Projet Logiciel Transversal

Fodus

Table des Matières

Table des Illustrations

I Objectif

I.A Présentation générale

L'objectif de ce projet est la réalisation d'un jeu type « tactical RPG » (système de combat similaire à Dofus).

Figure 1 : Dofus

Dofus est doté de 2 mécanismes de jeu : la phase d'exploration, et la phase de combat. Nous allons nous inspire de la phase de combat, avec une phase d'exploration propre à notre jeu et son univers.

I.B Règles du jeu

Contexte

Le joueur incarne le stagiaire du nécromancien d'un donjon : il lui a confié la tâche de nettoyer les salles du donjon après le passage d'aventuriers.

Le joueur va donc affronter des monstres survivants, ainsi que potentiellement des aventuriers perdus. Mais étant un stagiaire, il ne peut combattre directement et il doit invoquer des serviteurs mort-vivants pour combattre à sa place ; Il peut également corrompre les monstres et les aventuriers, qui changeront de camp et combattrons pour lui : mais certains aventuriers pourront également corrompre des monstres pour les rallier à leur camp.

Le jeu sera constitué d'une première phase dite d'exploration. Dans cette phase du jeu, le joueur sélectionne une zone de jeu sur laquelle combattre.

Lorsqu'une zone de combat est choisie, le jeu passe en mode combat : il s'agit là d'un combat au tour par tour, où le joueur affronte les monstres et/ou les aventuriers de la zone. Le joueur se déplace comme tout autre personnage (monstres et aventuriers), mais ne peut les attaquer (mais peut se faire attaquer) : il ne peut que corrompre les autres personnages pour les rallier à son camp et invoquer des minions pour l'aider. Un combat est gagné quand tous les monstres sont vaincus, un combat est perdu quand le personnage principal du joueur est mort.

Le jeu possède un système d'expérience où le joueur sera récompensé à la fin de ses combats victorieux ; il pourra gagner de niveau après un certain nombre de points d'expériences, et se verra gratifier d'amélioration de caractéristiques et de compétences.

I.C Conception Logiciel

II Description et conception des états

II.A Description des états

Le jeu se divise en deux grandes parties : la partie exploration et la partie combat. La partie exploration consiste en une carte générale du donjon où l'on peut se déplacer de section en section. Une suite de mission demande au personnage de se rendre successivement dans différentes zones indiquées. Une fois arrivé dans ces zones le combat se lance, et l'on passe à la seconde phase du jeu.

Le combat se passe dans une zone divisée en cases sur laquelle se trouvent le personnage principal et un certain nombre d'ennemis. L'objectif de chaque combat est de tuer tous les ennemis présents, tandis qu'une défaite à lieu si le personnage du joueur meurt. Les commandes se font au tour par tour, d'abord le personnage du joueur, puis les ennemis. Chaque tour le joueur peut effectuer un déplacement puis une action, de même pour chaque adversaire.

Partie combat :

Etat :

Représente l'état du combat à un instant précis. On y trouve ainsi une liste de tous les éléments du combat, que ce soit des éléments fixes comme des cases ou des personnages mobiles. On y trouve également la taille de la zone de combat.

Les éléments "case" et "personnage" possèdent tous les deux les attributs "posX" et "posY" représentant leur position sur la grille de jeu.

Les cases :

Les différentes cases constituent la zone de jeu. Chaque case possède des attributs indiquant si elle est : un mur ou une case libre, un attribut indiquant si elle est occupée ou non, et d'un attribut indiquant si elle est piégée.

Elles possèdent également l'attribut "départ", qui correspond au fait que la case soit ou non une case de départ pour le personnage ou un ennemi.

Les personnages :

Regroupe tous les personnages du jeu, que ce soient les zombies, le personnage principal ou les ennemis. Tous ces personnages ont les mêmes propriétés, qui sont :

- "santeMax", représentant le total de points de vie.
- "sante", représentant la somme actuelle de points de vie.
- "deplacement", correspondant au nombre de case dont le personnage peut se déplacer chaque tour.
- "classe", représentant pour les aventuriers la classe à laquelle ils appartiennent. Les classes possibles sont : Guerrier, Magicien, Prêtre, Archer.

