Projet Fire Emblem

Thanusan SATHIAKUMAR – Léo CHAZALLET

Table des matières

1 Objectif

1.1 Présentation générale

Le jeu va être un jeu de combat stratégique en tour par tour. Le principe se base sur la partie combat de la série de jeux Fire Emblem.

FIGURE 1 - Image tirée du jeu Fire Emblem Fates : Birthright

Le joueur possède une ou plusieurs unités qui sont placées sur un espace quadrillé. Chacune des unités aura ses propres caractéristiques (points de vie, point de mana, sorts, portée de déplacement, portée d'attaque, valeur d'attaque, valeur de défense). Des ennemis sont présents sur chaque carte/niveau.possédant eux aussi des caractéristiques spécifiques. Le but va donc d'éliminer tous les ennemis présent sur la carte pour obtenir une victoire. A chaque tour le joueur va pouvoir décider pour une unité une action (se déplacer, attaquer, utiliser un sort).

1.2 Règles du jeu

L'utilisateur jouera contre des unités ennemis contrôlées par une IA par défault. Il y aura cependant la possiblité de passer en mode 2 joueurs par la suite.

Voici une liste des règles principales du jeu :

- le/les unité/s du joueur apparaissent du côté opposé de la carte à celle des unités ennemies
- le joueur ne peut effectuer qu'une seule action par unité par tour
- une action peut être soit un déplacement, une attaque ou l'utilisation d'un sort/objet
- la partie se termine quand tous les ennemis sont éliminés ou lorsque le joueur n'a plus d'unité

Le jeu est constitué d'un seul niveau pour l'instant. Il y aura une unité alliée (un héros) et une unité ennemie (un monstre). Celles-ci apparaîtront à un emplacement prédéfini. Ces unités auront des caractéristiques basiques et communes. Chacune aura des coordonnées de position, son orientation, des points de vie, une capacité de déplacement, une portée d'attaque, une valeur d'attaque et une valeur de défense.
Au fur et à mesure des niveaux, le nombre d'unités ainsi que la difficulté va augmenter. D'autres unités seront aussi disponible comme des mages qui pourront avoir en plus différents sorts disponibles, auront des points de mana et qui pourront attaquer à distance.
Chaque niveau aura une carte dédiée qui aura une taille adaptée au nombre d'unités.

S'il reste du temps, voici les fonctionnalités que nous aimerions implémenter :

- présence sur le terrain d'éléments apportant des bonus (points de vie, attaque ou défense) répartis aléatoirement sur la grille du jeu. Le joueur doit choisir de manière stratégique les éléments dont il aura besoin pour vaincre l'ennemi
- présence sur le terrain d'éléments apportant des malus au joueur (par exemple des trous à éviter pour ne pas perdre de la vie...)
- présence de cases permettant la téléportation du joueur d'une case à une autre prédéfinie ou aléatoire (permettant d'offrir au jeu un aspect stratégique)

1.3 Conception Logiciel

FIGURE 2 – Carte de la zone de combat

La zone de combat est une image de 800 par 800 pixels décomposée en 25 tiles sur la longeur et la largeur (un tile est un carré de 32 par 32 pixels).

FIGURE 3 - Sprite d'un personnage

Pour tout ce qui est gestion du texte, nous utiliserions les fonctionnalités proposées par la bibliothèque SFML.
Nous n'avons pas pu mettre l'ensemble des ressources que nous allons utiliser pour le jeu pour éviter de surcharger le rapport. Il est possible de tous les trouver dans le dossier « rsc » récupérable sur le GitHub: https://github.com/LeoChazl/plt/tree/master/rsc .

2 Description et conception des états

2.1 Description des états

Un état du jeu est composé d'entités fixes qui vont constituer la carte (les obstacles et les espaces) sur laquelle des éléments mobiles vont évoluer (les personnages, le curseur). Ces derniers sont associés à un joueur et possèdent de nombreuses caractéristiques (vie, valeur d'attaque, armure, ...).

2.1.1 Etat général

Il faut être capable de savoir dans quelle situation et à quel avancement se trouve le jeu.
Un nombre permet d'identifier le tour de jeu actuel et un autre indique le niveau.
Un booléen informe le système si la partie est terminée.
L'état général a aussi accès à tout ce qui constitue le jeu :

- la liste des joueurs qui ont eux-mêmes une liste indiquant les unités qu'ils contrôlent
- la carte du niveau avec des informations sur ses dimensions
- le curseur permettant au joueur d'interagir avec le jeu

2.1.2 Etats éléments fixes

L'ensemble des éléments fixes du jeu sont contenus dans un tableau à deux dimensions. Le tableau a des dimensions différentes en fonction du niveau et donc de la carte (le premier niveau est de taille 25x25 cases).
Il n'y a que deux types de cases possible :

- les obstacles : les emplacements sur lesquels les unités ne pourront se déplacer
- les espaces : les emplacements sur lesquels les unités pourront se déplacer

Le type de l'emplacement est identifiable grâce à un booléen et la position de chaque case est accessible via leur emplacement dans le tableau.

