Projet Logiciel Transversal

Anand Candassamy & Paul Estano

Table des matières

1 Présentation Générale

1.1 Archétype

Notre jeu s'inspirera principalement du jeu *Pokemon Donjon Mystère*. En effet, nous avons prévu de conserver le mécanisme des combats et de donjon de ce jeu.
Dans notre logiciel l'utilisateur incarnera un pokemon dans les salles d'un donjon qui contiennent chacune des pokemons qui peuvent l'"agresser". Le nombre de pokemons dans une salle évolue en fonction de l'avancement du joueur dans le jeu.
Pour simplifier le jeu nous abandonnons également d'évolution des pokemons.

1.2 Règles du jeu

Le donjon contient un nombre fini de salles et le joueur gagne lorsqu'il sort de la dernière salle du donjon. Un donjon a 10 salles, le pattern de chaque étage n'est pas tiré aléatoirement. Le joueur a la possibilité d'accéder à des étages bonus, dîtes arène de combat où il pourra se battre contre un autre joueur en ligne.
Le joueur est provoqué en duel automatiquement par les pokemons qui sont autour de lui. Il faut qu'il tue l'IA ou le joueur adverse pour activer la case de l'esaclier qui leur permet de passer à l'étage suivant.
Les combats fonctionnent en tour par tour. A chaque tour, l'utilisateur peut effectuer une action :