- “niveau”, représente le niveau et donc la puissance globale du personnage.
- “status”, qui correspond à l’état du personnage : bien, poison, étourdi, ralenti, ou une combinaison quelconque.
- “corruption”, qui indique combien de fois le personnage à été corrompu.
- “competences”, qui liste les compétences auxquelles le personnage a accès.
- “limiteZombie” qui indique quel est le nombre maximal de Morts-vivants pouvant être invoqué à la fois par le personnage.
- “defense”, “puissance” : représentant des caractéristiques du personnage influençant respectivement les dégâts reçus et infligés.
- “effets” liste les effets qui affectent en permanence le personnage.
- “direction” indique dans quelle direction le personnage est dirigé

Les compétences : utilisables par les personnages, les compétences possèdent un certain nombre d’attributs les définissant :

- “degats” correspond au dégâts infligés par la compétence (ou soin)
- “postee” indique à combien de case peut être utilisées la compétence
- “zone” représente la zone qui sera affectée par la compétence (0 pour une case unique par exemple)
- “cible” représente quel type de cible peut être affecté par la compétence : les ennemis, les alliés ou les deux.
- “effets” liste les effets supplémentaires de la compétence

Partie exploration :

Carte :

Représente la carte du jeu dans cette phase. Le personnage s'y déplace pour se rendre dans les différentes zones, ainsi que des groupes d'aventuriers qui peuvent être affrontés par le joueur. Cet élément liste les différentes zones du jeu, les différents groupes d'aventuriers et le personnage.

Zone :

Correspond aux différentes zones où le joueur peut se rendre. Chacune d'entre elle peut potentiellement donner lieu à un combat contre les monstres du donjon, ou contre des

aventuriers si un groupe se trouve sur la même zone que le joueur. Chaque zone possède plusieurs attributs :

- "occupation" correspond au fait que la zone est occupée ou non, que ce soit pas un groupe d'aventurier ou par le joueur.
- "difficulte" représente la puissance des monstres présents dans la zone.
- "position" correspond à la position de la zone sur la carte.

Personnage :

Ne concerne ici que le personnage joueur. Ces différents attributs sont :

- “sante” qui est la sante totale du personnage
- “defense”, “puissance” sont des caractéristiques du personnage comme précédemment
- “Déplacement" exprime le nombre de case que peut parcourir le personnage en combat.
- “niveau” correspondant au niveau du personnage, et donc à sa puissance totale
- “expérience” correspond au nombre de points d’expérience engrangé par le personnage et permettant de passer au niveau supérieur.
- “posX” et “posY” représentent les coordonnées du personnage sur la carte.
- “inventaire” correspond à l’inventaire du personnage
- "emplacement" représente l'endroit où se trouve le personnage dans la carte.

Inventaire :

Regroupe tous les objets que le personnage a pu obtenir et ceux qui sont en cours d’utilisation.

- “equipe” liste tous les objets actuellement équipés sur le personnage
- “sac” liste tous les objets en possession du joueur.
- “limite” fixe une limite au nombre d’objet transportables par le joueur
- “or” correspond à l’argent accumulé par le joueur au cours du jeu.

Objet : représente tous les objets pouvant être utilisés par le joueur. Ils sont définis selon les attributs suivants :

- “type” représente la catégorie de l’objet, et donc l’emplacement où il sera équipé.
- “modificateur” représente l’efficacité de l’objet, et donc son influence sur les caractéristiques du personnage.
- “prix” représente la quantité d’or que le personnage peut obtenir en revendant l’objet.

Aventuriers :

Chacun de ces objets correspond à un groupe d'aventuriers se déplaçant dans le donjon et risquant de rencontrer le personnage et donc de l'affronter. Ils sont visibles sur la carte par le joueur, et sont décrits de la manière suivante :

- "niveau" représente le niveau global des aventuriers du groupe, et donc de la difficulté du combat en cas d'affrontement.
- "emplacement" correspond à la zone dans laquelle se trouve le groupe à un instant donné.