2.1.3 Etats éléments mobiles

Les éléments mobiles sont toutes les entités sur lesquelles le joueur pourra agir. Ils partagent tous une caractéristique commune : une position (x et y).

Les unités

Toutes les unités partagent les caractéristiques suivantes :

- points de vie
- capacité de déplacement
- valeur d'attaque
- valeur d'armure

- identité
- statut
- direction

L'identité permet de savoir le type d'unité. Il y a pour l'instant 3 types d'unités :

- le troll : une unité ennemie qui attaque au corps à corps et n'est capable que d'une attaque basique
- le chevalier : une unité alliée similaire au troll en terme de capacités
- le mage : une unité alliée ou ennemie qui en plus de pouvoir attaquer au corps à corps sera capable de lancer un sort et donc attaquer d'une plus grande portée. Ses capacités au combat rapproché est cependant bien inférieure au 2 unités précédentes. Pour pouvoir gérer le nombre de sorts que celui-ci peut envoyer, cette unité a une caractéristique mana supplémentaire.

Le statut indique l'état et la disponibilité du personnage. Il existe 4 status possible :

- sélectionné : le curseur pointe sur le personnage
- disponible : une action peut encore être effectué par le personnage pendant le tour actuel
- attente : le personnage a déjà effectué son action pour le tour
- mort : les points de vie du personnage sont à 0

Enfin la direction nous donne l'information sur l'orientation de l'unité et permet d'afficher la sprite correspondante. Il y a donc 4 direction possible : haut, bas, droite et gauche.

Le curseur

Cette entité va permettre d'indiquer au joueur ce qu'il sélectionne pour lui permettre de faire ses actions ou pour lui donner des informations.

2.1.4 Etat du joueur

Le joueur aura un identifiant pour le différencier des autres ainsi qu'un nom que le joueur pourra choisir selon sa préférence. Cet état contiendra aussi la liste de toutes les unités à son commandement.

2.2 Conception logiciel

Le diagramme des différents états de notre jeu est illustré via la figure (n°4). Ce diagramme permet de mettre en évidence les groupes de classes suivants.

2.2.1 Classes générales

La classe State est le conteneur principal à partir duquel on peut accéder à toutes les données de l'état. Ainsi l'appel à cette classe permet d'instancier l'ensemble de l'état du jeu à un moment donné. Cette classe est en relation de composition avec la classe EntityMap et la classe Player. Pour ce dernier c'est une liste de pointeurs vers ce type d'objet qui permet la composition. Grâce à cela il est possible d'échelonner l'aspect multijoueur de notre jeu à un nombre plus important de joueurs..

Player permet d'instancier pour chaque joueur prenant part à une partie les différentes unités qui lui sont associées via une liste de référence de pointeurs vers les unités mobiles. De cette manière, il sera possible de rajouter autant d'unité que nécessaire dans les cas où le joueur possèderait de nombreuses unités.

Les modifications de l'état du jeu sont possibles en suivant un design pattern Observer et notamment grâce à l'interface IObserver. Cette interface en relation avec le rendu permet de prendre en compte l'ensemble des différentes modifications apportées par le joueur pour en modifier l'état.
Ainsi à l'arrivée d'un événement, la classe StateEvent en relation de contrainte avec IObserver détermine le type de changement à apporter sur l'état du jeu selon la liste d'énumération qui lui est attribué (changement sur le joueur, des unités mobiles, changement du tour de jeu ou du niveau).
La vitesse de traitement des événements pouvant varier, les événements reçus sont enregistrés dans une liste "obsevers" de type Obsevers. Cette liste permet de notifier l'ensemble des observer des événements à traiter.

2.2.2 Classes fixes

Les éléments statiques de la carte sont regroupés sous la classe EntityMap. Celle-ci contient des objets StaticEntity dont hérite les classes filles Obstacle et Space.

