Projet Logiciel Transversal

Anand Candassamy & Paul Estano

Table des matières

1 Présentation Générale **1**
1.1 Archétype . . . 1
1.2 Règles du jeu . . . 1
1.3 Ressources . . . 2

2 Description et conception des états **4**
2.1 Description des états . . . 4
2.1.1 État de la carte . . . 4
2.1.2 État des joueurs . . . 4
2.1.3 État général . . . 4
2.2 Conception Logiciel . . . 4

3 Rendu : Stratégie et Conception **7**
3.1 Stratégie de rendu d'un état . . . 7
3.2 Conception logiciel . . . 7

4 Règles de changement d'états et moteur de jeu **9**
4.1 Règles . . . 9
4.2 Conception logiciel . . . 10

5 Intelligence Artificielle **10**
5.1 Stratégies . . . 10
5.2 Conception logiciel . . . 11

6 Modularisation **11**
6.1 Organisation des modules . . . 11
6.2 Conception logiciel . . . 12

7 Annexe **13**
7.1 Bibliographie . . . 13

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.
Les combats fonctionnent en tour par tour. A chaque tour, l'utilisateur peut effectuer qu'une seule action :

- attaquer avec une compétence
- soigner le pokemon
- se déplacer

1.3 Ressources

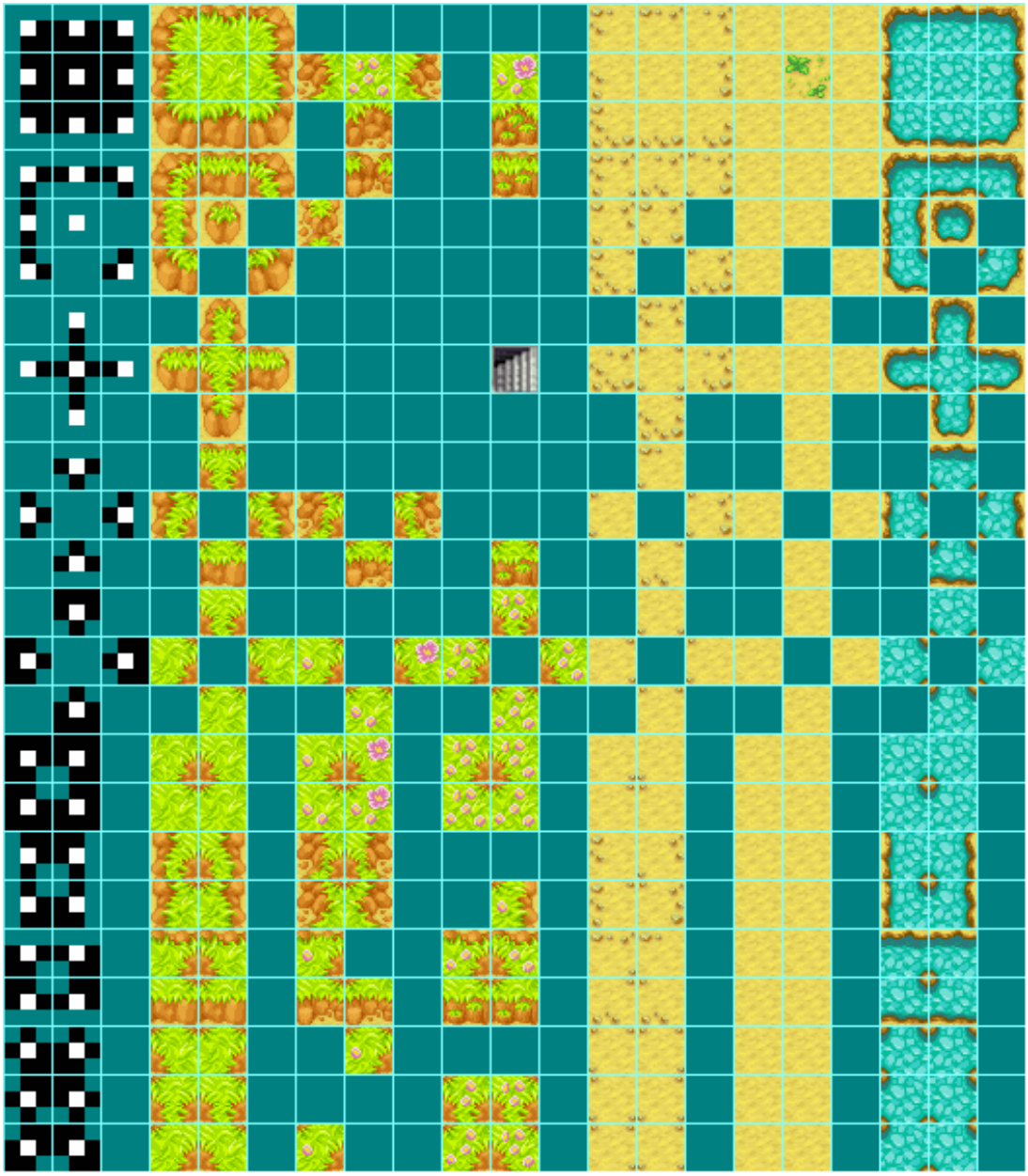

FIGURE 1 – Tileset utilisé pour construire un monde

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 (eau, feu, herbe).

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*. 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.

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 .

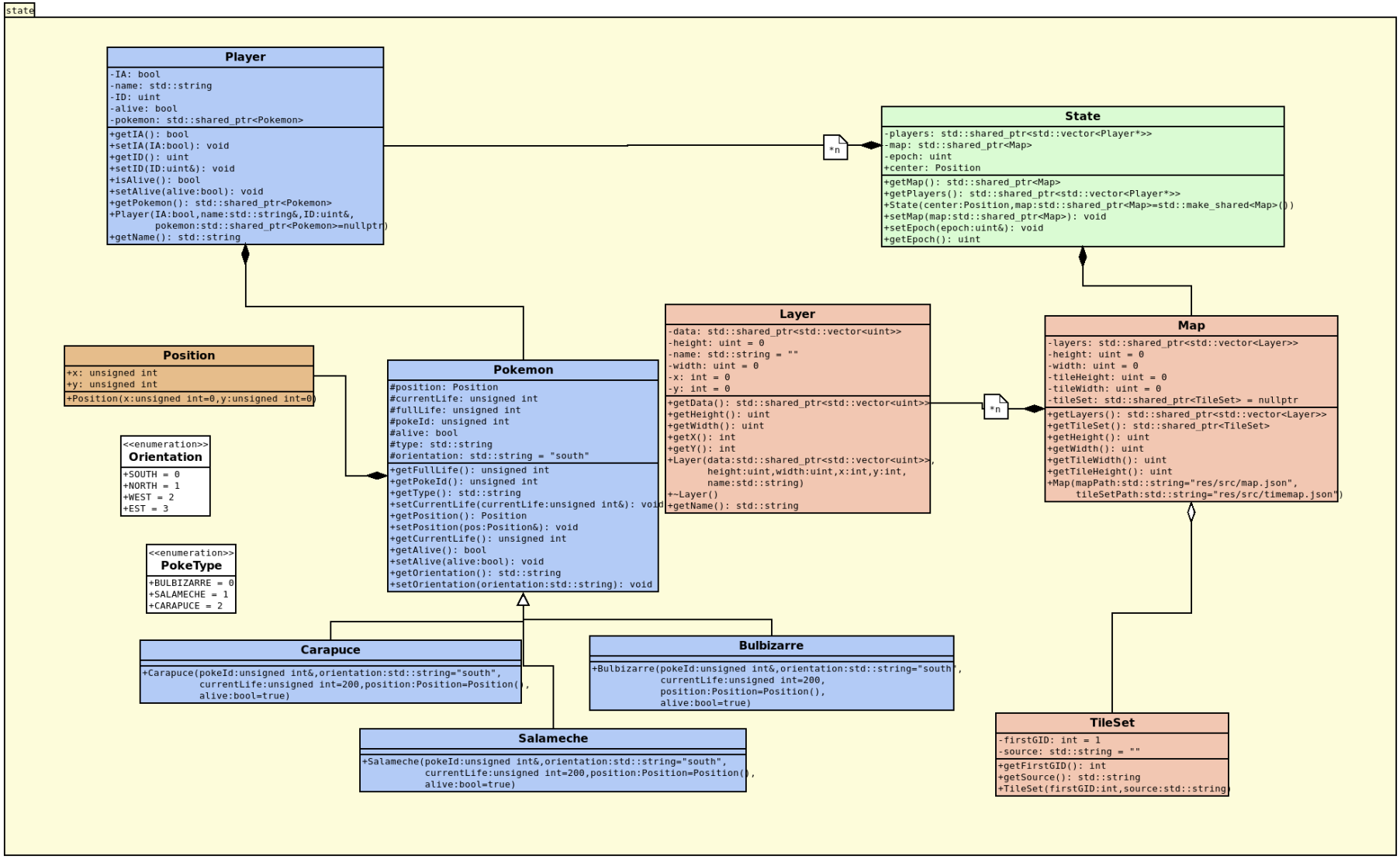

FIGURE 3 – Diagramme des classes d'état.