II.B Conception logiciel

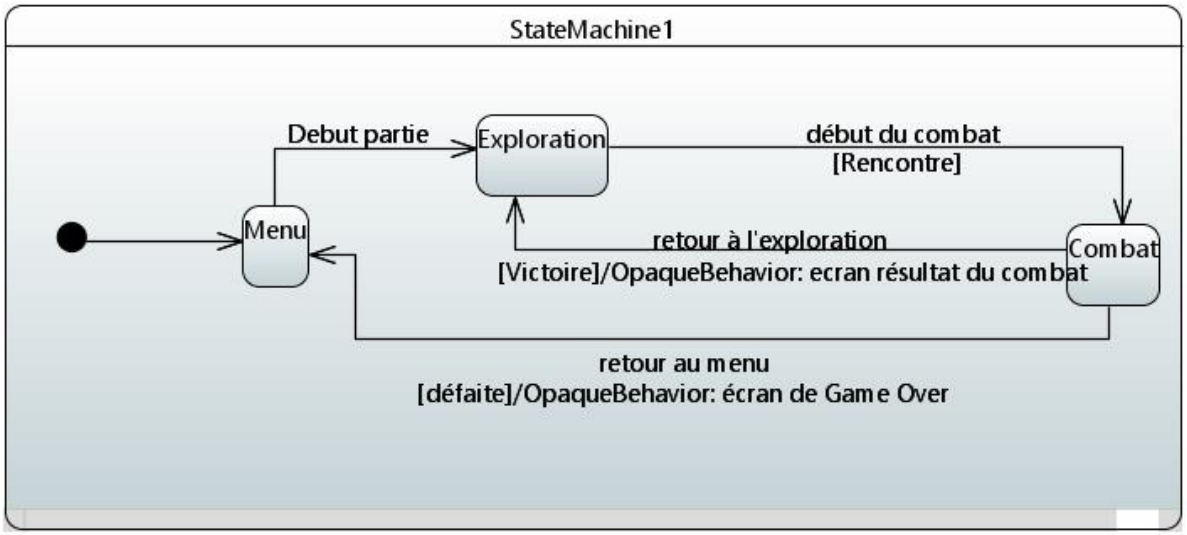

Figure 2 : Machine à état décrivant le déroulement du jeu

Représentation du workflow du jeu et des transitions entre différentes phases.

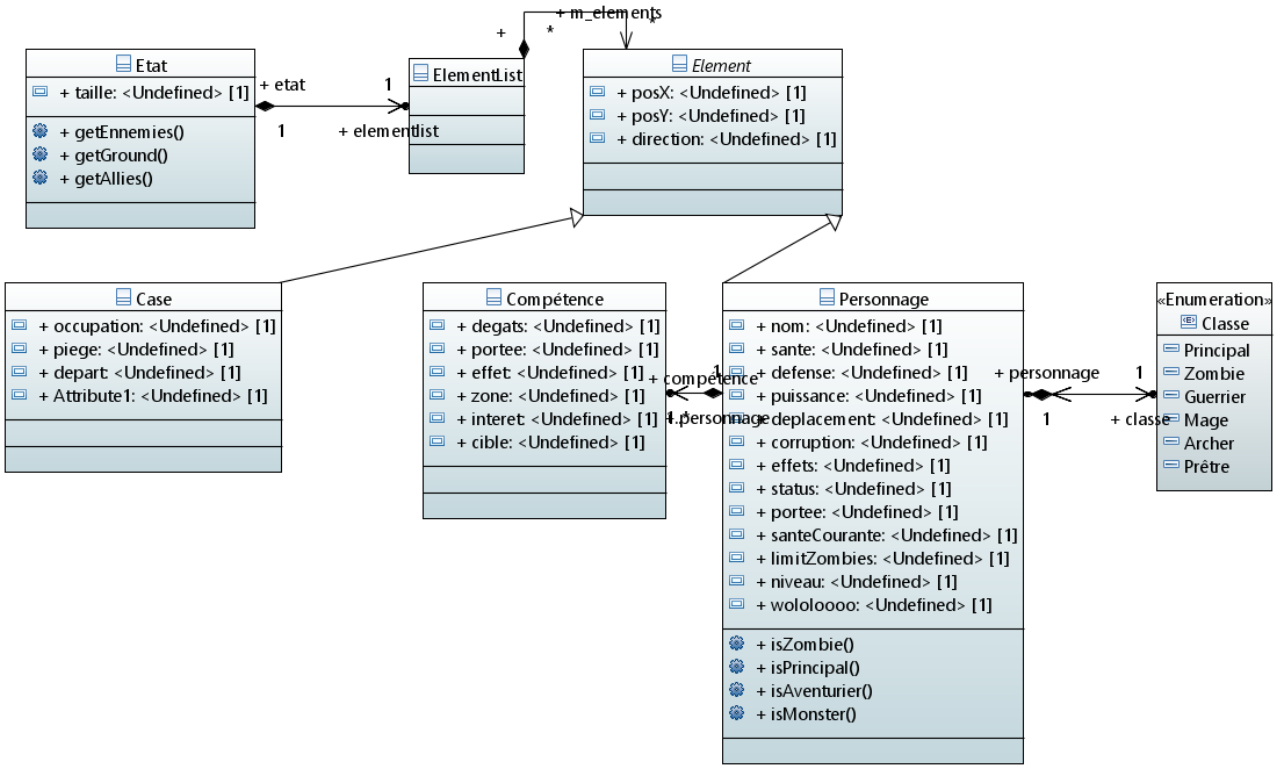

Figure 3 : Diagramme de classe d'état pour la phase de combat

Représentation des différentes classes utilisées lors de la phase de combat et des différents attributs de celles-ci, ainsi que des différents liens existants entre elles.

