



Projet Logiciel Transversal

Luigi CAPO-CHICHI – Yassert BOINALI

Table des matières

1	Objectif.....	3
1.1	Présentation générale.....	3
1.2	Règles du jeu.....	3
1.3	Ressources.....	3
2	Description et conception des états.....	6
2.1	Description des états.....	6
2.1.1	Etat des éléments fixes.....	7
2.1.2	Etat des éléments mobiles.....	7
2.2	Conception logiciel.....	7
3	Rendu : Stratégie et Conception.....	9
3.1	Stratégie de rendu d'un état.....	9
3.2	Conception logiciel.....	9
4	Règles de changement d'états et moteur de jeu.....	11
4.1	Changements extérieurs.....	11
4.2	Changements autonomes.....	11
4.3	Conception logiciel.....	11

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

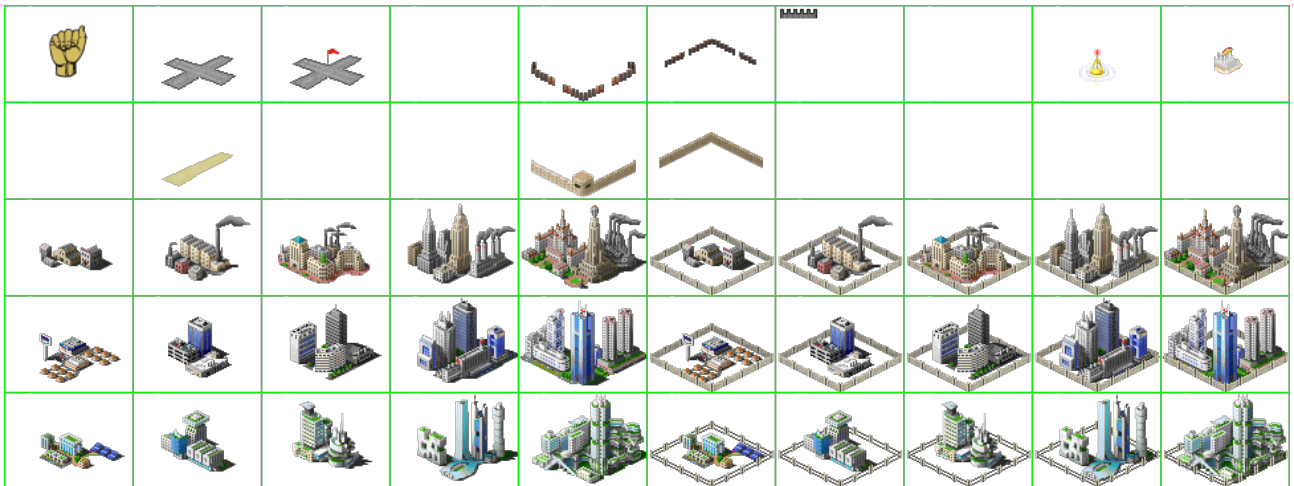


Figure 3 : Textures des bâtiments de l'ère moderne

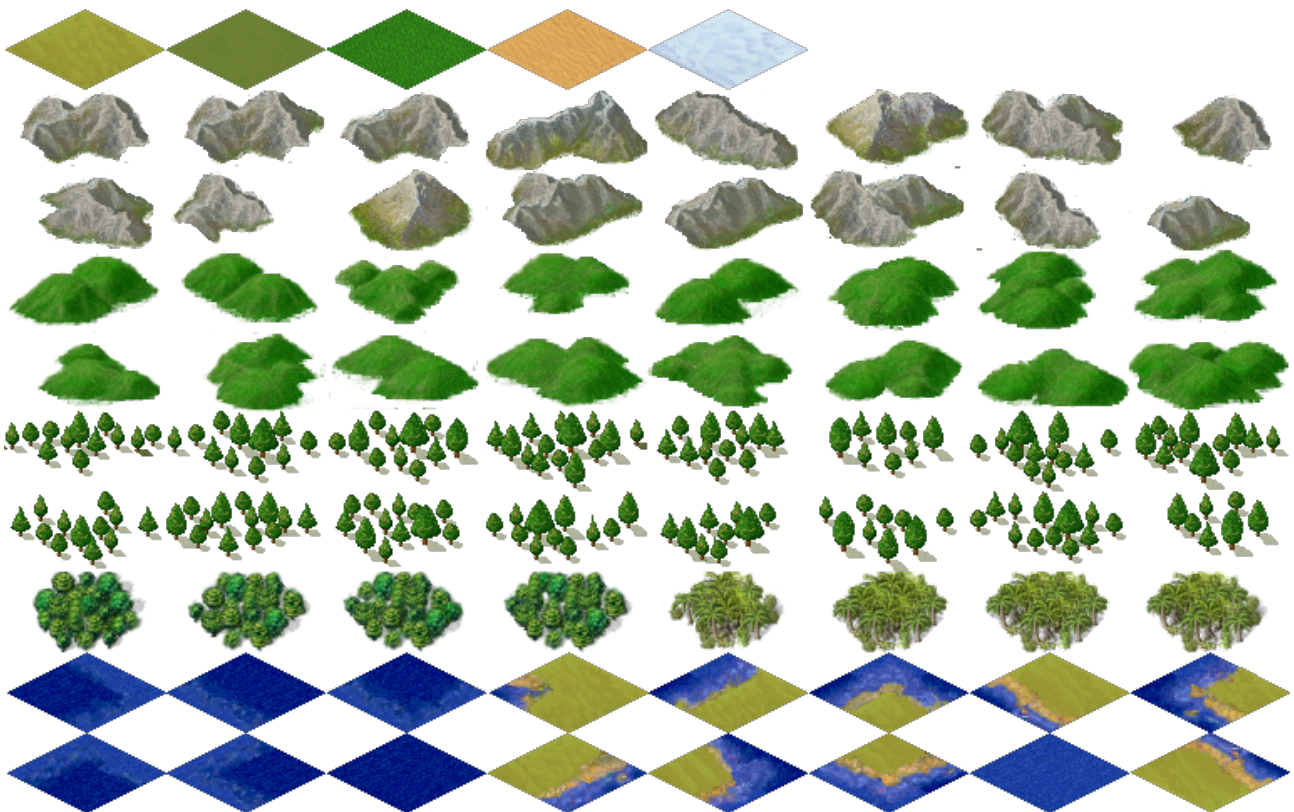


Figure 5 : Textures pour le terrain

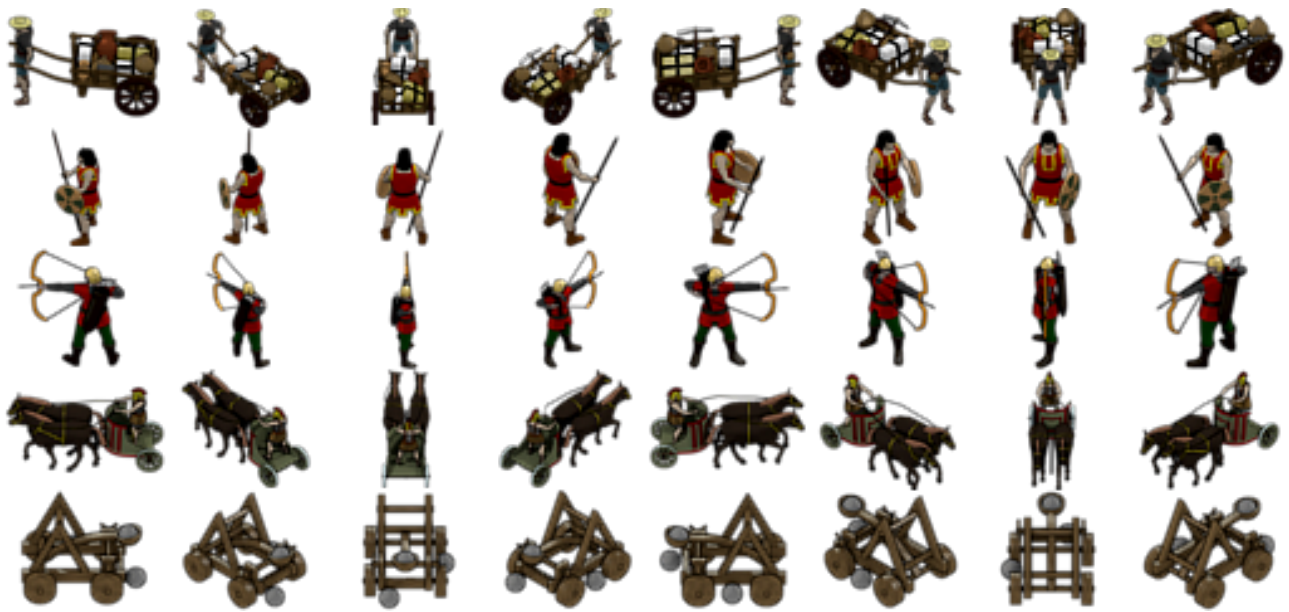


Figure 7 : Textures pour les unités



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 l'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.

