Projet Logiciel Transversal

Projet worms

Grégoire de Faup – Antoine Delavoypierre

Figure 1 - Worms Armageddon
© Steam

Sommaire

1 Présentation générale

1.1 Archétype

L'objectif que nous nous sommes fixé est de créer un jeu de type worms 2D mais bien sûr avec des fonctionnalités qui nous seront propres. C'est-à-dire :

- Génération d'une nouvelle carte à chaque partie
- Possibilité de détruire les éléments se trouvant sur notre carte
- Possibilité de jouer avec des amis ou contre une IA
- Création d'une équipe comprenant 1 à 5 personnages (peut varier selon le nombre de joueurs)
- Les positions de départ sont choisis par les joueurs sans qu'ils puissent savoir où se placent les autres
- Chaque personnage possède ses propres statistiques et ses attaques et a en plus des atouts.

Si cela est possible nous aimerions :

- Sauvegarder des données à chaque partie, ce qui permettrait de faire progresser les personnages (meilleures statistiques/atouts, attaques plus fortes etc)

Un personnage aura les statistiques suivantes :

- Vie
- Points de déplacements
- Nombre d'attaques/tour

Les atouts permettent aux personnages de booster ces caractéristiques : plus de vie ou résistance aux coups, plus de déplacements et plus d'attaques/tour.

1.2 Règles du jeu

Le joueur commence sa partie après avoir composé une équipe. Une carte de jeu est alors créée. Une fois la nouvelle carte affichée le joueur pourra placer un à un tous ses personnages en partant du haut de la carte et ce en un temps donné. Un joueur aura par exemple 15s pour déplacer ses personnages depuis le haut et les placer là où il le souhaite. Passé ce délai les personnages tomberont en chute libre.

Le jeu peut alors vraiment commencer. Puisqu'il s'agît d'un jeu tour par tour les joueurs joue l'un après l'autre et non simultanément. L'ordre de jeu est choisi aléatoirement au début de la partie.

- Pendant son tour le joueur sélectionne **un seul** de ses personnages
    - Son personnage peut alors se déplacer, utiliser son atout et attaquer
- Une fois ses points de mouvements et d'attaques utilisés le tour de ce joueur est automatiquement terminé (toutefois un joueur est libre de terminer son tour avant).
- Une fois qu'un atout a été utilisé il faut attendre qu'il se recharge après un temps aléatoire pour le réutiliser.
- Lorsqu'un personnage a perdu toute sa vie à cause des attaques des autres joueurs (ou de chutes) il disparaît.
- Le joueur possédant le dernier personnage encore en jeu gagne la partie.

1.3 Ressources

Pour réaliser la carte de jeu nous utiliserons, un ciel, un terrain, une frontière sol / terrain ainsi qu'un masque. Un masque généré de manière aléatoire permet de définir la frontière entre le sol et le terrain et ainsi des cartes aléatoires à chaque partie.

Figure 2 - Texture ciel terrain et frontière

Concernant les personnages nous utilisons des sprites trouvés sur internet. Voici quelques un des personnages choisis avec leurs mouvements et/ou des attaques. Cette section est amenée à évoluer.

