ENSEA
Ecole Nationale
Supérieure
de l'Electronique
et de ses Applications

Projet Logiciel Transversal

Alexis LOISON – Eric TAN

Option Informatique et Systèmes

Capture d'écran du jeu

SOMMAIRE

1 Objectif

1.1 Présentation générale

Présenter ici une description générale du projet. On peut s'appuyer sur des schémas ou croquis pour illustrer cette présentation. Éventuellement, proposer des projets existants et/ou captures d'écrans permettant de rapidement comprendre la nature du projet.

Le jeu proposé est basé sur « Advance Wars », un jeu de guerre militaire en 2 dimensions avec la possibilité d'avoir plusieurs joueurs qui puissent y jouer. Nous travaillerons avec 2 joueurs ou bien 1 joueur contre 1 IA.

Illustration 1 :Capture d'écran du jeu

1.2 Règles du jeu

Les joueurs doivent capturer toutes les infrastructures ennemies ou uniquement le quartier général.
Si le joueur perd toutes ses infrastructures ou son quartier général il perd la partie.
Chaque joueur peut créer des unités à partir de bâtiments de production qui seront construits grâce à des ressources récoltées/perçues en jeu.
Chaque unité a des points de vie, des avantages et des faiblesses face à d'autres unités. Elle pourra des points de vie en retournant à l'usine(terrestres) ou à l'aéroport(aériens).

1.3 Conception Logiciel

Package state : il s'agit du package contenant toutes les informations pour représenter un état de notre jeu.
Package render : celui-ci permet d'afficher un état de notre jeu en version graphique.
Package engine : son but est de permettre à une personne ou une IA de jouer au jeu. Ainsi ce package permet de modifier un état de jeu en fonction d'événements comme une action au clavier ou à la souris par exemple.

On peut ainsi représenter le diagramme des packages suivant pour notre projet :

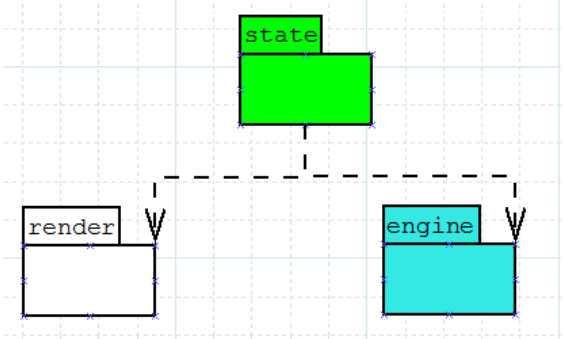


Illustration 2 : Diagramme des packages

2 Description et conception des états

2.1 Description des états

L'ensemble des états fixes et mobiles sont composés de :

- Coordonnées (x,y) dans la grille de chaque entité
- Identifiant du type d'élément : la nature de l'élément et sa couleur

2.1.1 Etat éléments fixes

Notre carte sera composée de « cases ». La taille est définie par défaut. Chaque case ne pourra accueillir qu'un maximum de 10 unités du même élément à la fois.

- **Cases « Obstacles »**. Celles-ci sont infranchissables par certains éléments mobiles ou franchissables avec pénalités.
  - ➔ « Rivière » : franchissable uniquement par les éléments mobiles « aériens »
  - ➔ « Forêt/Bois » : franchissable par les éléments mobiles « aériens » et les éléments « infanteries » avec pénalité

- **Cases « Franchissables ».** Toutes les unités mobiles peuvent franchir ces éléments.
  - ➔ « Vide » : case ayant pour texture esthétique : une route ou de l'herbe
  - ➔ « Mine » : case rapportant des ressources de métal au détenteur
  - ➔ « Immeuble » : procure une diminution des dégâts reçus pour les infanteries
  - ➔ « Quartier Général » : permet aux éléments mobiles de récupérer des points de vie. La capture de cette case par un élément mobile ennemie procure la défaite de son ex-détenteur.
  - ➔ « Usine » : permet de créer des éléments « terrestres » mobiles ou leur permet récupérer des points de vie
  - ➔ « Aéroport » : permet de créer des éléments « aériens » ou leur permet récupérer des points de vie

2.1.2 Etat éléments mobiles

Les éléments mobiles possèdent :

- Un **déplacement** : vers la gauche, vers la droite, vers le haut, vers le bas
- Une **vitesse de déplacement** définissant le nombre de cases que peut parcourir un élément mobile en 1 tour de jeu
- Une **position** [x,y] définit sur le cadre
- Des **dégâts**
- Des **points** de vie : entier allant de 0 à 10
- Une **couleur** : 1 pour le rouge, 2 pour le bleu et 3 pour le gris
- Un **rang** caractérisé par un chiffre : 1 pour une promotion et 0 pour une non-promotion
- Ces éléments peuvent être « aérien » ou « terrestre ».