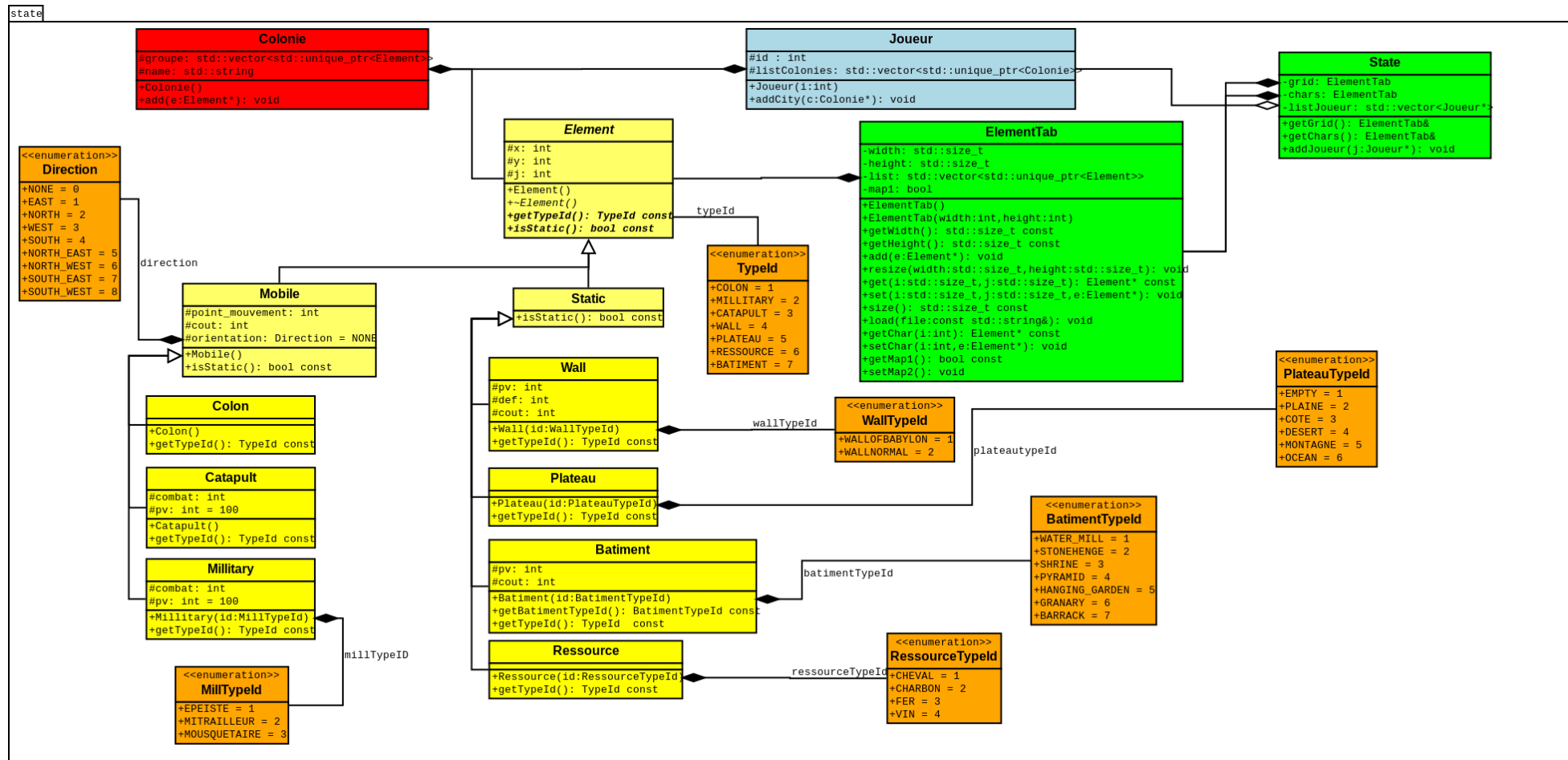
2.2 Conception logiciel

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.