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, nous regardons si le rendu doit être modifié, dans ce cas là, nous mettons à jour la vue et l'état des éléments mobiles.

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 pokémon 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 et *PokerRender* pour les pokemons. Puis, on utilise la méthode *Scene : :draw()* pour déclencher l'ouverture de la fenêtre et l'affichage de l'état. Lorsqu'un changement d'état survient on peut mettre à jour les objets graphiques sans interrompre l'affichage grâce à la méthode *Scene : :updateState()*. Cette méthode n'est utilisable que si les resources graphiques de la carte ne changent pas. Si l'on veut utiliser de nouvelles ressources graphiques pour afficher la carte, il est nécessaire de réinstancier une scène (ce qui fermera la fenêtre).

Chaque objet *LayerRender* parcourt l'ensemble des tuiles d'un étage de la map, détecte la position de la tuile sur la resource graphique (ie 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 map sur cette texture. L'objet *PokeRender*fait des opérations identiques pour cette fois ci un seul objet : le pokemon désiré.

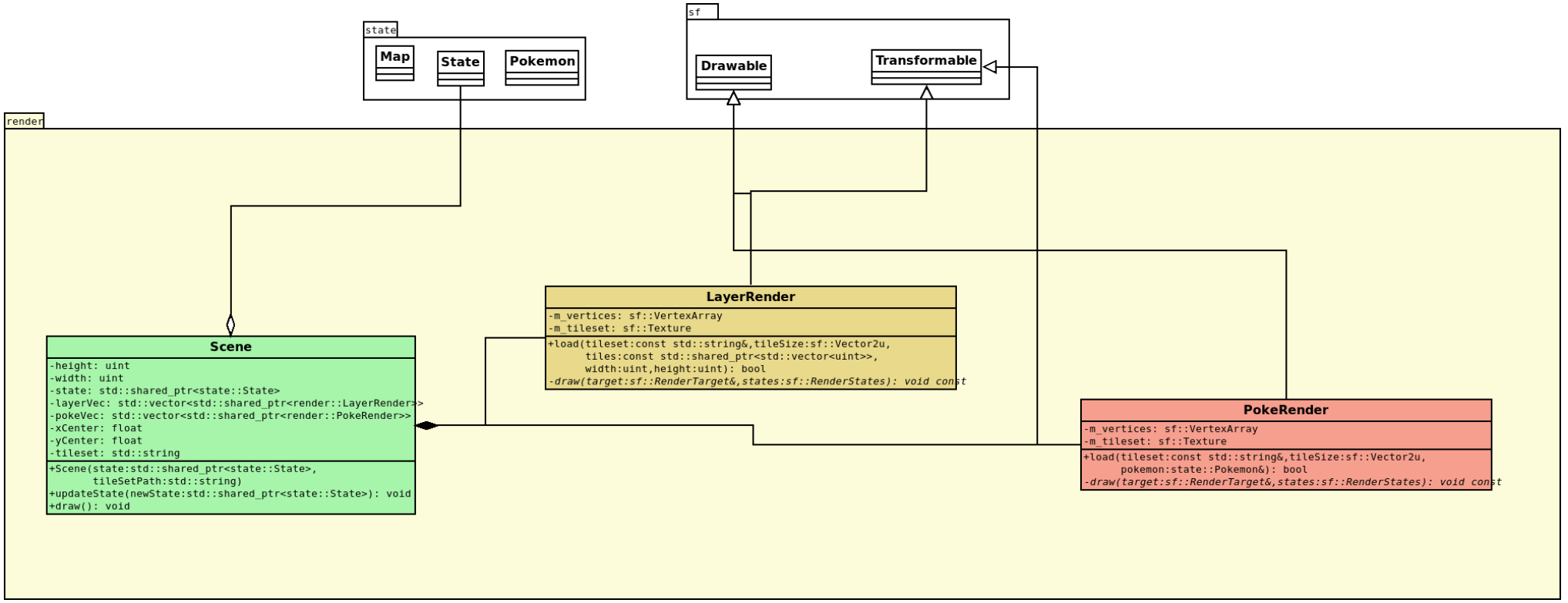

FIGURE 4 – Diagramme des classes de rendu.

4 Règles de changement d’états et moteur de jeu

4.1 Règles

4.2 Conception logiciel

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