2.1.3 Etat général

Les propriétés suivantes sont ajoutées :

Epoque : « l'horloge » du jeu, l'heure depuis le commencement de la partie

Fréquence : nombre de fois par minute que l'état du jeu est mis à jour

2.2 Conception logiciel

Nous utilisons le logiciel Dia afin de créer notre diagramme des classes. Décrivons en deux groupes nos différentes classes.

Classes Elément : composé de *Element*, *MobileElement* et *StaticElement*, elles permettent de décrire les éléments qui composent notre jeu de type mobile ou statique. *Element* englobe les deux autres classes filles citées.

On pourra par la suite choisir si un élément est aérien ou terrestre lorsqu'il appartient hérite de la classe *MobileElement*. Dans un autre cas, si celui-ci est statique, on passera par les deux classes filles *Wall* et *Space*.

Conteneurs d'élément : les trois classes *State*, *ElementList* et *ElementGrid* permettent de contenir tous nos éléments. *ElementList* sert à stocker sous forme de liste les éléments de notre jeu. Tandis que *ElementGrid* sert à stocker notre plateau de jeu sous la forme d'une grille. Bien sûr, les deux classes contiennent également les fonctions permettant de modifier une liste d'élément pour la première et une grille pour la seconde. *State* est la classe état qui permet de rendre compte de l'instant auquel on se trouve. Ainsi, il est composé avec les deux autres classes car on a besoin d'une grille et d'une liste d'éléments.

2.3 Conception logiciel : extension pour le rendu

Observateurs : Afin de pouvoir faire des changements d'états, nous avons besoin d'observateurs qui mettront à jour les tuiles de notre jeu. Nous utilisons donc le pattern Observer dans notre jeu. *StateObserver* sert à avertir lorsqu'un état a changé. Pour connaître la nature des changements dans l'état il faut alors passer par *StateEvent* (grâce à des opérateurs de comparaison et à une liste prédéfinie de changements possibles).

2.4 Conception logiciel : extension pour le moteur de jeu

2.5 Ressources

On codera notre carte en fichier texte afin de pouvoir traduire plus tard celle-ci en carte de jeu.

```
FGGGFGGGGGGGGGGGGGGGGGGGGGGG
GGGGFGGGGGGGGGGGGGGGGGGGGGGG
GRHMRGGGGRRRRRRRRRFWWGGGRSSSSSGGG
ARBBFGGGGRMSSSWSWRWWWWGGRSSSSSSGG
GRRRRRRRRRWSWWWWBRGGRRRRRRSSSSSSG
GGGGGGWRWWWWWRRRRRRRRRBWWWRSGSSSGG
GGGGGGGRWWWWWRMWGGGRSWWWWRSGSMSGG
GGGGGGWRWWWWBRWWGGSSSWWWWRAGGGGGG
GGGGGGARWWWWSSSGGWWRBWWWWRWGGGGGG
GGSMSGSRWWWWSRGGGWMRWWWWWRGGGGGGG
GGSSSGSRWWWBRRRRRRRRRWWWWWRWGGGGGG
GSSSSSSRRRRRRGGRBWWWWSWRRRRRRRRRR
GGSSSSSSRGGWWWWRWSWSSSMRGGGGFBBRA
GGGSSSSSRGGGWWFRRRRRRRRRGGGGRMHRG
```

GGGGGSGGRRRRRRRRGSGGGGGGGGGGRRRRG
GGGGGGGGGGGGGGGGGGGGGGGGGGGGFGGGF

Pour y voir plus clairement

R/Road/Gris
G/Grass/Vert
M/Mine/Jaune
B/Building/Gris clair
H/Headquarter/ Rose
F/Factory/Orange
A/Airport/Violet
S/Sea/Bleu/Marron
W/Woods

