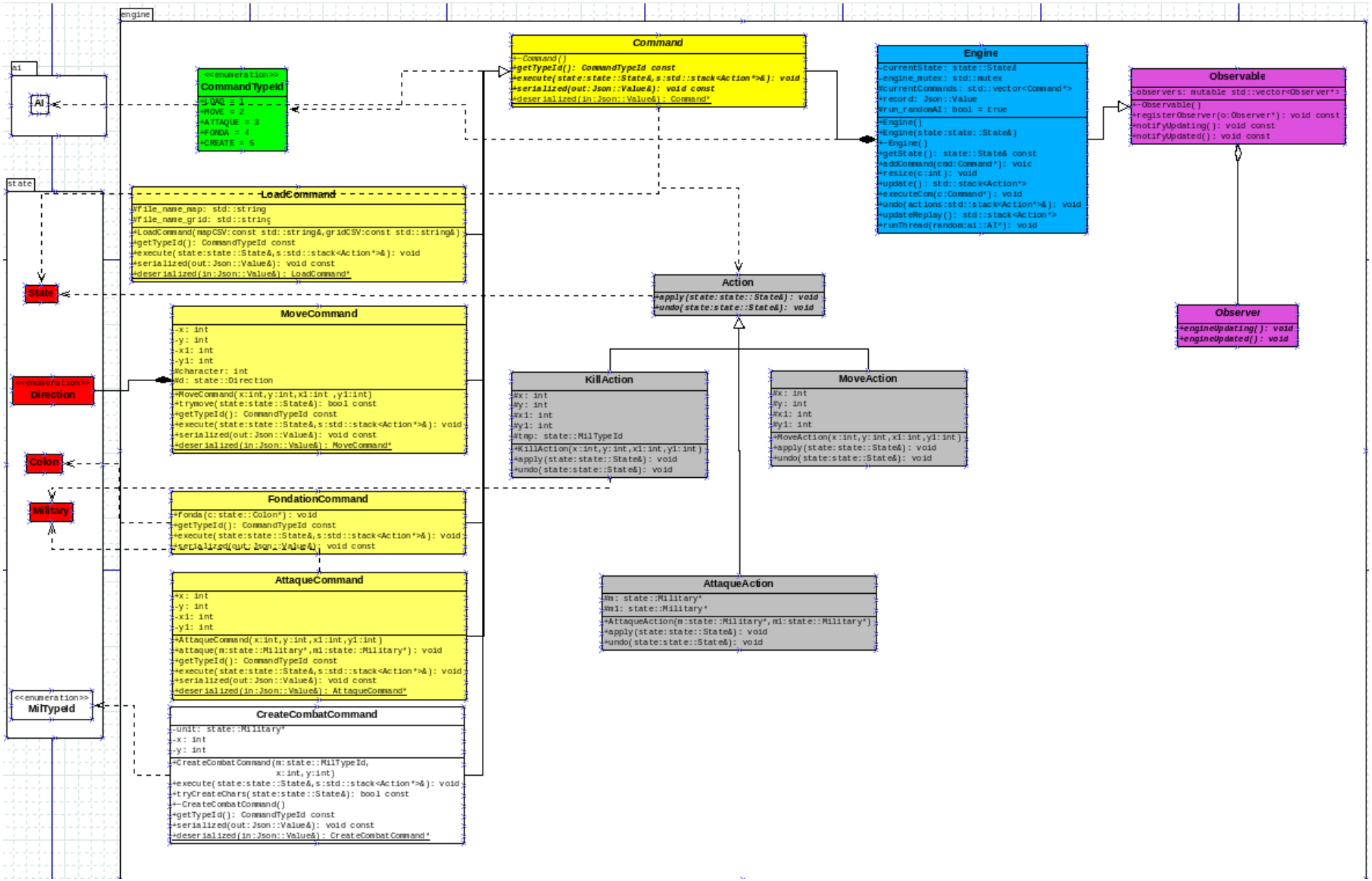
Engine. C'est le coeur du moteur. Elle stocke les commandes dans une std::vector. Lorsqu'une nouvelle époque démarre, on appelle la méthode update() va lancer les méthodes execute() de chaque commande, incrémente l'époque, puis supprime toutes les commandes.Il en résulte une pile d'actions qui, si on les appliquent en ordre inverse, permettent d'annuler les commande précédemment invoquées.

La classe Engine a été étendue pour permettre une exécution dans un thread séparé. Nous avons ajouté un mutex qui est utilisé pour bloquer lors de l' exécution d'une commande sur l'IA aléatoire.