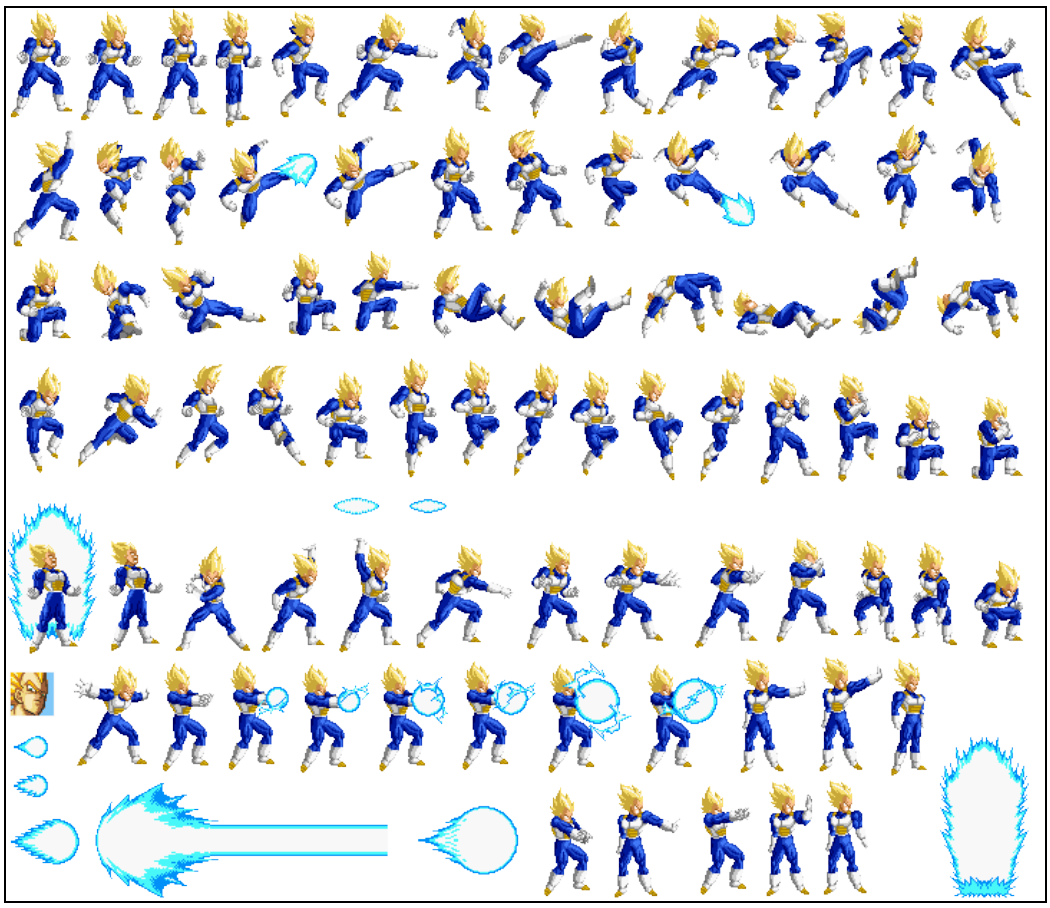

Figure 3 - Sprite Vegeta

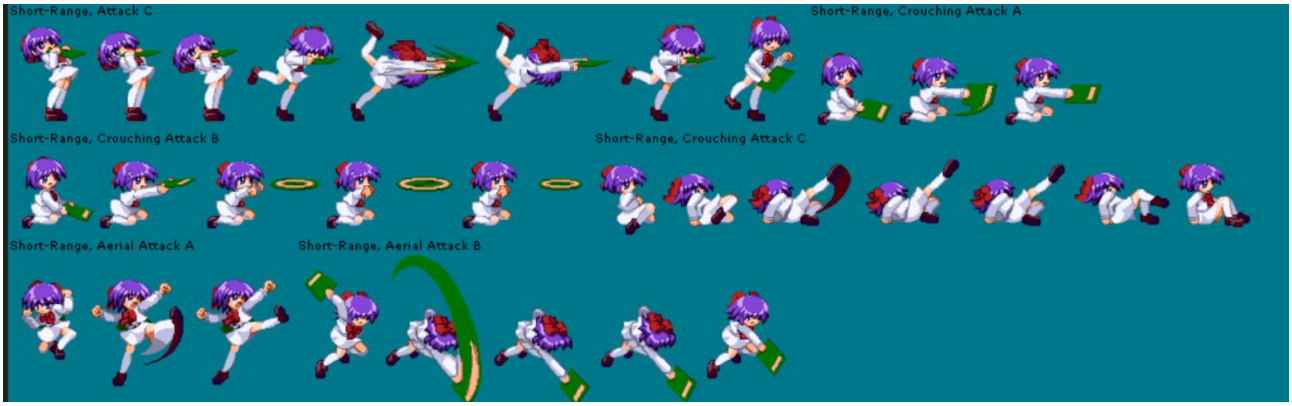

Figure 4- Mio Kouzuki

2 Diagramme et conception des états

2.1 Description des états :

Dans la conception, chacun de ces éléments est décrit par une classe :

La **classe GameState** contient une instance de l'ensemble des éléments qui compose l'état du jeu. C'est à partir de cette classe que le moteur pourra notamment consulter l'état actuel du jeu et le modifier

La **classe Player** défini un joueur. Cette classe contient notamment une chaîne de caractère indiquant le nom du joueur (celui qui sera affiché à l'écran) ainsi qu'une liste de pointeur vers les instances des personnages qui compose son équipe. Cette classe est directement instancié dans la classe GameState.