- attaquer avec une compétence
- soigner le pokemon
- se déplacer d'une case

1.3 Ressources

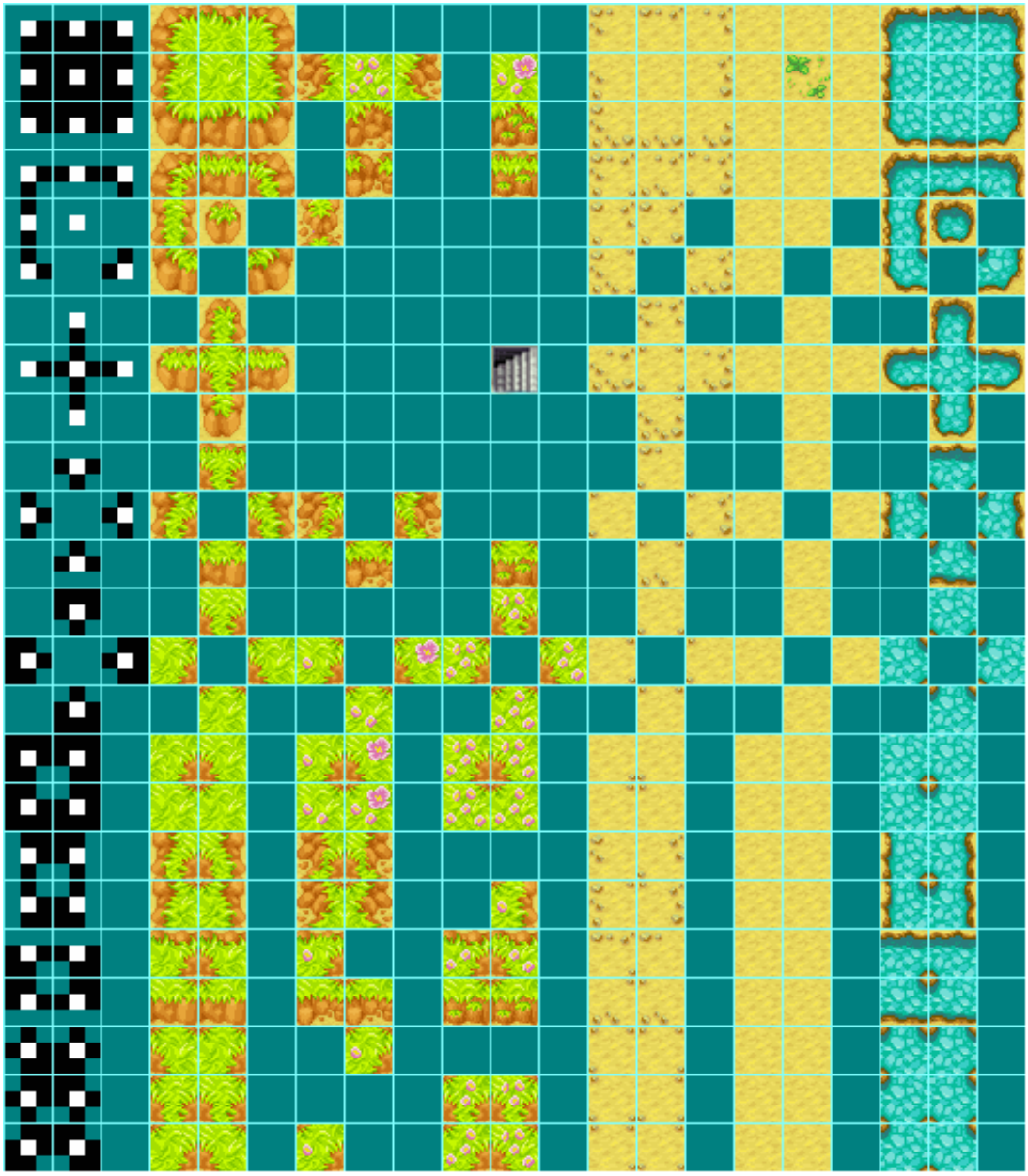

FIGURE 1 – Tileset utilisé pour construire un monde

FIGURE 2 – Tileset utilisé pour les pokémons

2 Description et conception des états

2.1 Description des états

Un état du jeu est formé d'une carte qui est statique au cours du temps et d'une liste de joueurs dont l'état varie au cours du jeu.

2.1.1 État de la carte

La carte est un élément fixe de l'état du jeu. Il se décompose en plusieurs cases.
Les cases se distinguent en 3 parties :
— les murs, qui ne sont pas franchissables par les éléments mobiles.
— les cases "conteneurs", elles peuvent être vide ou occupé par un seul élément mobile.
— les cases escaliers, il en existe un seul par étage, qui permet d'aller au niveau suivant du labyrinthe.

2.1.2 État des joueurs

Un pokemon est soit contrôlé par un joueur soit contrôlé par une Intelligence Artificielle (I.A), il possède une position, un nombre de points de vie à l'état t, un nom et un ensemble de statistiques qui lui sont attribués en début de partie. Le pokemon est considéré comme étant "mort" si son nombre de points de vie atteint 0.

2.1.3 État général

A l'ensemble des éléments statiques et mobiles, nous rajoutons les propriétés suivantes : — Époque : représente « l'heure » correspondant à l'état, ie c'est le nombre de « tour » passés globale depuis le depuis de la partie.

Ces statistiques correspondent aux attaques qu'il peut utiliser, à son nombre de point de vie en début de partie et son type (Salameche, Bulbizarre, Carapuce).

2.2 Conception Logiciel

Le package état peut se diviser en trois sous-partie :
— Une partie gérant les personnages, en bleu
— Une partie gérant l'environnement, en rouge
— Une classe représentant l'état global du jeu, en vert

La classe *Player* contient l'ensemble des éléments permettant de caractériser l'état d'un joueur. Chaque joueur est lié à un pokemon par une relation de composition : un pokemon ne peut pas exister sans joueur. Dans le cas où le pokemon est contrôlé par l'IA, l'IA est considérée comme un joueur et contrôle donc son pokemon.

Chaque pokemon possède une relation d'agrégation avec des statistiques qui lui sont attribués en début de partie.

Pour notre implémentation du jeu, nous avons prévu d'utiliser que les trois pokémon suivant :
— Carapuce
— Salamèche
— Bulbizarre

Le constructeur respectif de chacun de ces trois pokemons héritent de la classe Pokemon. Nous avons mis en place un identifiant pour chaque pokemon, ce qui permet d'avoir plusieurs pokemon construit à partir de la même classe.

La partie environnement est calqué sur le modèle proposé par les fichiers exportés par le logiciel *Tiled Map Editor* au format JSON. Les objets les plus consommateurs en mémoire tels les *data* dans la classe *Layer* sont par ailleurs stockés dans le tas pour éviter les copies entre les différentes classes.

L'état global contient un pointeur vers l'état de l'environnement et et une liste de pointeur de contenant l'état des différents joueurs de la partie.