Commande enregistrement

On doit pouvoir enrégistrer des json sur un fichier texte « replay.txt ». On réalise cela en implémentant un méthode serialized() sur les différentes commandes de notre moteur de jeu.Cette méthode permettant de transformer nos classes C++ en json.
On réalise l'opération inverse à l'aide de la méthode static deserialized() permettant ainsi de réaliser une exécution du jeu à l'aide de notre fichier « replay.txt ».

Figure 12: Diagrammes des classes pour le moteur de jeu



5 Intelligence Artificielle

5.1 Stratégies

Dans cette cinquième partie, nous définiront les différentes stratégies que peuvent appliquer notre machine pour faire fonctionner notre jeu suivant les différentes règles décrites précédemment.

5.1.1 Intelligence minimale

La première stratégie qu'on mettra en œuvre pour tester notre jeu sera l'intélligence aléatoire.Cette stratégie est la même pour tous les personnages. Ainsi à chaque époque, l'ordinateur choisit aléatoirement une commande parmi notre liste de commandes qu'il pourra aplliquer sur nos personnages.

5.1.2 Intélligence basée sur des heuristiques

Nous proposons ensuite un ensemble d'heuristiques pour offrir un comportement meilleur que le hasard, et avec une chance notable de résoudre le problème complet.
Le but des unités d'une armée est d'anéantir les unités ennemies ainsi que leurs batments :

- les unités d'un joueur vont se déplacer pour aller détruire les batiments d'un joueur ennemi
- lorsqu'un batiment d'un joueur est attaqué, alors l'unité le plus proche va aller défendre son camp en attaquant l'ennemi
- si deux unités ennemis se croisent, ils vont se battre jusqu'à ce que l'une des deux meurt.

La plupart des heuristiques proposées sont mis en œuvre en utilisant des cartes de distance vers un ou plusieurs objectifs. Pour calculer ces cartes, nous utilisons l'algorithme de Dijkstra.

5.1.3 Intélligence avancée

Nous proposons une intelligence plus avancée en suivant les méthodes de résolution de problèmes à états finis. Dans cette configuration, un état est un état du jeu à une époque donnée, tel que défini précédemmment. Les arcs entre les sommets du graphe d'état sont les changements d'états, définis dans la partie du moteur de recherhce.Les arcs représentant donc les différentes actions implémentés précedemment.Ainsi passer d'un sommet du graphe d'état à un autre reviens à passer d'une époque à une autre du jeu, fonction de l'ensemble des commandes reçues (clavier, réseau, IA,. . .). L'évaluation/score d'un sommet/état du jeu est déterminé par le nombre d'unités restantes. Un sommet/état du jeu avec plus aucune unités pour un joueur a un score infini, et n'a qu'un seul arc (sommet/état juste avant la mort de la dernière unité). Le meilleur choix de mouvement pour une unité est donc celui du plus court chemin dans le graphe d'état qui mène vers un score nul. Pour trouver ce chemin, nous suivons des méthodes basées sur les arbres de recherche , avec une propriété importante. En effet, nous n'envisageons pas de dupliquer l'état du jeu à chaque sommet du graphe d'état : compte tenu du nombre de nœuds que nous allons traiter, nous aurions rapidement des problèmes de mémoire. Nous n'allons considérer qu'un seul état que nous modifions suivant la direction choisie par la recherche. Si le sommet suivant est à

une époque suivante, i.e. on descend dans l'arbre de recherche, on applique les commandes associées, et notre état gagne une époque. Si le sommet suivant est à une époque précédente, i.e. on remonte dans l'arbre de recherche, on annule les commandes associées, et notre état retrouve sa forme passée.

5.2 Conception logiciel

Le diagramme des classes pour l'intélligence artificielle est présenté en figure 13.

ClassesAI. Les classes filles de la classe AI implante différentes stratégies d'IA, que l'on peut appliquer pour un personnage :

