Projet Ninja Saga

Moïse BEUGRE – Ouajih DADAOUA

Figure 1 - Exemple du jeu Ninja Saga

Table des matières

Objectif

1.1 Présentation générale

L'objectif de ce projet est la réalisation du jeu "Ninja Saga", avec les règles les plus simples. Un exemple est présenté en Figure 1. Des combats entre personnages pourront être faits en tenant compte des points de vie, des points de mouvement en possession selon le personnage choisi.

Les personnages auront des caractéristiques qui influencent le combat, par exemple plus un personnage sera fort grand et imposant moins il sera rapide mais ses attaques auront plus d'impact.

Modes de jeu :

-Multijoueur : Player vs Player, chaque joueur choisit un personnage et initie ensuite le combat.

-Solo : Player vs IA , le joueur choisit un personnage et le personnage contre qui il veut se battre et initie un combat contre l'ordinateur (possibilité de choisir la difficulté).

-Survie : Player vs IA, le joueur choisit un personnage et va se battre contre tous les personnages du jeu en gardant à chaque fois le nombre de points de vie avec lesquelles il a terminé le combat précédent sachant que la difficulté augmente crescendo.

Type de personnages :

-Thork -> Fort, grand et robuste il se caractérise par ses attaques dévastatrices bien aidé par son marteau ravageur. Cependant il accuse un déficit en terme de vitesse et de réactivité.

-Flint -> Connu pour sa vitesse éclair ainsi que ses pyro-attaques, il se démarque par ses réflexes supersoniques qui lui permettent d'avoir toujours un temps d'avance sur ses adversaires.

-Seku -> Grand maître des arts martiaux il est connu pour ses attaques corps à corps redoutables et précises.

-Kuro -> Célèbre Ninja reconnu pour ses attaques furtives, maître du Kung fu et du Nunchaku, très véloce mais aussi moins robuste que ses compères.

Type de coups:

- Coup de poing & Coup de pied : ce coup cause des dommages valant 20 points de vie à l'adversaire sinon 10 point s'il est en position de défense.
- Uppercut & Flash Kick : ce coup cause des dommages valant 30 points de vie à l'adversaire sinon 15 points s'il est en position de défense.

  L'uppercut est très puissant pour repousser l'adversaire il a un seul désavantage c'est que sa zone d'impact est très réduite. Quant au Flash Kick, il est rapide et efficace. Il couvre un large champ d'action et limite les interventions et parades de l'opposant.

1.2 Règles du jeu

Au début de la partie, chaque joueur choisit un combattant et les combattants ont le même nombre de points de vie. La partie se termine lorsqu'un joueur n'a plus de points de vie.

1.3 Conception Logiciel

L'affichage repose sur les textures suivantes:
Les textures de personnages sont de tailles 100x100 pixels tandis que les textures de terrains de taille 700x400.

Les textures de personnages ont été ajustées de sorte à avoir quatres différentes lignes de sprites:

- La première ligne représente la position de ''garde''. C'est la position initiale du Fighter en début de partie.
- La deuxième ligne met en animation l'attaque ''coup de poing" d'un Fighter.

- La troisième ligne est le mode ‘’defense” qui permet de réduire les dégâts subis lors d’une attaque
- La dernière sera utilisé lors le Fighter n’a plus de points de vie. C’est le status ‘’DEAD”.

Figure 2 - Textures pour les personnages

Figure 3 - Textures pour les terrains de combat

Figure 4 - Textures pour les tuiles de lettres et chiffres

2 Description et conception des états

2.1 Description des états

Un état du jeu est formé d'un ensemble d'éléments mobiles (les combattants) dont l'état varie au cours de la partie et d'un décor statique.