EntityMap permet d'instancier une grille de jeu sous la forme d'un tableau à deux dimensions contenant des pointeurs vers des objets Obstacle et Space. Cette grille appelée mapArray, est formée via la lecture d'un fichier texte constitué d'une suite de "1" (pour les espaces) et "0" (pour les obstacles). Ainsi à chaque niveau du jeu la génération de la map peut être rendu flexible selon le fichier texte ouvert par la fonction de génération du tableau mapArray.

Les classes Obstacle et Space possèdent une méthode qui permet de les différencier dans le tableau et ainsi connaître les emplacements sur lesquels les unités pourront se déplacer.

2.2.3 Classes mobiles

Les éléments mobiles de notre jeu sont regroupés via la classe abstraite MobileEntity. A partir de cette classe abstraite hérite plusieurs classes filles : Troll,Knight,Mage. Ces classes héritent des caractéristiques similaires issues de la classe mère (vie, armure, capacité de déplacement, valeur d'attaque, vie maximale possible, direction, statut et type d'unité). Les 3 dernières caractéristiques sont gérées avec des énumérations. Pour la direction c'est une énumération de 4 éléments (haut, bas, gauche et droite) alors que le statut en possède 4 (sélectionné, disponible, en attente et mort). Enfin, l'EntityId permet leur unicité en tant que classe avec 3 types possibles correspondant aux classes filles.
Les classes filles possèdent cependant des particularités permettant de les différencier notamment une méthode renvoyant l'élément de l'énumération d'EntityId correspondant ou encore via les différents types d'attaque possible pour un Mage amenant des méthodes et attributs particulière à cette classe.

Les coordonnées des différentes unités mobiles sur la carte sont instanciées via la classe Position dont hérite la classe fille MobileEntity.

De plus cette classe Position est la classe mère de la classe Cursor. Cette classe permet de déplacer les coordonnées du curseur sur une case particulière de la map. Ainsi elle peut être utilisé pour visualiser les caractéristiques de l'ennemi, pour choisir les coordonnées de déplacement d'une unité par le joueur ou pour viser une cible.