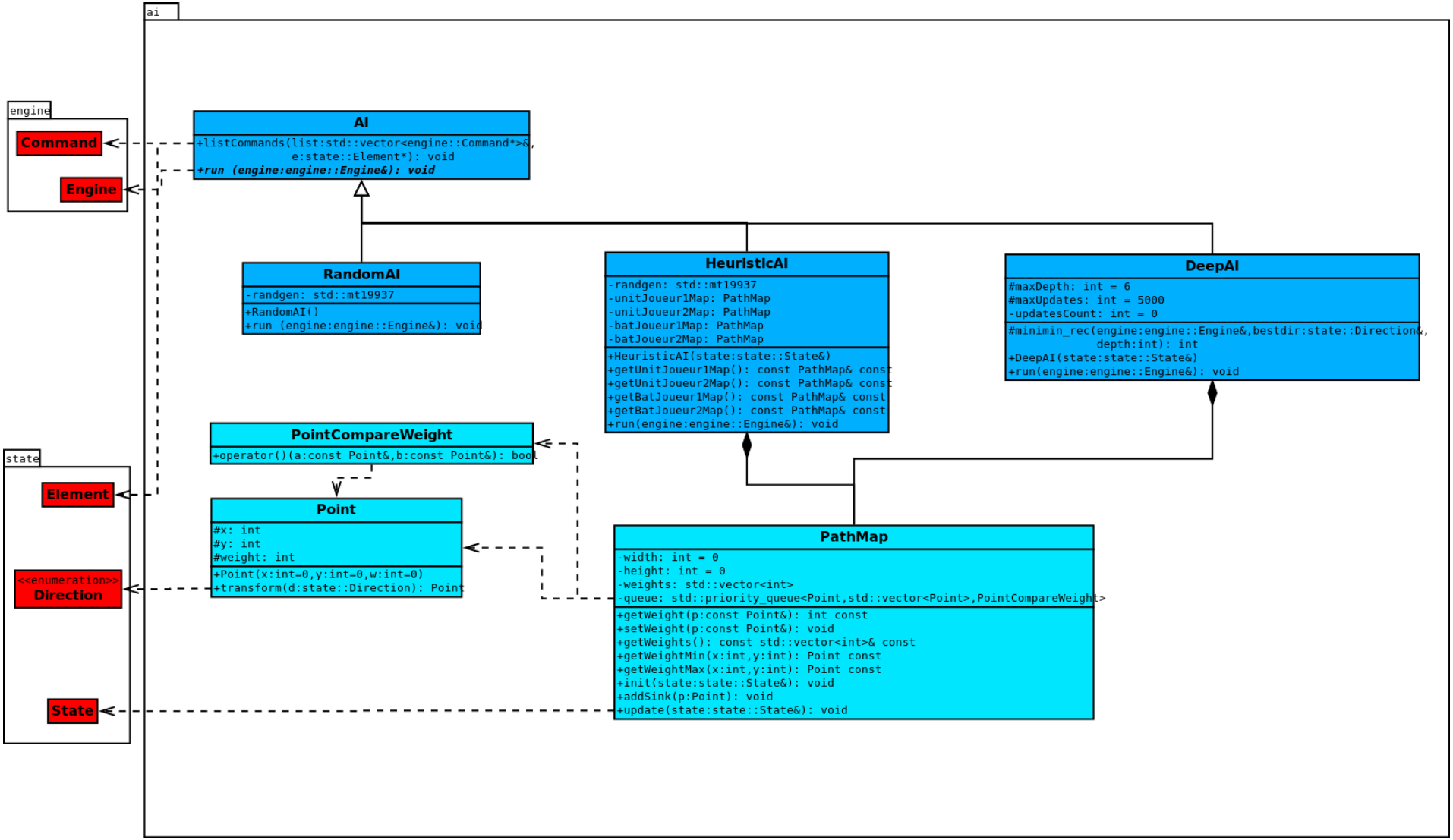
- RandomAI : Intelligence aléatoire
- HeuristicAI : Intelligence heuristique
- DeepAI : Intelligence Avancée

PathMap.
La classe PathMap permet de calculer une carte des distances à un ou plusieurs objectifs. Plus précisément, pour chaque case du niveau ne contenant pas d'océan ni mur, on peut demander un poids qui représente la distance à ces objectifs. Pour s'approcher d'un objectif lorsqu'on est sur une case, il suffit de choisir la case adjacente qui a un plus petit poids. Même si cela n'est pas optimal, on peut également utiliser ces poids pour s'éloigner des objectifs, en choisissant une case avec un poids supérieur. Plusieurs cartes de distantes sont mises à jours régulièrement, en fonction des événements émis :

- unitJoueur1Map : Objectif unités du joueur 1, pour pouvoir s'en approcher.
- unitJoueur2Map : Objectif unités du joueur 2, pour s'en approcher au plus vite.
- batJoueur1Map : Objectif batiments du joueur 1, pour s'approcher des batiments du joueur 1 afin de les détruire.
- batJoueur2Map : Objectif batiments du joueur 2, pour s'en approcher.