La **classe Characters** est la classe qui défini ces personnages. Cette classe contient le nom du personnage, le coût, les effets et les champs d'actions des différentes attaques réalisable par le personnages et enfin une instance de ses statistiques ainsi que de sa position. Cette classe est instancié dans la classe Player permettant ainsi de savoir à quelle équipe appartient chaque personnage.

La **classe Statistics** est la classe qui défini les statistiques d'un personnage, c'est à dire ces points de vie, de déplacement et d'attaque disponible à chaque tour. Cette classe est instancié dans la classe Characters étant donné qu'elle est directement lié au personnage.

La **classe Position** défini simplement la position d'un personnage sur la carte. Elle est instancié dans la classe Characters.

Enfin, la **classe Map**, défini l'état actuel de la carte sous forme de matrice d'une taille égale à la résolution de l'écran de l'ordinateur utilisé. Les valeurs de cette matrice définissent la nature du terrain à cette endroit de la carte. Cette classe est directement instancié dans GameState.

Ainsi notre état de jeu est défini comme le diagramme UML ci-dessous le représente :

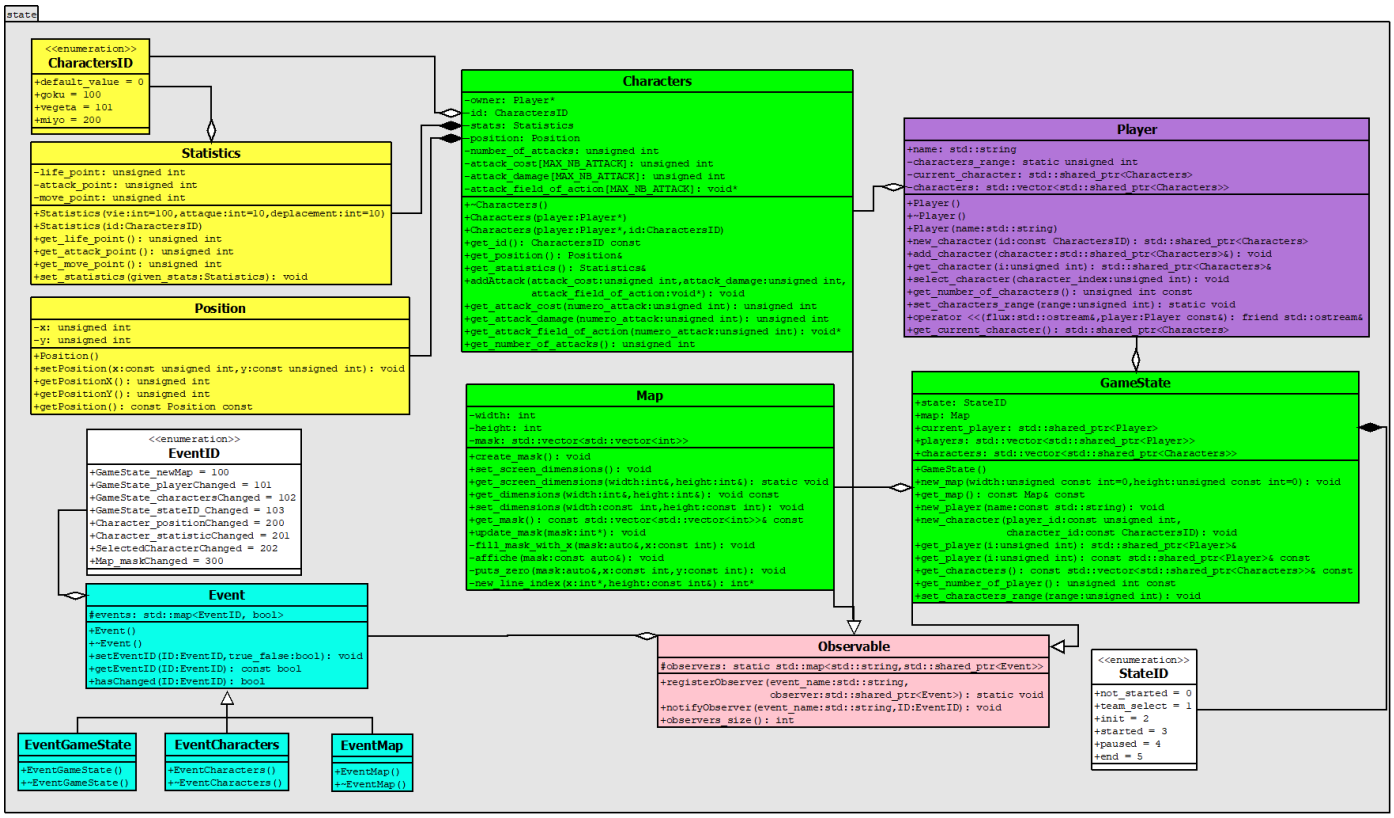

Figure 5 - Diagramme des classes d'états

3 Rendu : Stratégie et Conception

La stratégie adoptée pour réaliser le rendu d'un état de jeu est celle de la décomposition en layer. Tous les éléments graphiques ont une nature différente et leur affichage peut être traité de manière séparée étant donné qu'ils ne sont pas modifiés à la même fréquence.