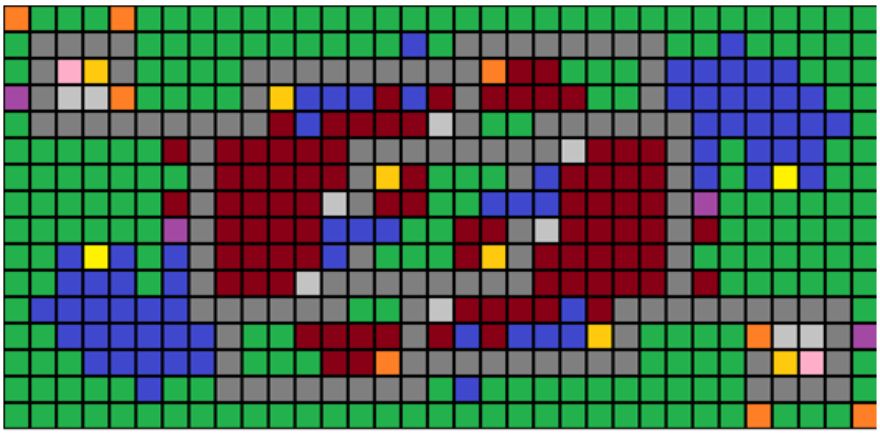
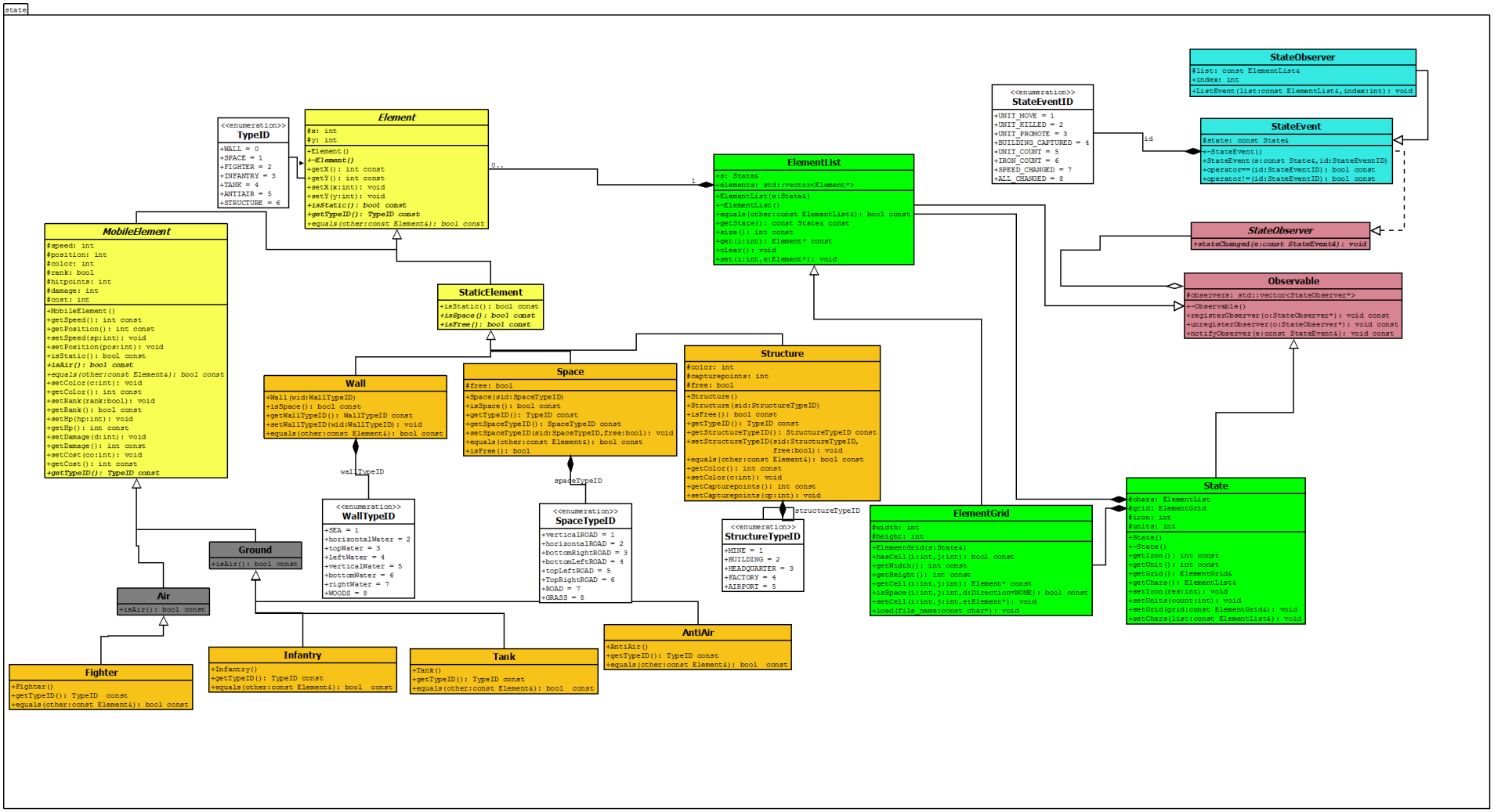