On utilise le pattern *Observer* pour notifier les éléments dépendant de l'état lorsque celui-ci change. La classe *State* hérite de la classe *Observable* qui contient une liste d'objets héritant de la classe abstraite *Observer*. Lorsque la classe *State* subit un changement, l'utilisateur utilise la méthode *notifyObservers()* pour notifier ses observers avec l'évènement correspondant au changement appliqué. Il existe 2 types d'*Event* :

— Les *TAB_EVENT* qui représente un changement d'état lié aux joeurs (mort, changement de position , changement d'orientation)

— Les *STATE_EVENT* qui correspondent à des changement liés à l'environnement (changement de niveau) ou à des changements liéés aux statistiques de la partie

ditemie Afin d'éviter des problèmes techniques suite à notre implémentation, nous avons mis en place des fonctions afin de capturer les exceptions rencontrées lors de l'exécution, principalement dans la classe qui charge les données de la carte. De plus nous avons utilisés des entiers signé afin que les valeurs et statistiques des pokemons ne prennent jamais de valeurs négatives.

Nous avons pour l'instant implémenter que les fonction set et get qui permettent d'instancier nos objets liés aux états .

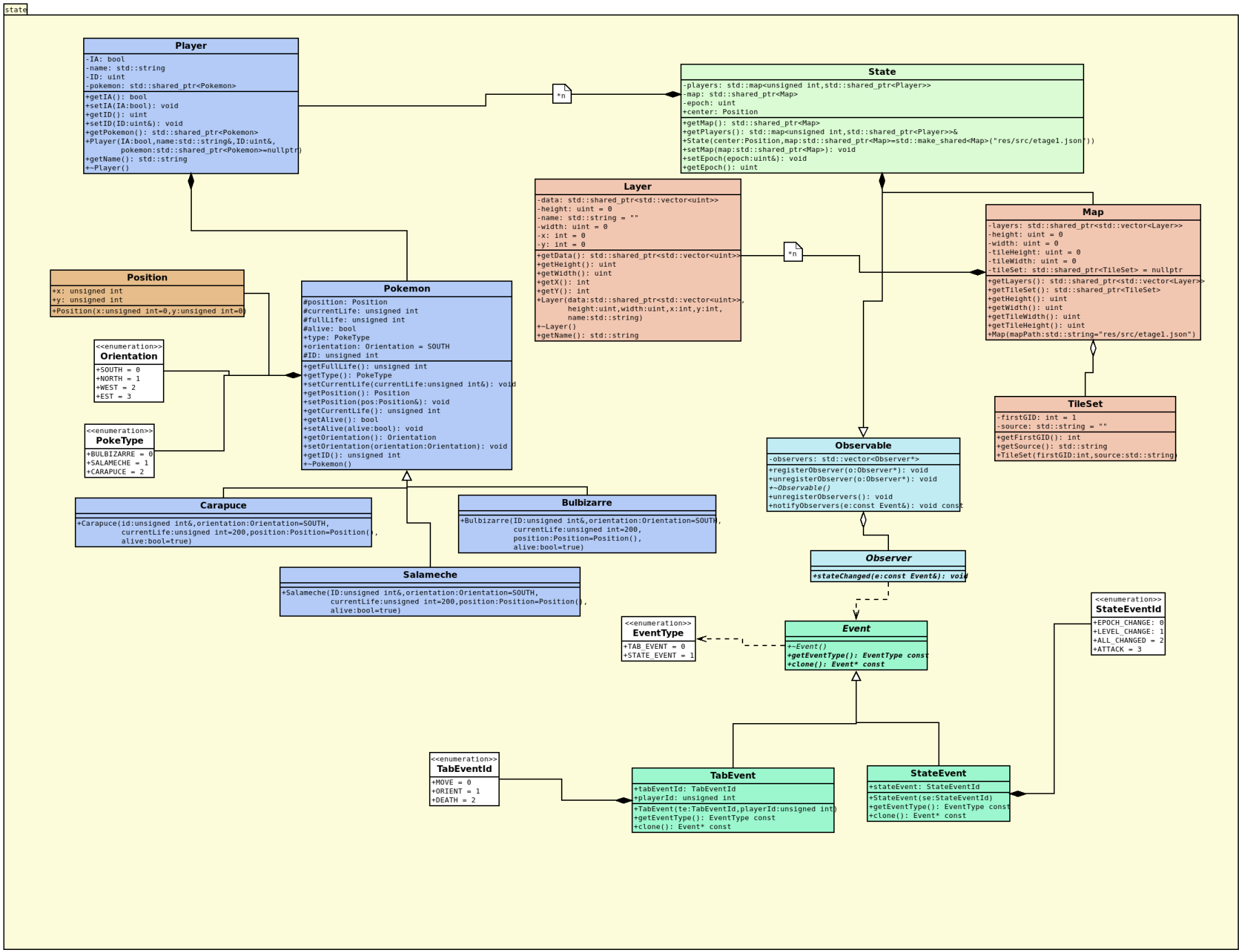

FIGURE 3 – Diagramme des classes d'état.