2.1.1 Etats des combattants

Un combattant est soit contrôlé par un joueur soit contrôlé par une Intelligence Artificielle (I.A), il possède un nombre de points de vie à l'état t, un nom et un ensemble de statistiques(Points de vie, Points de magie, Combo) qui lui sont attribués en début de partie. Lorsqu'un combattant effectue un nombre de coups réussi dépassant un seuil, il entre dans un statut *Spécial* qui lui donne droit à des attaques plus puissantes. Le combattant est considéré comme étant "mort" si son nombre de points de vie atteint 0.

2.1.1 Etats de l'arrière plan (FighterTerrain)

L’arrière plan représente un terrain attribué à chaque combattant. Les combattants sont donc favorisés lorsque le combat se déroule sur le terrain qui leur ait attribué. Les statistiques d’attaques et de défense sont améliorées.
Le passage d’un terrain à un autre peut représenter le passage à un niveau supérieur lors d’une partie.

2.2 Conception logiciel

Le package état peut se diviser en trois sous-partie:

- Une partie gérant les personnages, en rouge
- Une partie gérant l'environnement, bleu
- Une classe représentant l'état global du jeu, en orange

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

Pour notre implémentation du jeu, nous avons prévu d'utiliser les quatre combattants suivant :

- Thork
- Flint
- Seku
- Kuro

Le constructeur Fighter permet d’avoir les quatre types de combattants en utilisant les différents identifiants qu’il a à sa disposition.

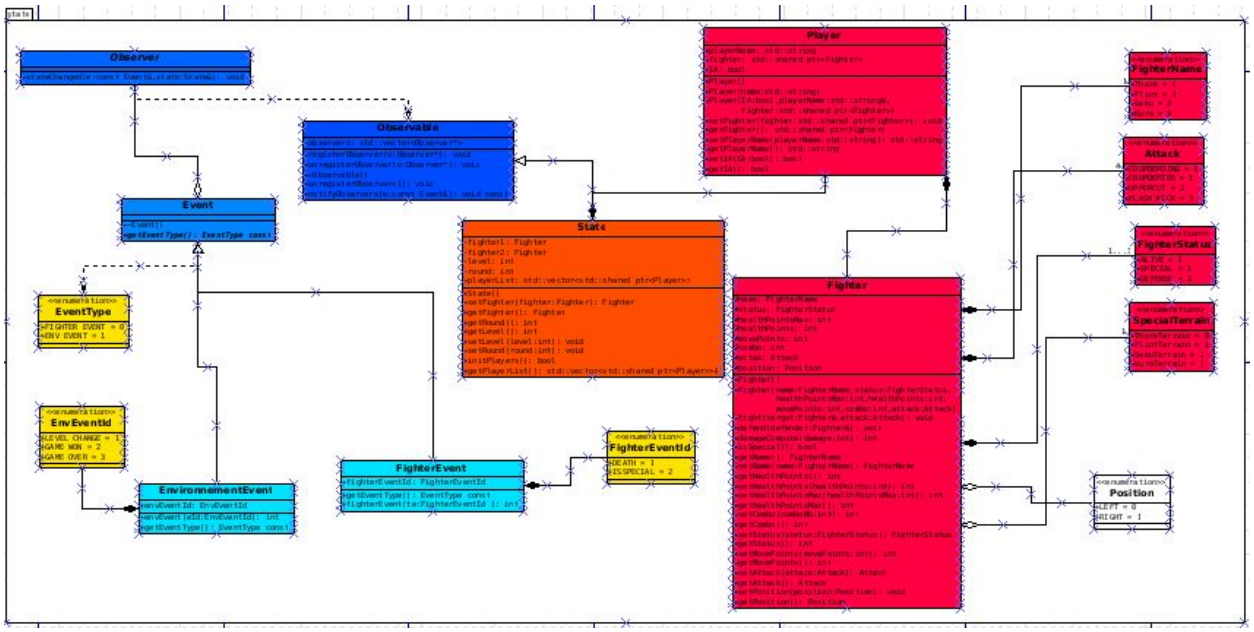
2.3 Conception logiciel : extension pour le rendu et moteur de jeu

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'Events:

Les FighterEvent qui représente un changement d'état lié au statut des combattants (En vie, mort, en mode super) .
Les environnement Event qui correspondent à des changement liés à l'environnement (changement de niveau) ou à des changements liées aux statistiques de la partie.