Illustration 3 : carte de jeu

Illustration 4 : Diagramme des classes d'état



3 Rendu : Stratégie et Conception

3.1 Stratégie de rendu d'un état

Pour faire le rendu d'un état, nous utilisons SFML : Simple and Fast Multimedia Library. Celle-ci va nous permettre d'alléger le traitement des tuiles faites par la carte graphique de l'ordinateur.

Le principe utilise un système avec des vertex contenant plusieurs informations : une position, une couleur et une paire de coordonnées pour un jeu en 2 dimensions. Lorsque l'on groupe ces vertex ensemble (primitives), on obtient un tableau qui nous donne une forme.

Dans notre jeu, on affiche d'abord toute la carte avec les textures correspondantes aux bons tableaux de vertex. Ensuite, on peut par la suite changer les attributs des vertex qui nous intéressent pour modifier le rendu (par exemple signaler un changement d'unité/bâtiment d'une case de notre terrain).

Il faut également noter que nous aurons deux « fenêtres », une qui affiche le terrain et une autre qui montre les informations du joueur (ressources, couleur) ainsi que celle d'une unité/bâtiment qu'il aura sélectionnée. Ainsi, on peut définir une scène qui sera découpée en plusieurs plans : un pour les unités mobiles, un pour les cases franchissables et obstacles et un dernier servant à afficher les infos utiles au joueur. Chaque plan possède donc les caractéristiques suivantes : la position de ses tuiles ainsi que leurs textures.

3.2 Conception logiciel

Dans le cadre de notre jeu, nous allons utiliser le diagramme des classes ci-après.

Pour afficher notre rendu, nous devons utiliser une classe *Tile* pour générer des tuiles pour ensuite les afficher. Cette classe peut être séparé aura pour classe fille : *StaticTile* la liste des tuiles de notre jeu.

Pour chaque animation plusieurs tiles seront affectées, toutes nos tiles auront les mêmes animations. Pour gérer l'ensemble nous utiliserons une classe *Layer* qui fournira les informations bas-niveau nécessaires pour la carte graphique, ces informations seront dans notre cas traitées par le biais de la librairie Sfml implanté dans *Surface*. Le traitement de ses informations pour l'affichage est fait par la classe *Surface* qui sera en charge du chargement et du placement des textures.

Tout changement de l'état entraînera une réaction de *StateObserver* qui engendrera une actualisation du rendu.

Le rendu sera composé de trois plans : une qui s'occupera de l'affichage des éléments immobiles, une pour les éléments mobiles et une autre pour l'interface.

