



ECOLE NATIONALE SUPÉRIEURE DE
L'ELECTRONIQUE ET DE SES
APPLICATIONS

PROJET LATÉRAL TRANSVERSAL

Tales of Kornwal

OUAZZAGHTI Reda
et
ZOUHDI Zakaria

Projet de troisième année supervisé par
M. GRANIER et Prof. GOSSELIN

29 Septembre 2016

Table des matières

1	Objectif	2
1.1	Présentation générale	2
1.2	Règles du jeu	3
2	Description et conception des états du jeu	4
2.1	Description des états	4
2.2	Conception logicielle	5
2.3	Lien avec le rendu	5
3	Stratégie et conception de rendu	7
3.1	Stratégie de rendu	7
4	Règles de Changements d'Etats	9
4.1	Moteur de rendu	9
4.2	Changements extérieur	9
5	Stratégie d'intelligence artificielle	11
5.1	Stratégie de rendu	11
5.1.1	Stratégie d'IA simple : le zombie	11
5.1.2	Stratégie d'IA moyenne : le "groupe"	12

Chapitre 1

Objectif

1.1 Présentation générale



By David Revoy / Blender Foundation - Own work, CC BY 3.0

Tales of Kornwal est un jeu vidéo basé sur les mêmes règles de Fallout Tactics, *i.e* un jeu d'aventure doté d'un système de combat en tour par tour, permettant aux joueur de progresser et interagir avec un univers à l'allure originale à mi-chemin entre le médiéval-fantastique et le post-apocalyptique.

Les interactions seront basées sur un système de gestion d'inventaire et d'au moins une caractéristique, qui feront office de modificateurs lors d'actions entreprises par le personnage (*e.g* : la caractéristique "Force" influera grandement sur les dégâts infligés par un ennemi ou par le héros, ainsi que l'utilisation de telle ou telle arme).

1.2 Règles du jeu

Le jeu pourra posséder plusieurs aspects dépendant de l'étude du cahier des charges :

- Déplacement d'un personnage sur une "zone" de la mappemonde, accédant aux différentes cases nord-sud-est-ouest de la map en cliquant sur l'une des extrémités de l'écran.
- Système de combat tour par tour : lors d'une rencontre avec un ennemi, la map se vide de tous les sprites autres que le personnage joueur et ses adversaires, laissant donc place au duel entre le héros et l'ennemi. Le joueur commencera en premier (sauf modification) et disposera de deux choix possibles : se déplacer d'une case dans la zone, ou attaquer l'ennemi, faisant baisser son capital de points de vie. Le nombre de points de vie retirés dépendra de la caractéristique FORCE du personnage, ainsi que de son ARME équipée. Le tour se termine après que l'une des actions suivante a été effectuée, laissant le tour à l'ennemi (qui se déroulera de la même manière).
- Système d'interaction avec les personnages joueur ou non-joueur (discussions, interface d'échange d'objets)

Chapitre 2

Description et conception des états du jeu

2.1 Description des états

Un état du jeu est défini par les éléments suivants :

- **Les éléments fixes** (classe StaticElement). Elle est composée de deux classes filles : d'une part, une classe "Static", correspondant aux éléments immobiles permettant de définir les zones où les personnages peuvent se déplacer (*e.g* herbe, sable, neige...). Ces éléments peuvent être de type "CombatZone" si le personnage se trouve dans une zone d'influence ennemie, "NoCombatZone" s'il ne se trouve pas dans une telle zone, ou de type "ChangeMap" : lorsque le personnage se trouve dans cet espace, il bascule dans une nouvelle map, différente de celle où il était. D'autre part, une classe "Wall", définissant les zones infranchissables où le personnage ne peut se déplacer. IL peut s'agir d'un obstacles (arbre) ou tout simplement d'un bord de la map. (*e.g* mur).
- **Les personnages** (classe Character), éléments mobiles se déplaçant sur la grille (ou plus exactement sur les cases définies par les classes Space). Ces éléments peuvent être contrôlés soit par l'humain (Personnage joueur), soit par les IA (Personnages non joueurs). Ils possèdent tous une position définie par ses coordonnées X et Y, ainsi qu'une direction. Chaque personnage possède un certain nombre de données : niveau, expérience, force, points de vie etc .. Les personnages possèdent trois status possibles :
 - YOURTURN : Le tour du personnage. Il aura le droit de dépenser

des points de mouvements pour se déplacer sur la map, ou des points d'actions pour endommager les adversaires.