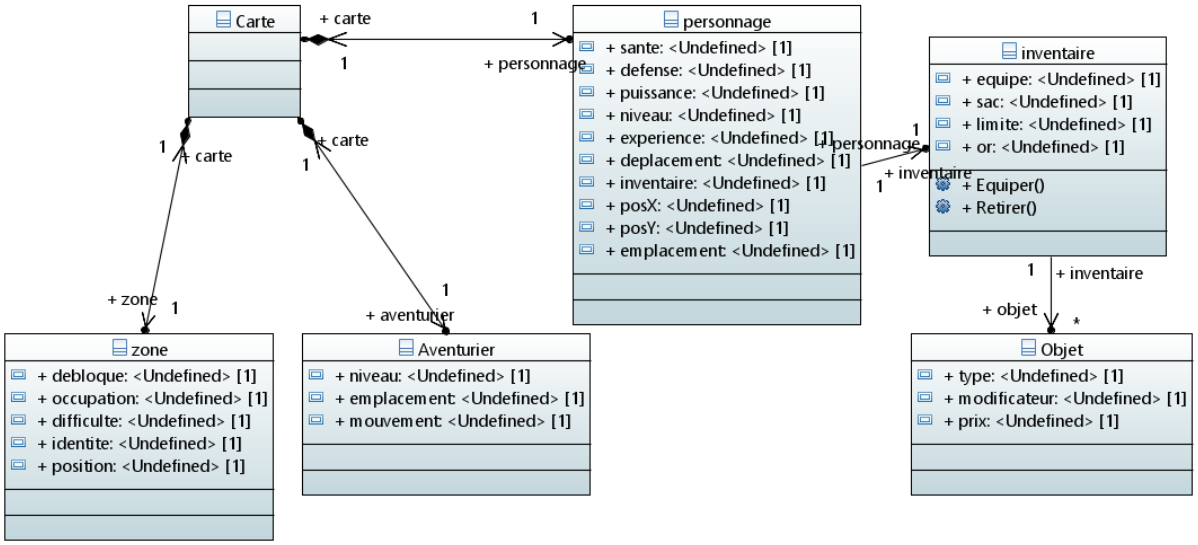

Figure 4 : Diagramme de classe d'état pour la phase d'exploration

Représentation des différentes classes utilisées lors de la phase d'exploration ainsi que leurs attributs et les liens entre elles.

II.C Conception logiciel : extension pour le rendu

II.D Conception logiciel : extension pour le moteur de jeu

II.E Ressources

L'ensemble des ressources (autre que les fichiers images et sons) est codé en JSON, ce qui inclus le fichier de déclaration des tiles, et les fichiers de niveau (ainsi que tout fichier de configuration). Celle permet de n'avoir aucun élément (ou presque) hardcodé dans le code du jeu, et d'offrir une souplesse de développement.

Exemple : fichier de Tiles, qui permet de générer les tiles en fonction des éléments

```
 "header": {
  "version": "0.2",
  "name": "Default Tileset"
 },
 "tiles": {
  "UNDEFINED": {
   "y": 0,
   "x": 0
  },
  "VOID": {
   "y": 0,
   "x": 2
}
```

Exemple : Fichier d'un niveau

```
{
  "header": {
    "version": "0.0",
    "name": "My Little Level",
    "desc": "Misc"
  },
  "level": [
    [
      [
        {
```

```
            "key": "WALL_DUNGEON_TOP_LEFT"
        },
        {
            "key": "VOID"
        }
    ],
    [
        {
            "key": "WALL_DUNGEON_TOP_MID"
        },
        {
            "key": "VOID"
        }
    ],
    ...
],

[...],

...

]

}
```


III Rendu : Stratégie et Conception

III.A Stratégie de rendu d'un état

Pour rendre un état, nous avons regroupé tout le rendu en Scène, qui est l'image de l'état du jeu, et nous l'avons découpée en plans. Ces plans se répartissent différents types de données : un plan pour les informations, 2 plans pour les élément fixes (2 niveaux de profondeur), un pour les éléments mobiles (personnages), et un plan de debug. Chaque plan contient une texture et une matrice donnant la position des éléments dans la fenêtre de jeu, et les coordonnées de sa partie de la texture. Ces 2 éléments sont envoyés à la carte graphique ce qui permet un affichage rapide et optimisé des éléments (plutôt que de tracer sprite par sprite)

Actuellement, seuls les éléments fixes sont été implémentés, pas les mobiles ni les informations.