Nous commençons donc par créer un système d'héritage pour tous les layers. Une classe parente Layer fournit les méthodes et attributs nécessaires pour gérer tous types de layer et les filles redéfinissent si besoin les méthodes selon les spécificités de leur layer.
Les classes filles de **Layer**, **Character** et **Background** définissent respectivement les éléments de terrain et les différents personnages décrits dans l'état de jeu. À tout moment notre rendu est donc composé de deux instances de **Layer.** C'est également dans ces classes que l'on stocke les ressources visuelles permettant l'affichage.

La classe **Surface** décrit nos éléments « dessinable », c'est à dire les éléments sprites de SFML qui seront affiché à l'écran ensuite. Les méthodes de cette classe permettent directement de charger des textures dans ces sprites et de les afficher ensuite. Chaque instance de layer possède sa propre instance de Surface.

La classe **TileSet** décrit les ressources visuelles nécessaires pour créer nos sprites. Concrètement c'est une suite d'élément Image de SFML destiné à servir de texture. Cette classe est également instanciée dans chaque layer.

La classe **Tile** permet simplement de définir les rectangles des sprites, c'est à dire quelle partie des spritesheets le sprite devra afficher.

Enfin, la classe **Scene** permet de piloter tout cela ! Lorsque l'on créé une instance de Scene on fournit l'état courant aini que la map courante. Le constructeur créé ensuite des instances pour chacun des layer définit.
Pour dessiner une scène ou la mettre à jour on utilisera sa méthode draw ou update qui appelle les méthodes correspondantes des layers.