Illustration 1: 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 découpé notre affichage graphique sur deux niveaux, l'arrière plan qui représente le terrain de combat et les éléments mobiles à savoir les combattants (Thork, Flint, Seku, Kuro) qui se superposent sur la couche précédente. On transmet l'état du jeu ainsi que les textures des différents terrains et la texture des combattants à afficher.

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 combattants, seule le rendu des combattants 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. Dans le cas où le jeu est fini la fenêtre est automatiquement fermée.

3.2 Conception logiciel

Pour afficher un état, on utilise les trois classes suivantes: TileSet, TextureManager et StateLayer.

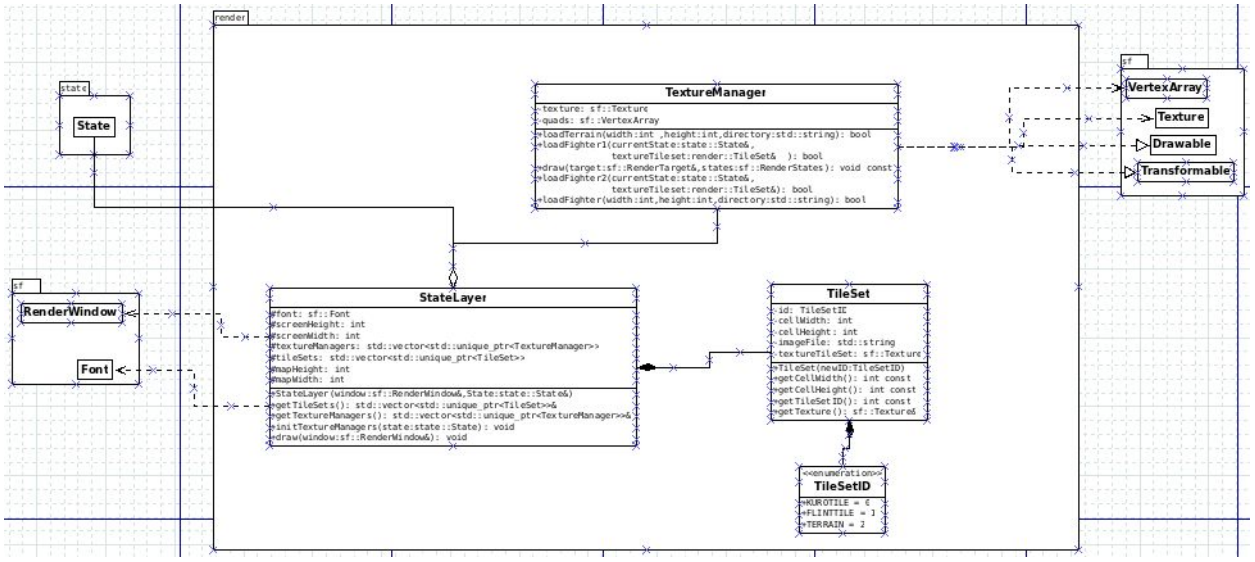
La classe TileSet permet de retrouver les fichiers correspondants au terrain à afficher ou aux "Fighter" à afficher. A partir de l'énumération TilesetID, la bonne texture est affichée.

La classe **TextureManager** permet d'afficher avec les textures au bon endroit avec les bonnes dimensions. Les méthodes **LoadFighter** ou **LoadTerrain** utilisent les dimensions et les chemins des textures pour les afficher.
La classe **draw** quant à elle se charge de associer les textures à la fenêtre d'affichage crée.

Pour finir**, StateLayer** récupère l'ensemble des TextureManagers et des TileSet puis les affiche en utilisant sa méthode draw.

3.5 Exemple de rendu

Illustration 2: 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 3: 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 4: Diagramme de classes pour la modularisation