III.B Conception logiciel

Le diagramme des classes pour le rendu est en [Figure 5 : Diagramme de classe du moteur de rendu].

Utilisant la bibliothèque SFML, nous avons choisi d'implémenter l'interface sf ::drawable à nos objets à rendre : scene et ses layers. Cette implémentation permet de simplifier les appels graphiques, puisqu'il nous suffira de faire un App.draw(scene) dans la boucle principale pour rendre l'intégralité des plans.

Layer

Un layer (plan) est un objet réalisant le lien entre une liste d'éléments et leur position spatiale dans la fenêtre ainsi que leur représentation (texture). Layer hérite de sf ::drawable : un appel à App.draw(layer) permet de dessiner tout le layer. Dans le cas de l'ElementLayer (fixe ou mobile), le layer possède une propriété de profondeur (pour gérer plusieurs couches de tiles, ex. des décorations sur un mur), et une TileFactory, qui s'occupe de gérer la correspondance Element – Texture. Une fonction update() est déclenchée par l'Observer de l'état du jeu pour mettre à jour les éléments du plan.

Scene

Elle contient tous les plans, et son implémentation de draw() appelle les draw() de ses layers.

Elle possède une fonction update(ElementList), qui est appelée lors d'une modification de l'état (pattern Observer), et qui de façon analogue à draw() appelle les update(ElementList) de ses layers.

TileFactory

Cette Factory (pattern Factory donc) génère une tile à partir d'une clef d'identification. Ces clefs sont les identifiants des éléments (ex. FLOOR_WOOD_1), ce qui permet de trouver l'emplacement de leur texture via le chargement préalable du fichier tiles.json. Ce fichier est l'index de toutes les clefs d'éléments avec leurs coordonnées de texture associés : il n'y a pas de données en dur dans le code.

Tile

Cet objet est simplement une représentation de la clef et les informations de texture associée.