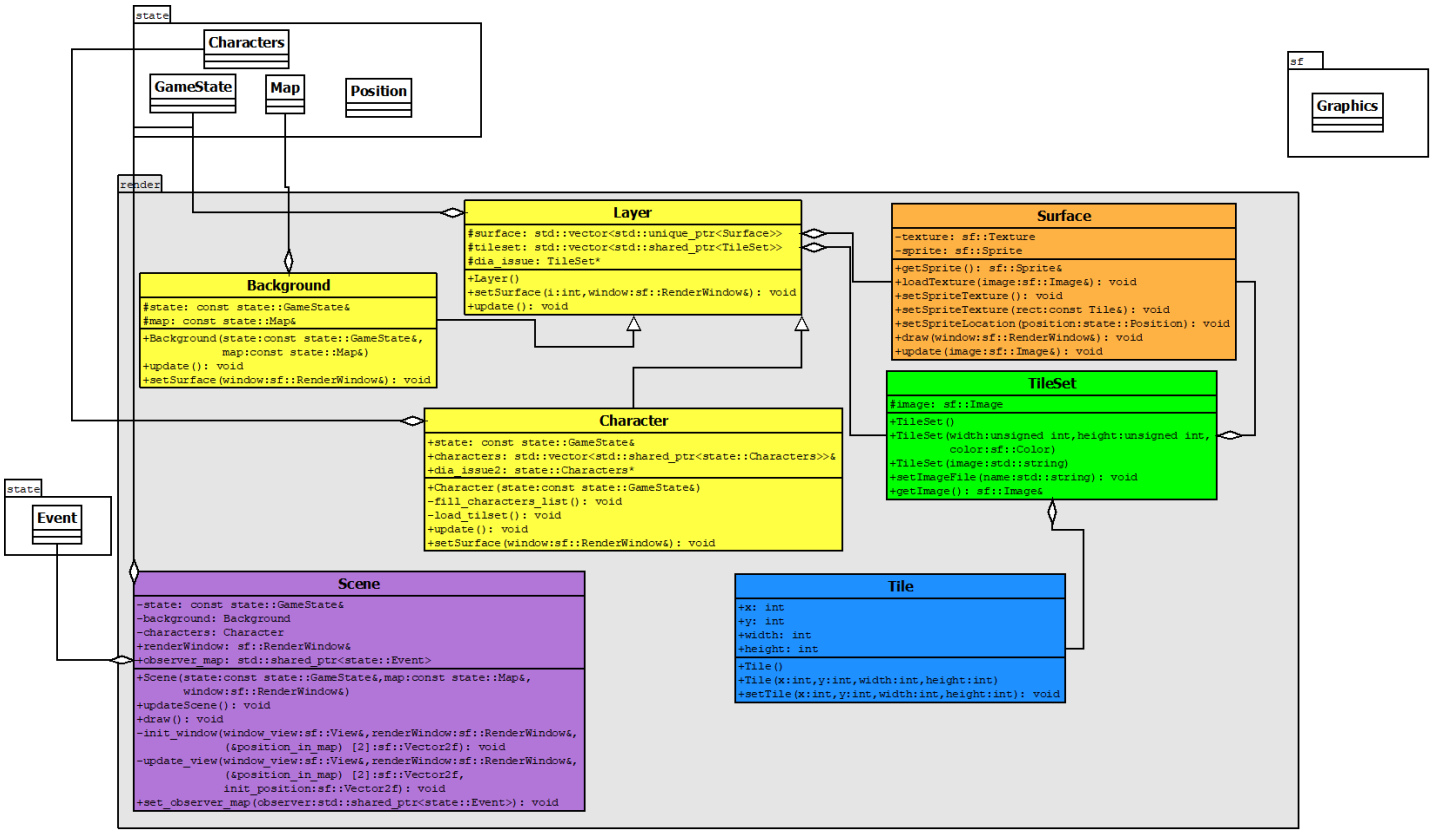

Figure 6 - Diagramme des classes de rendu

4 Règles de changement d'états et moteur de jeu

Détaillons maintenant le fonctionnement du moteur de jeu qui nous permet de modifier l'état de jeu.

Tout d'abord le principe du moteur de jeu est de réagir à des commandes provenant soit d'un joueur soit d'une IA pour déclencher des actions modifiant l'état de jeu courant. Il existe à l'heure actuelle 4 commandes :

- Déplacement :

  Un joueur peut déplacer le personnage actuellement sélectionné en utilisant les flèches directionnels du clavier (haut, bas, gauche, droite). Cela à pour effet de modifier sa position sur la carte du jeu.

- Lancement d'une attaque :

  A l'aide de la souris un joueur peut ordonner au personnage qu'il a choisi de lancer une attaque. Cette commande change les statistiques des différents personnages impactés par cette attaque et également l'aspect de la carte de jeu.

- Choisir un personnage :

  Permet de choisir le personnage que l'on utilisera pendant notre tour de jeu.

- Changement de joueur :

  Cela revient en fait à finir son tour. Cette commande change le joueur sélectionné et passe la main au joueur suivant.

Nous avons construit le GameEngine à l'aide de classes – décrites plus loin -.

La classe GameEngine utilise les entrées données par un utilisateur (décrite dans UserInput) pour instancier des commandes et lancer l'exécution de celle-ci.