DeepAI.
Nous proposons ici une implantation pour une IA basée sur la résolution de problèmes à état fini, tel que défini dans la section précédente. L'algorithme utilisé est une version du minimax vu en cours, mais adapté à 1 joueur contre un autre. Le critère est le nombre de d'unités restantes. Le résultat pour réaliser celà est une sorte de minimaximaximaximax.
Pour les cas où aucune unité n'a été trouvée à proximité, on utilise l'heuristique de l'IA précédente.

Illustration 13: Diagrammes des classes d'intélligence artificielle



6- Modularisation

6.1 organisation des modules

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 (contrainte 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.

6.2 Rassemblement des joueurs

La première étape pour pouvoir jouer en réseau est la création d'une liste de client pour le serveur. Pour ce faire, on utilisera des services CRUD sur la donnée joueur via l'API Web REST.

Requête GET /player/<id>
Pas de données en entrée
Cas joueur <id> existe Statut OK Données sortie : type: "object", properties: { "name": { type:string }, }, required: ["name"]

Cas <id> négatif
Statut OK
Données sortie :
type: "array",
items: {
type: "object",
properties: {
"name": { type:string },
},
},
required: ["name"]

Cas joueur <id> n'existe pas Statut NOT_FOUND Pas de données de sortie

Requête PUT /player
Données en entrée : type: "object", properties: { "name": { type:string }, }, required: ["name"]

Cas il reste une place libre	Statut CREATED Données sortie : type: "object", properties: { "id": { type:number,minimum:0,maximum:4 }, }, required: ["id"]
Cas plus de place libre	Statut OUT_OF_RESOURCES Pas de données de sortie

Requête POST /player/<id>	
Données en entrée : type: "object", properties: { "name": { type:string }, }, required: ["name"]	
Cas joueur <id> existe	Statut NO_CONTENT Pas de données de sortie
Cas joueur <id> n'existe pas	Statut NOT_FOUND Pas de données de sortie

Requête DELETE /player/<id>	
Pas de données en entrée	
Cas joueur <id> existe	Statut NO_CONTENT Pas de données de sortie
Cas joueur <id> n'existe pas	Statut NOT_FOUND Pas de données de sortie