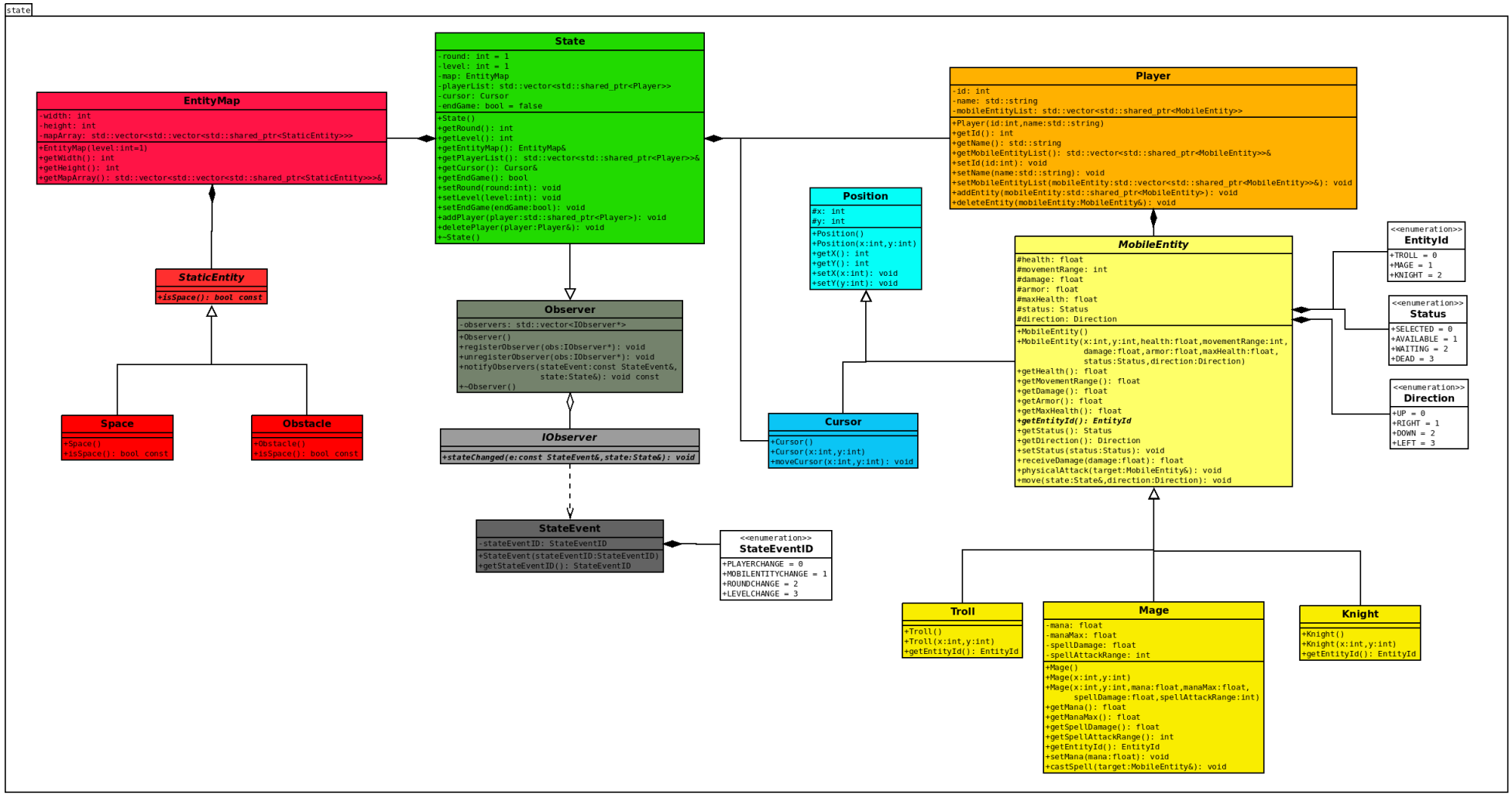

FIGURE 4 - Diagramme des classes d'état

3 Rendu : Stratégie et Conception

Présentez ici la stratégie générale que vous comptez suivre pour rendre un état. Cela doit tenir compte des problématiques de synchronisation entre les changements d'états et la vitesse d'affichage à l'écran. Puis, lorsque vous serez rendu à la partie client/serveur, expliquez comment vous aller gérer les problèmes liés à la latence. Après cette description, présentez la conception logicielle. Pour celle-ci, il est fortement recommandé de former une première partie indépendante de toute librairie graphique, puis de présenter d'autres parties qui l'implémente pour une librairie particulière. Enfin, toutes les classes de la première partie doivent avoir pour unique dépendance les classes d'état de la section précédente.

3.1 Stratégie de rendu d'un état

3.2 Conception logiciel

3.3 Conception logiciel : extension pour les animations

3.4 Ressources

3.5 Exemple de rendu

Illustration 1: Diagramme de classes pour le rendu

4 Règles de changement d'états et moteur de jeu

Dans cette section, il faut présenter les événements qui peuvent faire passer d'un état à un autre. Il faut également décrire les aspects lié au temps, comme la chronologie des événements et les aspects de synchronisation. Une fois ceci présenté, on propose une conception logiciel pour pouvoir mettre en œuvre ces règles, autrement dit le moteur de jeu.

4.1 Horloge globale

4.2 Changements extérieurs

4.3 Changements autonomes

4.4 Conception logiciel

4.5 Conception logiciel : extension pour l'IA

4.6 Conception logiciel : extension pour la parallélisation

Illustration 2: Diagrammes des classes pour le moteur de jeu

5 Intelligence Artificielle

Cette section est dédiée aux stratégies et outils développés pour créer un joueur artificiel. Ce robot doit utiliser les mêmes commandes qu'un joueur humain, ie utiliser les mêmes actions/ordres que ceux produit par le clavier ou la souris. Le robot ne doit pas avoir accès à plus information qu'un joueur humain. Comme pour les autres sections, commencez par présenter la stratégie, puis la conception logicielle.

5.1 Stratégies

5.1.1 Intelligence minimale

5.1.2 Intelligence basée sur des heuristiques

5.1.3 Intelligence basée sur les arbres de recherche

5.2 Conception logiciel

5.3 Conception logiciel : extension pour l'IA composée

5.4 Conception logiciel : extension pour IA avancée

5.5 Conception logiciel : extension pour la parallélisation

6 Modularisation

Cette section se concentre sur la répartition des différents modules du jeu dans différents processus. Deux niveaux doivent être considérés. Le premier est la répartition des modules sur différents threads. Notons bien que ce qui est attendu est un parallélisation maximale des traitements: il faut bien démontrer que l'intersection des processus communs ou bloquant est minimale. Le deuxième niveau est la répartition des modules sur différentes machines, via une interface réseau. Dans tous les cas, motivez vos choix, et indiquez également les latences qui en résulte.

6.1 Organisation des modules

6.1.1 Répartition sur différents threads

6.1.2 Répartition sur différentes machines

6.2 Conception logiciel

6.3 Conception logiciel : extension réseau

6.4 Conception logiciel : client Android

Illustration 3: Diagramme de classes pour la modularisation