La classe Command est une classe abstraite symbolisant une commande et définissant des méthodes virtuelles devant être présente dans chacune des commandes qui seront défini comme des classes fille de Command.
La classe Move décrit les paramètres d'une commande de mouvement et contient les méthodes permettant de vérifier la légalité de la commande puis de l'exécuter

La classe Attack décrit les paramètres d'une commande d'attaque (position à attaquer et nature de l'attaque) et contient les méthodes permettant de vérifier la légalité de la commande puis de l'exécuter

La classe ChangeCharacter décrit les paramètres d'une commande de changement de personnage sélectionné et contient les méthodes permettant de vérifier la légalité de la commande puis de l'exécuter

La classe ChangePlayer décrit la commande permettant de passer son tour et implémente la méthode permettant de l'exécuter.

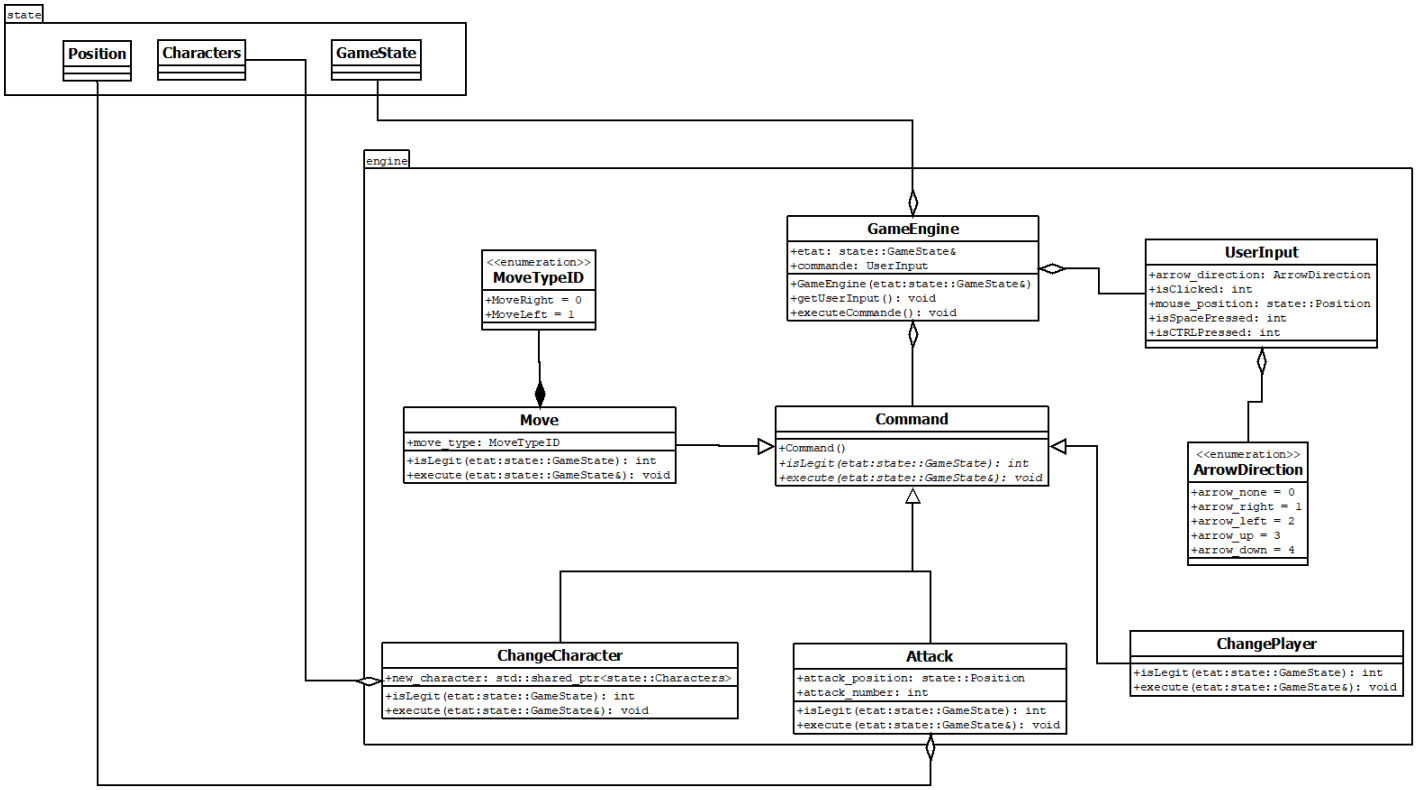

Figure 7 - Diagramme des classes de moteur de jeu

Table des figures :