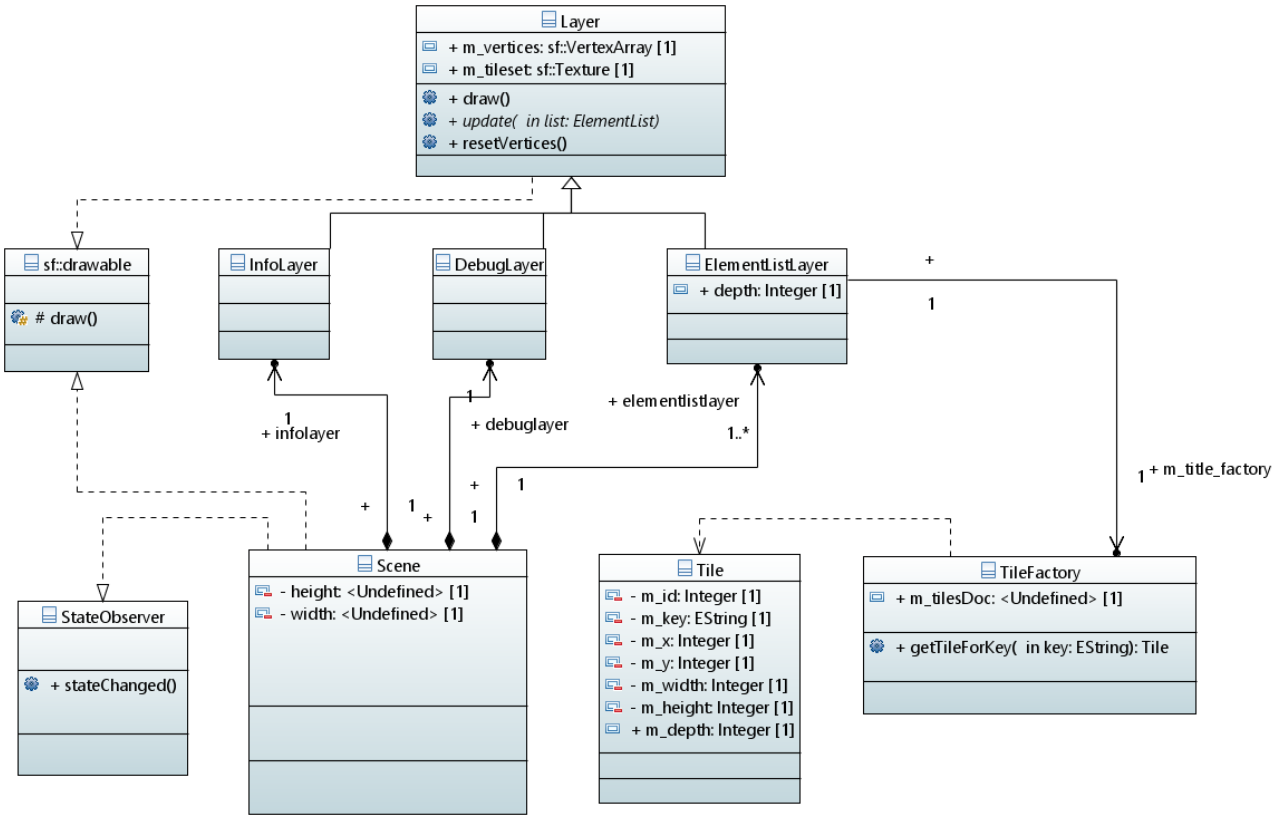

Figure 5 : Diagramme de classe du moteur de rendu

III.C Conception logiciel : extension pour les animations

III.D Ressources

Afin de gérer les ressources JSON, 2 programmes ont été réalisés : un script Python 3 permettant d'ajouter des tiles à l'index, et un éditeur graphique de niveau.

III.E Conception logiciel : éditeur

L'éditeur est pleinement intégré au jeu et peut être choisi au démarrage du programme.

Il contient 2 fenêtres : celle de droite permet de sélectionner une tile à appliquer, et celle à gauche est une vue du niveau en cours de modification, et est géré par une unique classe Editeur qui regroupe l'ensemble des fonctions, et ne déborde donc pas dans les fichiers « classiques » du reste du jeu.

Il contient la liste des éléments du niveau en cours (chargée depuis son JSON), la liste des tiles possibles (chargées depuis l'index des tiles), et permet de modifier les tiles du niveau (une par une ou plusieurs par drag and drop). La sauvegarde est faite automatiquement à la sortie de l'éditeur.