3 Rendu : Stratégie et Conception

3.1 Stratégie de rendu d'un état

Notre stratégie de rendu d'un état se base sur l'utilisation de l'interface SFML qui s'appuie sur OpenGL afin de générer un rendu en 2D. Nous avons utiliser les fonctions de SFML afin de charger dans le processeur, la liste des éléments à afficher par le processeur graphique.

Nous avons découper notre affichage graphique sur deux niveaux, la carte avec les éléments décoratifs (mur, sol, escalier) et les éléments mobiles à savoir les pokemons (Carapuce, Salamèche, Bulbizarre) qui se superpose sur le claque précédent. On transmet l'état du jeu ainsi que les textures de toute la carte avec leurs coordonnées et la texture des pokemons avec leurs coordonnées et leur orientation à afficher.

La carte étant très grande, nous avons entièrement chargé dans la mémoire pour l'affichage et on a crée une vue (un zoom) centrée sur le pokemon du joueur qui évolue sur la carte, car le but du jeu reste avant tout d'évoluer dans un labyrinthe.

Lorsque qu'un changement d'état se produit, la vue est modifiée en fonction du changement appliqué à l'état. Si le changement modifie uniquement la position des pokemon, seule le rendu des pokemon est mis à jour, si ce changement s'applique à l'environnement l'ensemble du rendu de la carte est mis à jour. Lorsque l'ensemble de l'état est modifié le rendu de l'état est entièrement mis à jour.

Nous avons rajouter sur la scène finale, les statistiques des pokemon (point de vie actuelle et id des pokemon) en haut à gauche.

Pour simplifier au mieux notre logiciel, nous n'avons pas inclut les animations de déplacements lorsque le pokemon évolue d'une case à l'autre ainsi que les animations d'attaques quand un pokemon lance une compétence sur un autre.

3.2 Conception logiciel

Pour afficher un état on crée une scène qui génère instancie un ensemble de pointeurs sur des objets graphiques de types *LayerRender* pour la carte. Pour les Pokemon, on instancie un *Sprite* par Pokemon. Ce *Sprite* contient à la fois les coordonnées du Pokemon sur la fenêtre et sa position sur le Tileset utilisé. Puis, on utilise la méthode *Scene : :draw()* pour déclencher l'ouverture de la fenêtre et l'affichage de l'état. Il est à noter que l'état contient d'ores et déjà toutes les informations nécessaires à l'affichage de la carte. En effet, à la création de l'état celui-ci parse un fichier JSON (ici *map.json*)contenant toutes les informations nécessaires à l'affichage de la carte.

classe *Scene* est un des *Observer* de *State*. Ainsi, lorsque l'état subit un changement, *Scene* est notifié et met à jour le rendu en fonction du type d'évènement qui lui est transmis :

- Si un *TAB_EVENT* (correspondant à changement d'état pour l'un des joueurs) la méthode *updatePlayers()* responsable de la mise à jour du rendu des pokemon est appelée.
- Si un *STATE_EVENT* (correspondant changement d'état lié à l'environnement ou à la partie) la méthode *updateMap()* est appelée dans le cas où le joueur change de niveau (évènement *LEVEL_CHANGE*, les méthode *updateMap()* et *updatePlayers()* sont toutes deux appelées si l'état subit un changement global.

Par ailleurs, les points de vie de chacun des pokemon sont, pour le moment, affichés et mis à jour en temps réel à l'intérieur de la méthode *draw()* de la classe *Scene*.

Chaque objet *LayerRender* parcourt l'ensemble des tuiles d'un étage de la carte, détecte la position de la tuile sur la ressource graphique (i.e : l'image du tileset) et calcule sa position sur la fenêtre. Pour optimiser les performances seule l'image du tileset est chargée dans une *Texture*, l'objet *LayerRender* conserve seulement la position de chaque tuile de la carte sur cette texture. L'objet *PokeRender*fait des opérations identiques pour cette fois ci un seul objet : le pokemon désiré.