- HISTURN : Le tour des autres personnages. Le personnage doit rester immobile et subir les actions des autres personnages jusqu'à ce que son tour arrive.
- DEAD : Le personnage ne peut plus bouger, son tour n'arrivant jamais : il est mort.

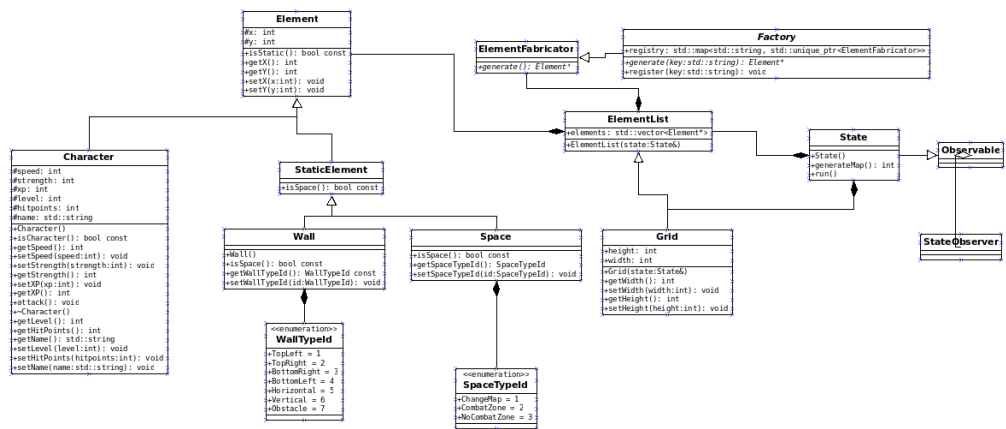
2.2 Conception logicielle

L'architecture du logiciel est composé des éléments suivants :

- une classe "Elements" : les classes "Character" et "Space" héritent toutes les deux d'une classe "Elements". La méthode de type *bool isStatic* permet de distinguer la classe d'éléments statiques de celle des éléments mobiles en renvoyant *true* si l'objet est de type "StaticElement", *false* sinon.
- des conteneurs d'éléments : la classe "ElementLists" contient une liste d'éléments fixes et mobiles. La classe "Grid" hérite de la classe "ElementList" afin de disposer ces éléments sur une map, ou "grille". Le tout correspond à un état donné, qui est un objet de la classe "State".
- une "Factory" ou fabrique d'éléments : il s'agit d'une interface permettant d'instancier une liste d'éléments sans avoir à spécifier la famille d'objets.

2.3 Lien avec le rendu

Il se fait via la classe "StateObserver", qui suit le pattern design "Observer". Cette classe "observe" les changements d'états et en avertit le client via le rendu. A terme, il faudra rajouter une interface afin de distinguer les différents événements (changement de map etc ..).