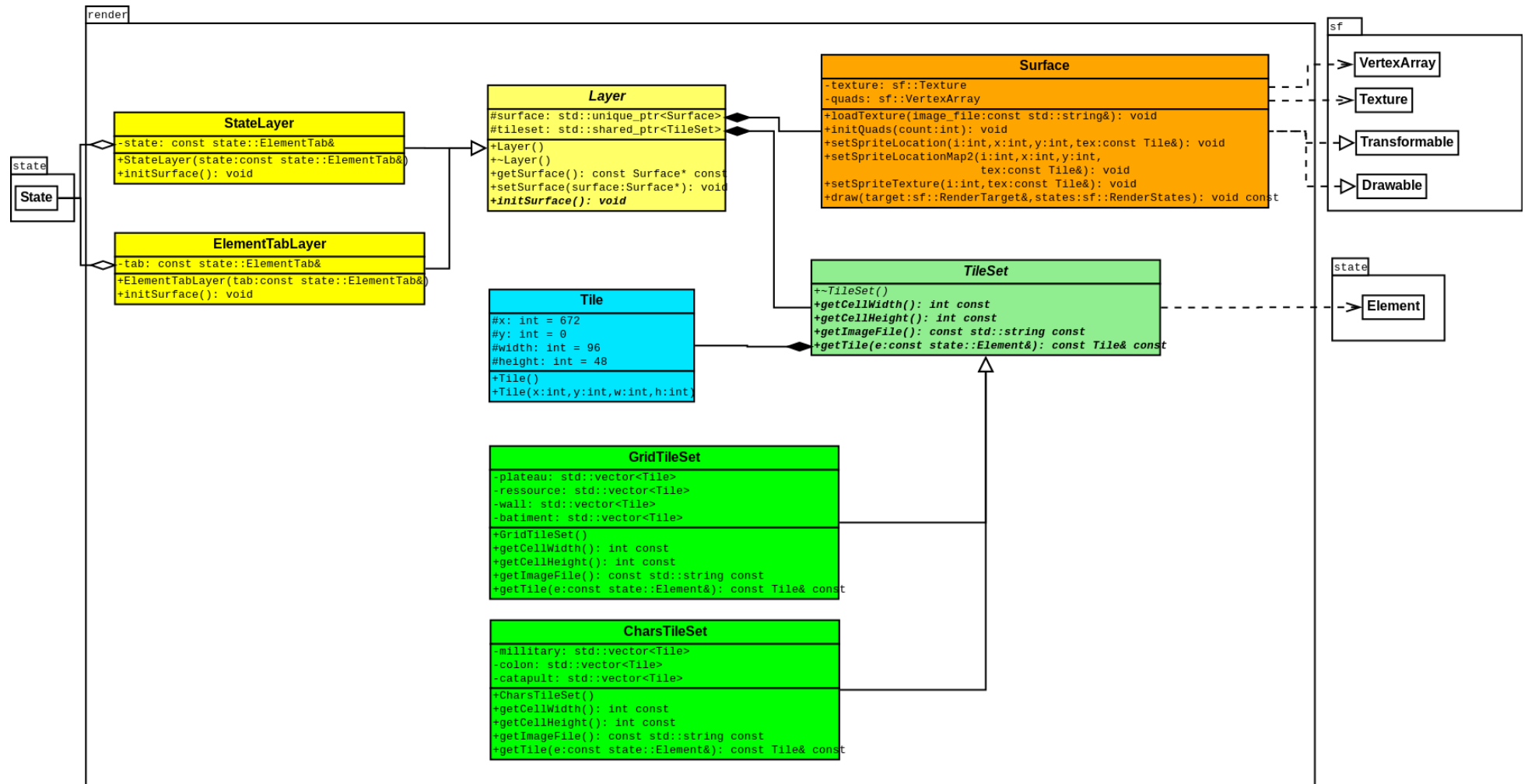
Layer: Cette classe est le moteur du rendu qu'on veut réaliser. Le principal but des objets de cette classe est de former des éléments basiques qu'on pourra transmettre à notre carte graphique par l'intermédiaire d'une classe surface. Les classes filles de cette classe permettent de définir les différents 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 une 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, la 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 possibilité 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 territoire.