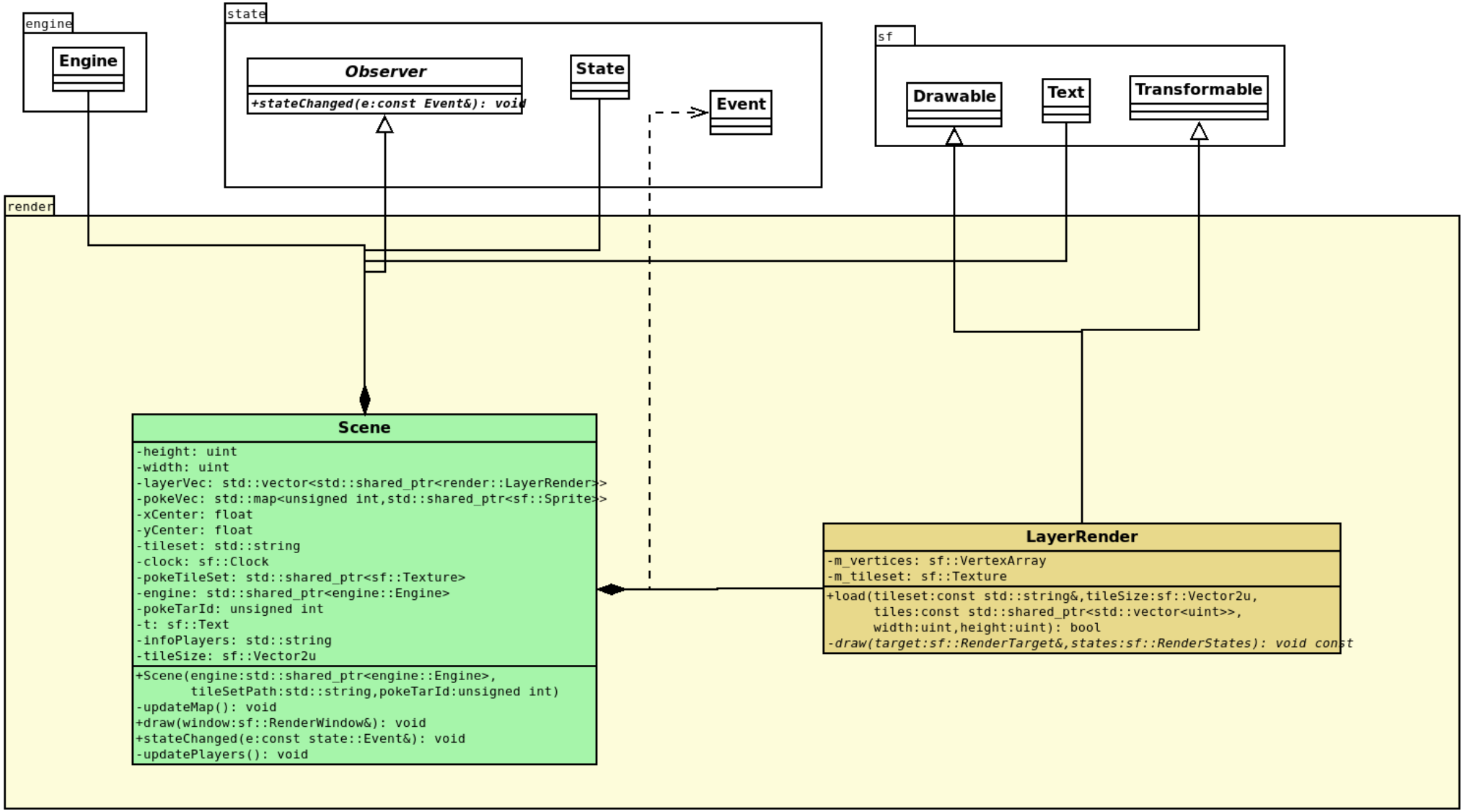

FIGURE 4 – Diagramme des classes de rendu.

4 Règles de changement d'états et moteur de jeu

4.1 Règles

Les changements d'états sont liés aux commandes exécutées par le moteur. Les commandes extérieures sont :

— Commande de déplacement
— Commande d'attaque
— Commande de soin

Une fois qu'une commande extérieure de la part du joueur et de l'IA ont été exécutés (joueur puis IA), on passe au tour suivant.