Chapitre 3

Stratégie et conception de rendu

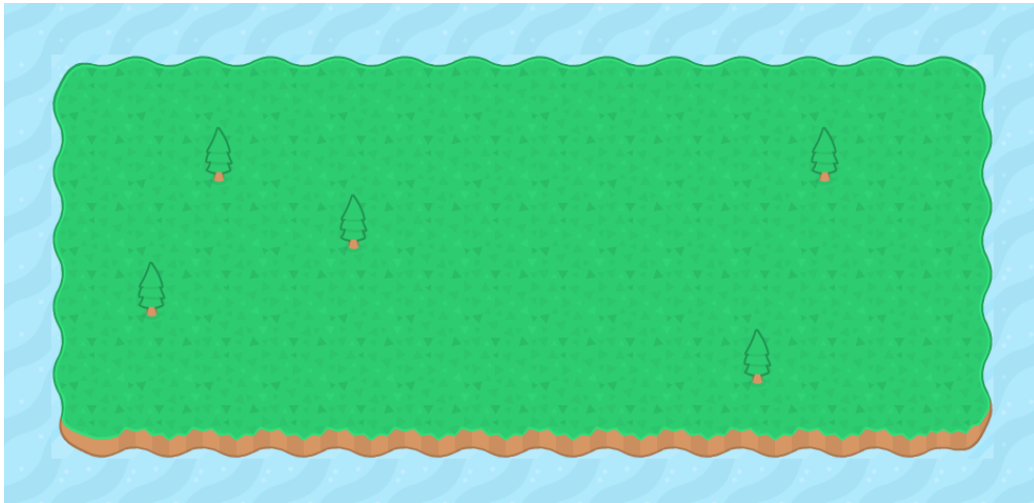
3.1 Stratégie de rendu

Le premier objectif a été de pouvoir générer une map à partir d'une tileset. Il s'agit d'une image constitué d'unités élémentaires de texture (tuiles), qui serviront de base pour la map. Nous écrivons ensuite un code qui attribue à chaque tuile un identifiant entier, puis qui stocke dans un tableau le numéro de chaque tuile que l'on veut voir apparaître. La position de la tuile dans le tableau détermine sa position dans la future map. Voici un exemple de



tileset :

Voici une map généré grâce à ce TileSet, obtenu en lançant le main :



On crée une classe `Tile` dans un package "rendu" ; deux classes héritent de cette classe `Tile`, les classes `Map` et `CharacterMap` (exploitée plus tard). De la même manière que le code mentionné avant, la classe `Map` stocke les coordonnées d'une tuile et la classe `Tile` stocke l'ensemble des tuiles pour un état donné. Il faudra donc associer à chaque instance d'élément une tuile, puis associer une liste d'éléments et une "Grid" à un état, puis générer la map correspondant à cet état (pour l'instant statiques).