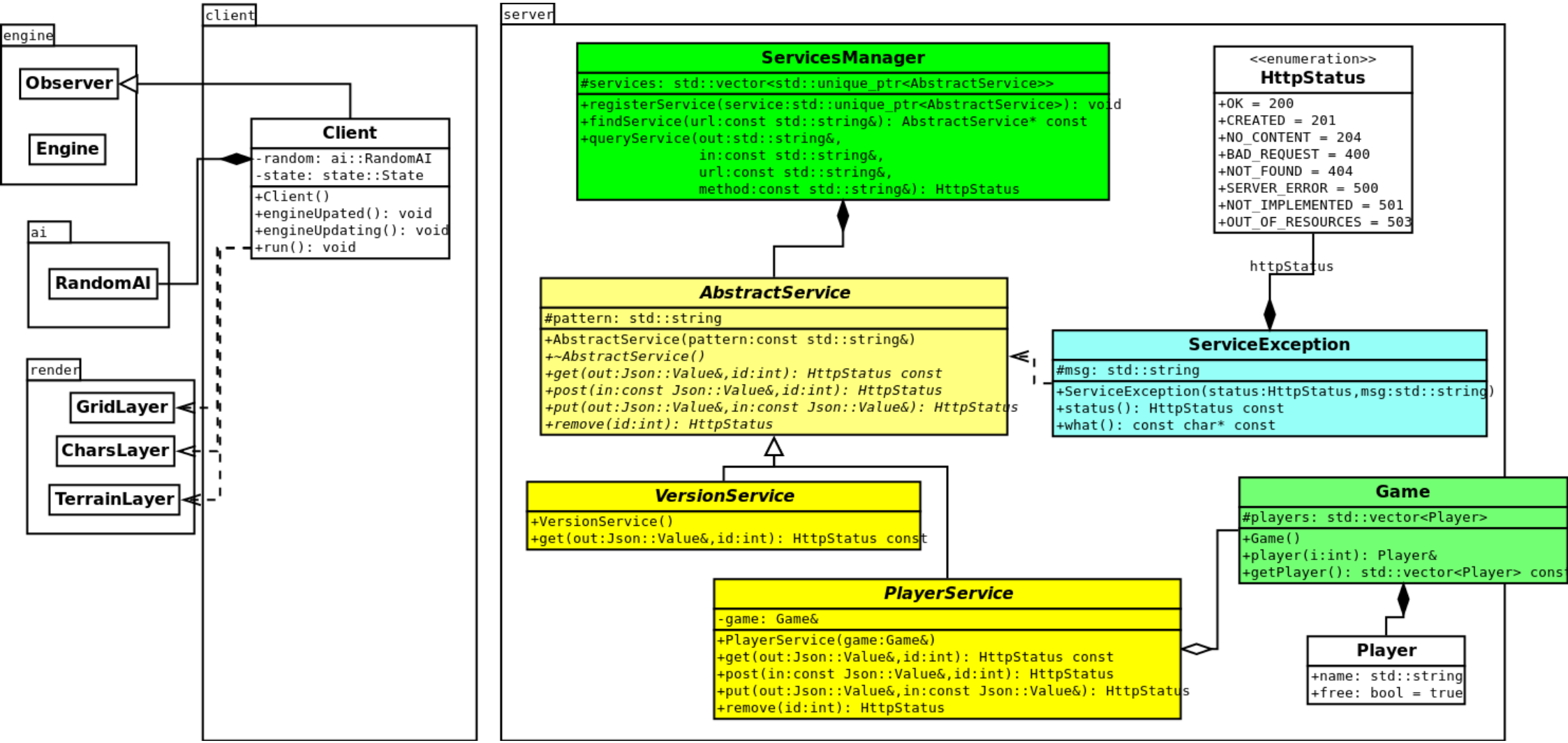
6.2 conception logiciel

Le diagramme des classes pour l'intelligence artificielle est présenté ci dessous.
Client. La classe Client contient toutes les informations pour faire fonctionner le jeu : Moteur de jeu (avec état intégré), intelligences artificielles, et rendu. Cette classe est un observateur du moteur de jeu :

- Lorsque le moteur est sur le point d'exécuter ses commandes (méthode engineUpdating()), il appelle les IA pour ajouter les commandes des robots ;
- Lorsque le moteur a terminé d'appliquer les commandes (méthode engineUpdated()), il vide le cache des événements émis vers le moteur de rendu.

Game. La classe game et les éléments associés représente les éléments d'une partie.On trouve ainsi une liste de joueur présent ou non dans une partie (attribut Player:free).

Service. Les services REST sont implantés via les classes filles de AbstractService, et gérés par la
classe ServiceManager :
— VersionService : le traditionnel service qui renvoie la version actuelle de l'API. Indispensable dans
toute API pour prévenir les conflits de version.
— PlayerService : fournit les services CRUD pour la ressource joueur. Permet d'ajouter, modifier,
consulter et supprimer des joueurs.


client
engine
Observer
Engine
Client
-random: ai::RandomAI
-state: state::State
+Client()
+engineUpated(): void
+engineUpdating(): void
+run(): void
ai
RandomAI
render
GridLayer
CharsLayer
TerrainLayer
server
ServicesManager
#services: std::vector<std::unique_ptr<AbstractService>>
+registerService(service:std::unique_ptr<AbstractService>): void
+findService(url:const std::string&): AbstractService* const
+queryService(out:std::string&,
in:const std::string&,
url:const std::string&,
method:const std::string&): HttpStatus
<<enumeration>>
HttpStatus
+OK = 200
+CREATED = 201
+NO_CONTENT = 204
+BAD_REQUEST = 400
+NOT_FOUND = 404
+SERVER_ERROR = 500
+NOT_IMPLEMENTED = 501
+OUT_OF_RESOURCES = 503
httpStatus
AbstractService
#pattern: std::string
+AbstractService(pattern:const std::string&)
+~AbstractService()
+get(out:Json::Value&,id:int): HttpStatus const
+post(in:const Json::Value&,id:int): HttpStatus
+put(out:Json::Value&,in:const Json::Value&): HttpStatus
+remove(id:int): HttpStatus
ServiceException
#msg: std::string
+ServiceException(status:HttpStatus,msg:std::string)
+status(): HttpStatus const
+what(): const char* const
VersionService
+VersionService()
+get(out:Json::Value&,id:int): HttpStatus const
Game
#players: std::vector<Player>
+Game()
+player(i:int): Player&
+getPlayer(): std::vector<Player> const
PlayerService
-game: Game&
+PlayerService(game:Game&)
+get(out:Json::Value&,id:int): HttpStatus const
+post(in:const Json::Value&,id:int): HttpStatus
+put(out:Json::Value&,in:const Json::Value&): HttpStatus
+remove(id:int): HttpStatus
Player
+name: std::string
+free: bool = true