3.3 Ressources

Voici les ressources utilisées pour notre jeu :

3.4 Exemple de rendu

Exemple de rendu du niveau de jeu :

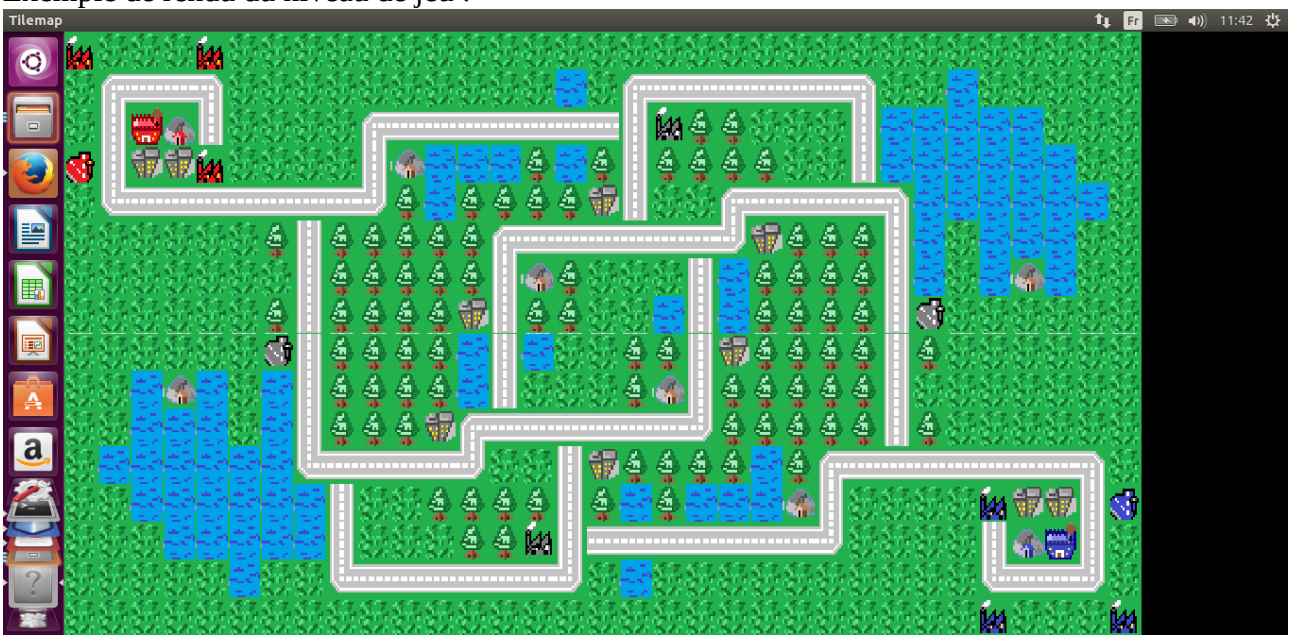
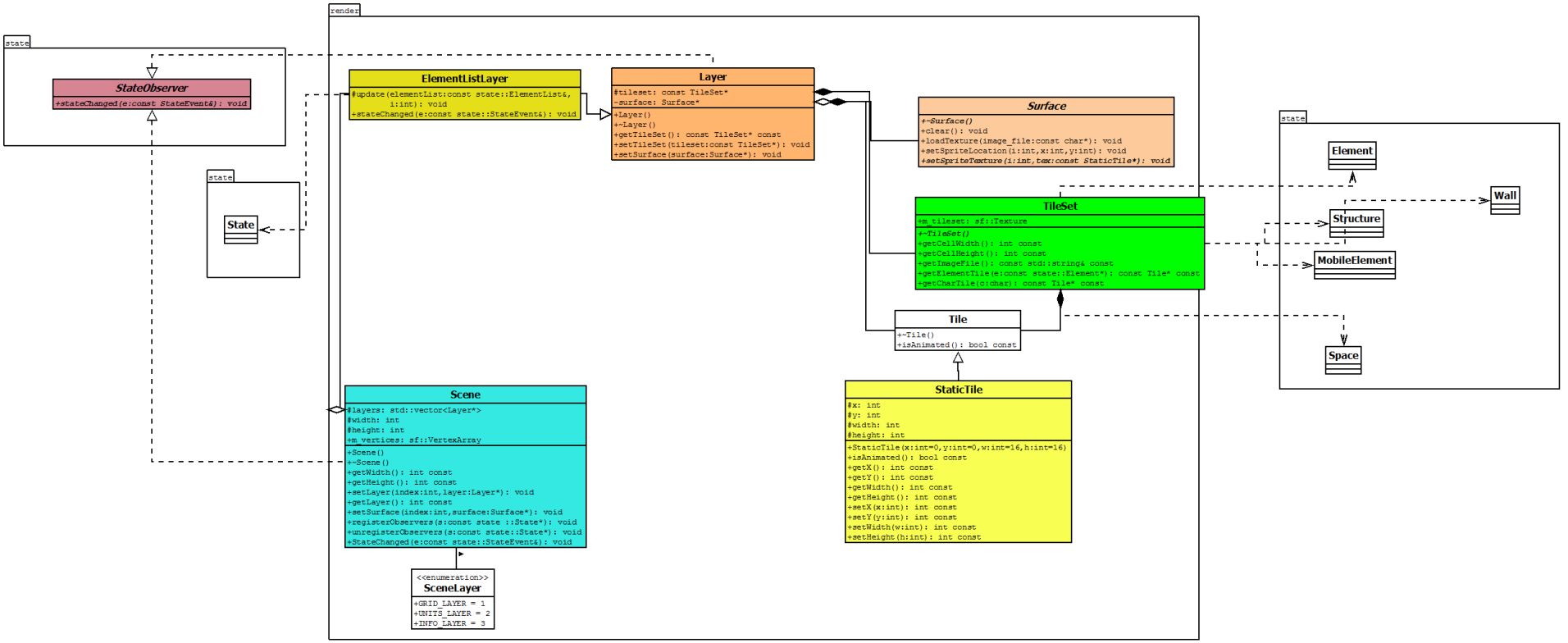