Chapitre 4

Règles de Changements d'Etats

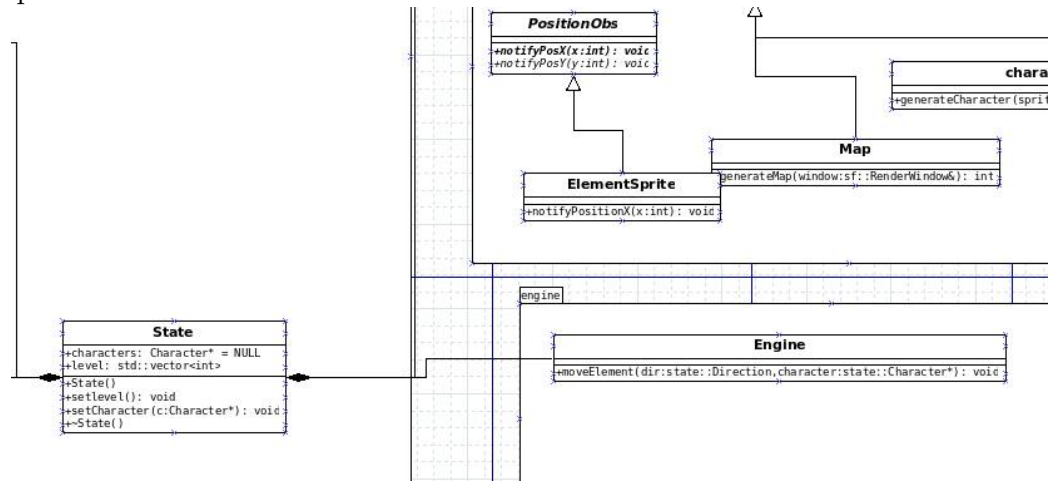
4.1 Moteur de rendu

Les Changements d'états se fait avec le moteur de rendu. Lors d'un mouvement (commandé par une touche du clavier), il y a un ordre géré par le moteur (package Engine) qui met à jour l'état instancié, qui à son tour met à jour la position des personnages, et le personnage est alors déplacé vers la zone indiquée. Le personnage peut se déplacer à droite, à gauche, vers le haut ou vers le bas. Pour cela, chaque objet "Character" dispose d'une énumération "Direction". Pour permettre le déplacement, la classe Engine dispose d'une méthode "moveCharacter()", qui prend en arguments un pointeur de Character et une direction. Selon la direction entrée, la fonction modifie les coordonnées X et Y via par exemple les méthodes "getX()" (récupération de la position X actuelle) et "setX" de l'élément pour la coordonnée X. Afin que le changement de position du personnage soit aussi appliqué sur le sprite, on met en place un observable "PositionObs", qui dispose de deux méthodes "notifyPosX()" et "notifyPosY()" pour synchroniser le changement de position du personnage à celui du sprite. On n'a plus qu'à appliquer la méthode moveCharacter dans le main, dans la boucle pollEvent. Enfin, on évite les sorties de map via de simples conditions if.

4.2 Changements extérieur

Le constructeur de State, contenue dans le package du même nom, lui permet d'initialiser l'attribut level de la classe, chargeant ainsi un niveau. Le second attribut de type Element* permet d'accéder directement à l'élément et par conséquent, à le modifier. Une amélioration permettra d'accéder à une liste d'éléments, afin d'instancier et mettre à jour plusieurs sprites en même

temps.



Chapitre 5

Stratégie d'intelligence artificielle

5.1 Stratégie de rendu

Dans le cadre d'un jeu de rôle, les différents types personnages adversaires suivent chacun un comportement distinct, en fonction de leur "intelligence" au sein de l'univers. Aussi, un zombie aura le comportement prévisible d'errer sur la map jusqu'à croiser un personnage, et de l'attaquer jusqu'à ce que mort s'en suive. Un goblin, sensé être frêle et sournois, aura tendance à adopter un comportement plus complexe en commençant d'abord par chercher les gobelins les plus proches, s'unir, et se déplacer au hasard en bande... Avant de lâchement battre en retraite s'il se retrouvent en situation de 1 contre 1. Enfin, un orque rodé aux méthodes d'embuscades tentera de traquer le personnage en prenant le chemin le plus court menant à lui, mais en ayant pour priorité de joindre ses forces aux autres orques de la map avant d'attaquer le personnage.

En terme d'implantation, toutes les IA sont capables de jouer le jeu et les différentes entités. Il sera même également possible d'implanter des personnages à des niveaux d'intelligence variés, l'entrée des méthodes de la classe IA comprenant un argument "State", donnant accès à la liste des personnages.

L'ajout d'un sleep (fonction de la bibliothèque `unistd`) pour une durée d'une seconde permet d'observer le comportement du personnage en détail.

5.1.1 Stratégie d'IA simple : le zombie

L'intelligence artificielle des zombies est analogue à celle des canons fantastiques : peu intelligent, vagabondant de manière aléatoire jusqu'à croiser

5.1.2 Stratégie d'IA moyenne : le "groupe"

```
sequenceDiagram
    participant Client
    participant IA
    Client->>IA: executeDumb(moteur:engine::Engine,etat:state::State): state::Direction
    activate IA
    IA->>Client: state::Direction
    deactivate IA
    Client->>IA: executeGroupAI(moteur:engine::Engine,group:std::vector<state::Character*>): state::Direction
    activate IA
    IA->>Client: state::Direction
    deactivate IA
```