III.F Exemple de rendu

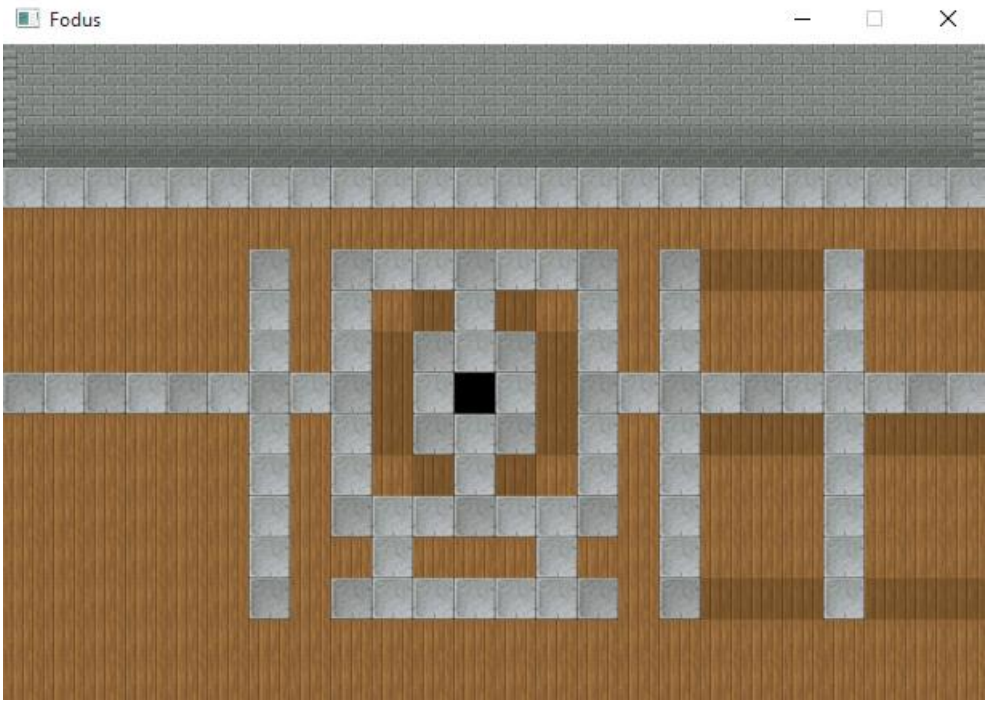

Figure 6 : Exemple de rendu

IV Règles de changement d'états et moteur de jeu

IV.A Horloge globale

Le jeu est un jeu à états finis : le jeu est rythmé par une horloge globale qui cadence l'exécutions des actions et donc les mises à jour de l'état.

IV.B Changements extérieurs

Les changements extérieurs sont des commandes exécutées par le moteur de jeu et provenant d'entrées utilisateur :

Commandes globales :

- Mode de démarrage : Editeur ou jeu
- Nouvelle partie, charger niveau, quitter, …

Commandes d'action :

- Déplacement
- Attaque
- Magie
- …

IV.C Changements autonomes

Les changements autonomes sont un jeu de commandes automatiques exécutées lors de la mise à jour de l'état. Elles ne requièrent pas de commande d'entrée.

Il s'agit typiquement :

- Conditions de défaite : le personnage joueur n'a plus de vie (santé = 0)
- Vérifier la santé d'un personnage : si la santé est à 0, la détruire
- Conditions de victoire : il n'y a plus d'ennemis
- Appliquer la liste d'actions en attente

IV.DConception logiciel

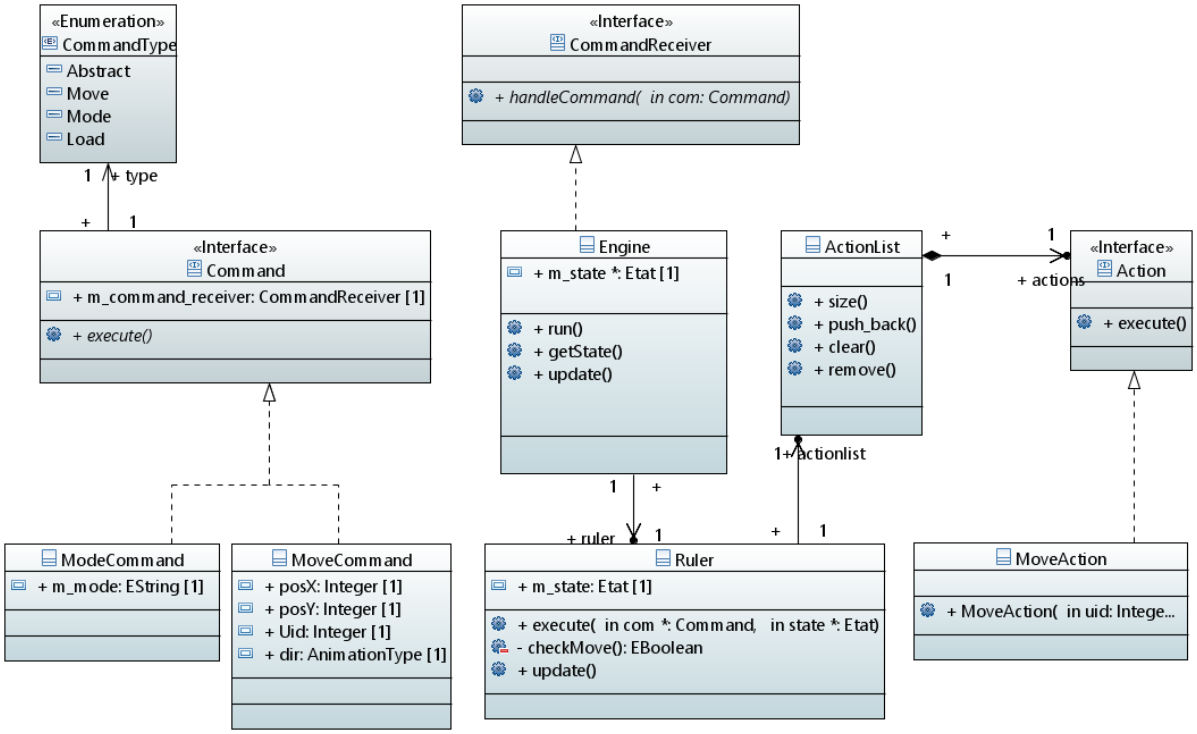

Figure 7 : Diagramme des classes du moteur de jeu

Le moteur du jeu repose sur le Pattern Command et sur l'exécutions des commandes en différé, grâce à leur stockage dans une liste de commandes. *Attention, pour cette première version, toute commande est directement exécutée par le moteur et non mise dans la liste de commande. Par conséquent, le moteur de jeu n'obéit pas encore à une mise à jour cadencée par une horloge temporelle, mais par évènements. Le diagramme UML n'inclus donc pas la CommandList.*