4.2 Changements autonomes

Les changements autonomes interviennent à la fin de chaque changement d'état lié à une commande extérieure et s'exécutent dans l'ordre suivant :

1. Si le joueur est mort, on affiche "Fin de jeu"
2. Si l'IA est morte, on active la case escalier pour passer à l'étage suivante
3. On met à jour les statistiques (point de vie) de l'IA et du joueur en fonction des règles d'attaque et de soin.
4. On applique les règles de déplacement du joueur et de l'IA.
5. Si le joueur est sur la case d'escalier, on vérifie s'il est autoriser à passer au niveau suivant.

4.3 Conception logiciel

On utilise ici le pattern *Command* : un ensemble d'utilisateur peut ajouter des "commandes" à l'objet *Engine* puis lui demander de les exécuter à l'aide de la méthode *runCommands()*.

Les commandes hérite de la classe abstraite *Command* contient une méthode *execute()* virtual pure qui est appelée par le moteur lorsqu'un utilisateur appelle la méthode *runCommands()*. Cette méthode applique en fait directement des changements à l'objet *State* conservé par la classe *Engine*.

On a implémenté, pour le moment, 3 types de commande :

— *MoveCommand* qui correspond à un changement de position d'un des pokemon

— *AttackCommand* qui correspond à l'attaque d'un pokemon

— *HealCommand* qui est déclenché lorsqu'un joueur veut soigner son pokemon

Par ailleurs, le changement de niveau étant provoqué par un déplacement sur la case escalier après qu'un joueur (non IA) ait éliminé tous ses ennemis ; cet évènement est entièrement géré par la commande*MoveCommand*.

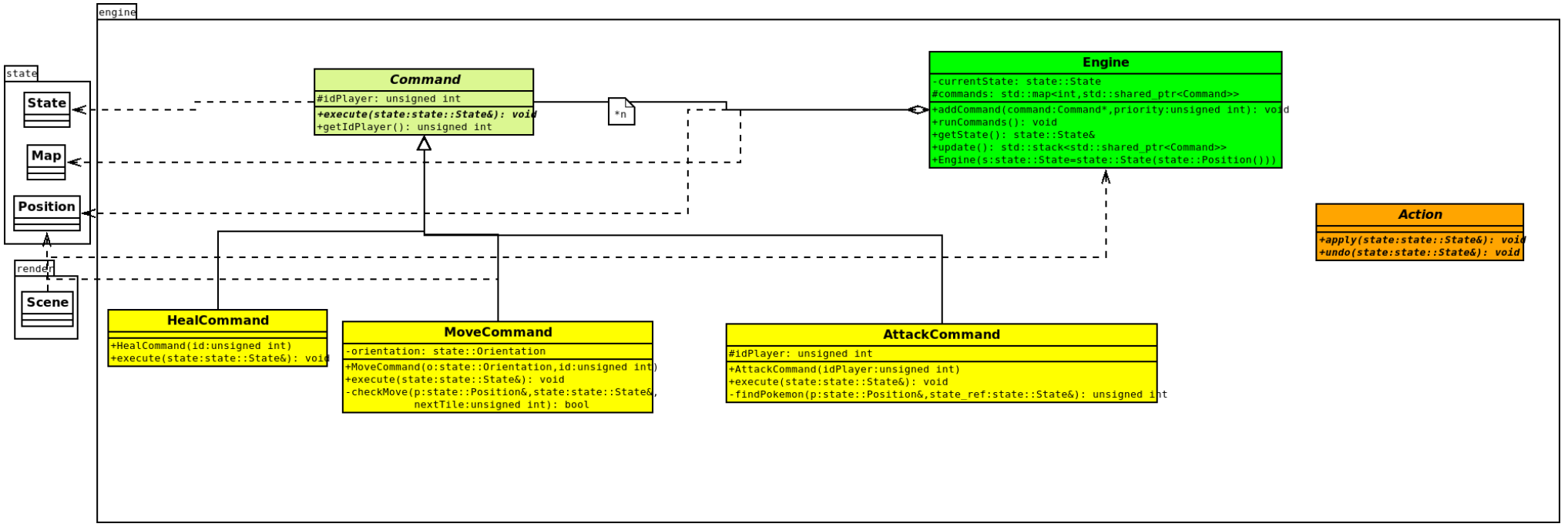

FIGURE 5 – Diagramme des classes de moteur de jeu.

5 Intelligence Artificielle

5.1 Stratégies

5.2 Conception logiciel

6 Modularisation

6.1 Organisation des modules

6.2 Conception logiciel

7 Annexe

7.1 Bibliographie