Illustration 5 : Diagramme des classes du rendu



4 Règles de changement d'états et moteur de jeu

Notre jeu a besoin d'un moteur de jeu afin de passer d'un état à un autre.

4.1 Changements extérieurs

Les changements extérieurs sont réalisés par des commandes extérieurs (appuis sur des touches par exemple).

Commandes principales : « LOAD » : on définit un état initial comme début de jeu

4.2 Changements autonomes

On parle de changement autonome lorsqu'un état se met à jour après l'effet de changements extérieurs. On a :

1. Si une unité se déplace sur une case sur laquelle :
    - Une unité ennemie est présente, celle-ci ira à une case adjacente la plus proche de l'ennemi (depuis sa position) et l'attaquera
    - SPACE est présent : celui-ci se placera sur la case si cela est possible
    - Un bâtiment allié sans unité est présent : celui-ci se place sur la case du bâtiment
    - Un bâtiment ennemi sans unité est présent : celui-ci se place sur la case du bâtiment et le capture
    - Un bâtiment ennemi avec unité est présent : celui-ci se place à une case adjacente la plus proche de lui et attaque

2. Si les points de vie d'une unité tombent à 0, celui-ci disparait de l'écran.

3. Compteur d'unité : si une unité meurt, le compteur du joueur le possédant est décrémenté de 1. A l'inverse, celui-ci s'incrémente de 1 lorsqu'une unité est créée.

4. Compteur de ressources : à chaque début de leur tour, le joueur reçoit une quantité de ressources en fonction du nombre de mine(s) qu'il possède.

5. Lorsque le *Quartier Général* est capturé par une unité ennemi, le bâtiment change alors de couleur et le joueur ennemi remporte la partie.

6. La vitesse de déplacement de *Infantry* est diminuée lorsqu'il se trouve dans une case WOODS.

7. Lorsqu'une unité est créée, elle doit se trouver sur le bâtiment concerné. De plus, il faut que la ressource « iron » soit mise à jour.