Classes Command :

Interface de commande suivant le Pattern Command. Elle implémente un attribut (objet de classe Command Receiver), un attribut CommandType (enum) et une méthode abstraite execute(), qui appelle la fonction handleCommand(Command*) de l'objet Receiver.

Les différentes implémentations (ModeCommand, MoveCommand,…) sont les différents types de commandes, avec leurs attributs associées.

Exemples :

MoveCommand : position X, position Y, UID (identifiant de l'objet se déplaçant), direction

ModeCommand : type de mode (editor / game)

Classe Command Receiver :

Interface de receveur suivant le Pattern Command. Elle implémente une méthode abstraite handleCommand(Command*) qui est appelé pour le traitement des commandes.

Classe Engine :

Classe au coeur du moteur du jeu. Son rôle est de stocker les commandes dans la CommandList qui seront exécutées lorsqu'une mise à jour de l'état sera lancée (ie quand update() sera appelé à un rythme régulier). *Ce fonctionnement est encore non implémenté dans cette première version : pour l'instant lorsque le moteur reçoit une commande, il l'exécute aussitôt.*

Les commandes donnant à des actions en jeu sont envoyées au Ruler pour traitement, tandis que les autres peuvent donner lieu à des actions directes dans le moteur (ex. pause, quitter, chargement…) et ne sont pas soumises aux règles du jeu.

Classe Ruler :

Cette classe vérifie les commandes qu'elle reçoit et en réalise un dynamic cast en fonction de leur type. Elle vérifie alors la faisabilité de la commande, et le cas échéant crée une Action qui sera appliqué sur l'état pour le modifier. C'est également elle qui gère les commandes autonomes.

Classe Action :

Cette interface représente une action validée par le Ruler, issues d'une commande externe ou de changements autonomes.

Exemple :

MoveAction : position X, position Y, UID (identifiant de l'objet se déplaçant), direction

IV.E Conception logiciel : extension pour l'IA

IV.F Conception logiciel : extension pour la parallélisation

V Intelligence Artificielle

V.A Stratégies

V.A.1 Intelligence minimale

V.A.2 Intelligence basée sur des heuristiques

VI.A.1 Logger

Afin d'avoir un débogage plus efficace et une meilleure traçabilité des bugs et exécutions, le logiciel implémente la Library EasyLogging++, qui met à dispositions des macros permettant d'afficher dans la console et simultanément d'enregistrer dans le fichier global.log (même dossier que l'exécutable).

VI.A.2 Amorceur du logiciel

Afin d'obtenir une organisation robuste et efficace, une classe d'amorce Bootstrap a été conçue. Lors de l'exécution du main(), celle-ci instancie le Bootstrap, l'initialise avec la fonction start() puis la lance avec run().

Cette fonction run() permet de choisir le type de mode à lancer : éditeur ou jeu, ce qui instancie un objet Game ou Editor selon le choix et le lance.

Le Bootstrap réalise une couche d'abstraction entre les autres classes et par exemple les fichiers de configurations et de ressources, ceci afin de découpler au maximum toutes les classes du programme. Les classes ne savent même pas qu'ils manipulent des fichiers et ont directement accès au contenu (parsé si fichier JSON) ou au lien du fichier ressource : aucune configuration n'est écrite en dur dans le code. *Sauf pour les sprites qui n'utilisent pas encore le Bootstrap dans cette version.*

Exemple :

Game : chargement du niveau « test » : appelle de bootstrap ::loadLevel(« test ») => Bootstrap : chargement depuis le disque de « res/levels/ »+ «test»+ « .json » , puis

parsing du document JSON chargé, stockage dans une std ::map [« test »] et renvoie un shared_ptr sur ce membre de la map .

VI.A.3 Répartition sur différents threads

VI.A.4 Répartition sur différentes machines

VI.B

VI.C Conception logiciel

VI.D Conception logiciel : extension réseau

VI.E Conception logiciel : client Android