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,

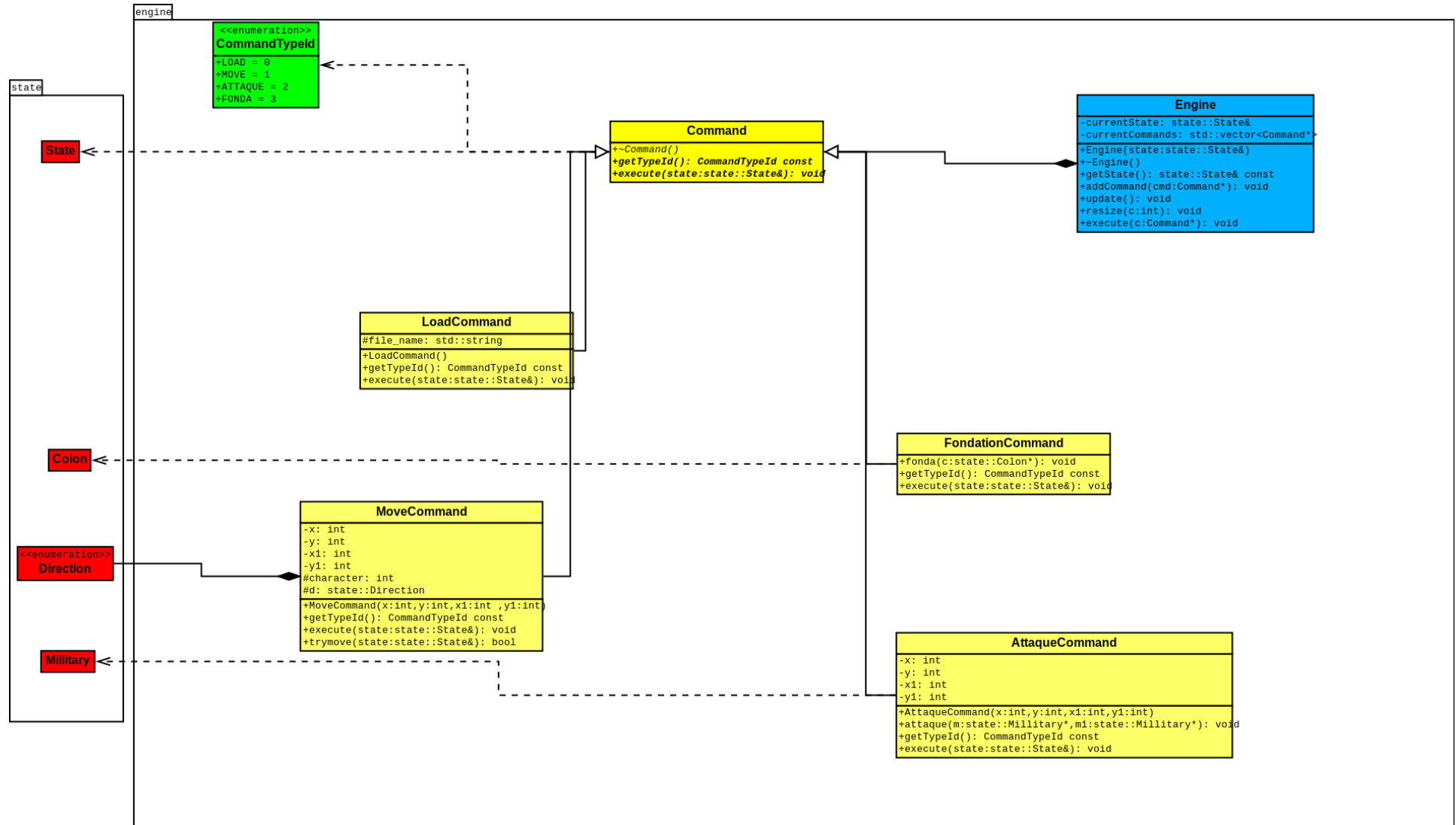
quelque soit sa source(automatique, clavier, réseau,...).Notons bien que ces classes ne gèrent absolument pas l'origine des commandes. Ce sont d'autres éléments en dehors du moteur de jeu qui fabriquerons les instances de ces classes.

A ces classes, on a défini un type de commande avec `CommandType` pour identifier précisément la classe d'une instance.

- `LoadCommand`. Charge un niveau depuis un fichier
- `OrientationCommand`. Permet de réaliser le déplacement d'un personnage selon une direction.
- `AttaqueCommand`. Réalise une attaque sur des unités ennemis.
- `FondationCommand`. Fonde une nouvelle colonie.

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.

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'intelligence 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 appliquer sur nos personnages.

5.2 Conception logiciel

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

Classe AI. Cette classe représente la classe mère de toutes les différentes stratégies d'intelligence artificielle qu'on utilisera. Sa méthode `listCommandes` permet de lister les différentes possibilités de commandes qu'on peut réaliser sur un élément quelconque.

- **RandomAI.** Classe fille de AI. Elle nous permet ainsi de réaliser une intelligence aléatoire à l'aide de notre attribut `randgen`.

Illustration 13: Diagrammes des classes d'intelligence artificielle