4.3 Conception Logiciel

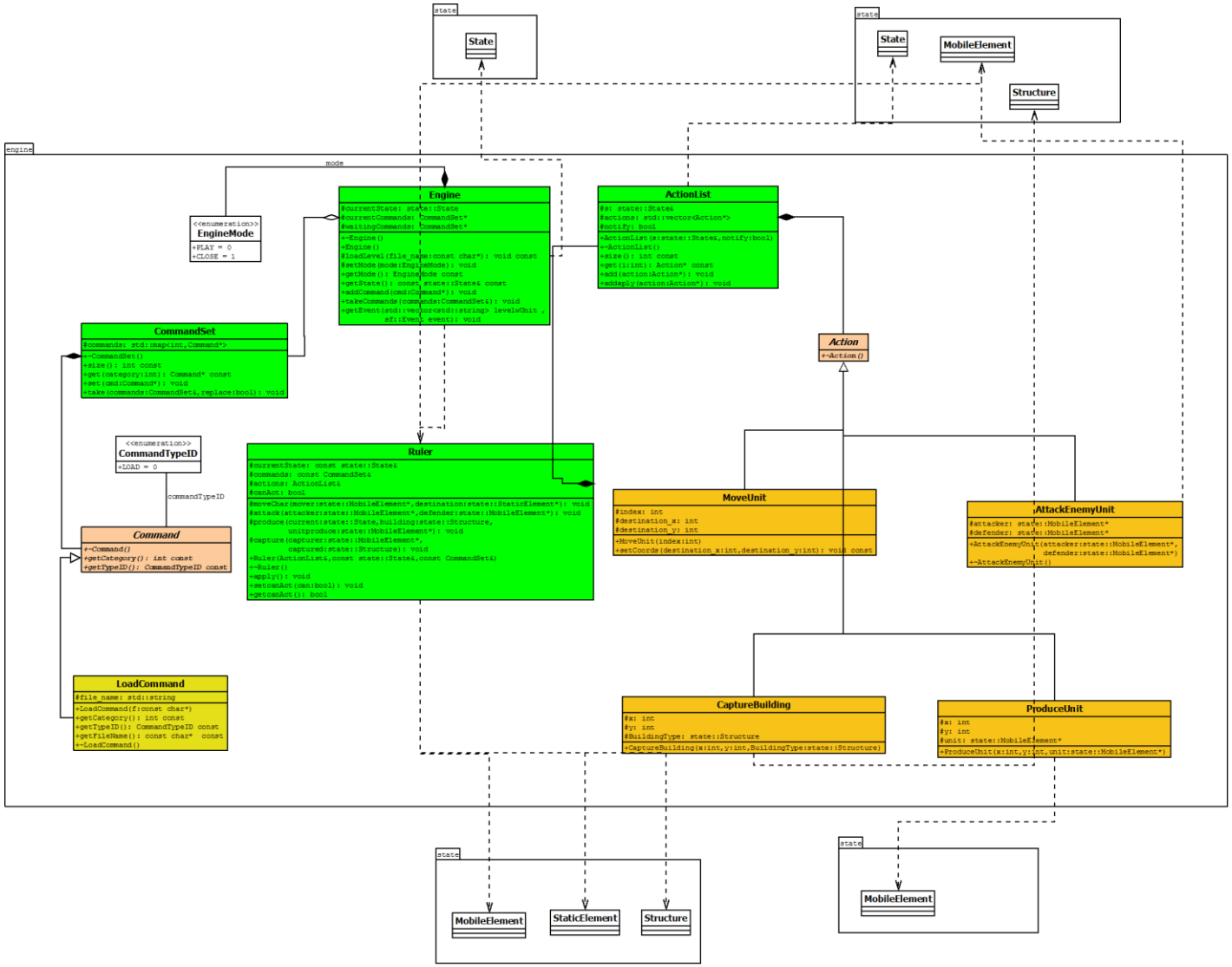
Le diagramme des classes que nous allons utiliser est présenté ci-après.

L’ensemble du moteur de jeu repose sur la classe Command principalement, c’est elle qui va permettre de gérer les commandes extérieures qui influent sur l’état du jeu.

Classe Action : la classe au joueur d’interagir avec l’état de son jeu concernant les actions qu’il pourra effectuer en jeu.

On ajoute également l’observer, dès qu’un événement est présent (sf ::Event), un ordre est envoyé sur le rendu pour que celui-ci s’actualise.

Illustration 6 : Diagramme des classes du moteur



5 Intelligence artificielle

5.1 Stratégies

Afin de créer notre intelligence artificielle, nous allons procéder par plusieurs niveaux d'intelligence. D'un niveau simple et basique à une intelligence plus robuste et développée. Les différences entre celles-ci relèvent donc des réactions de l'IA en fonction de l'environnement.
Son fonctionnement sera également illustré par un diagramme des classes (Illustration 7).

5.1.1 Intelligence minimale

Dans un premier temps, l'IA implémentée sera simple et ne tiendra compte que de très peu de paramètres de l'environnement.

- ➔ L'IA avance jusqu'à rencontrer un obstacle puis se retourne et fait le chemin inverse à l'aide d'une unité.

Illustration 7 : Diagramme des classes de l'IA