Option IS

Rapport PLT

Final fantastique

FALLOURD Yohann - VAN ESPEN Jérémie

Table des matières

1 Objectif **3**
- 1.1 Présentation générale . . . 3
- 1.2 Règles du jeu . . . 4
- 1.3 Conception Logiciel . . . 4

2 Description et conception des états **6**
- 2.1 Description des états . . . 6
  - 2.1.1 Etat éléments vivants . . . 6
  - 2.1.2 Etat éléments non vivants . . . 6
  - 2.1.3 Situation du joueur . . . 7
- 2.2 Conception logiciel . . . 8

3 Rendu : Stratégie et Conception **9**
- 3.1 Stratégie de rendu d'un état . . . 9
- 3.2 Conception logiciel . . . 9

4 Règle de changement d'état **10**
- 4.1 Conception logiciel . . . 10

5 IA **11**
- 5.1 Comportement . . . 11
- 5.2 Intelligence artificielle haut niveau . . . 12

6 Engine et modularisation **14**
- 6.1 Stockage de commandes . . . 14
- 6.2 Exécution des commandes . . . 14

7 Server : conception de l'API **15**
- 7.1 API REST . . . 15
  - 7.1.1 Envoi des commandes au serveur . . . 15
  - 7.1.2 Récupération des commandes . . . 15
  - 7.1.3 Ajout/modification d'utilisateur . . . 16

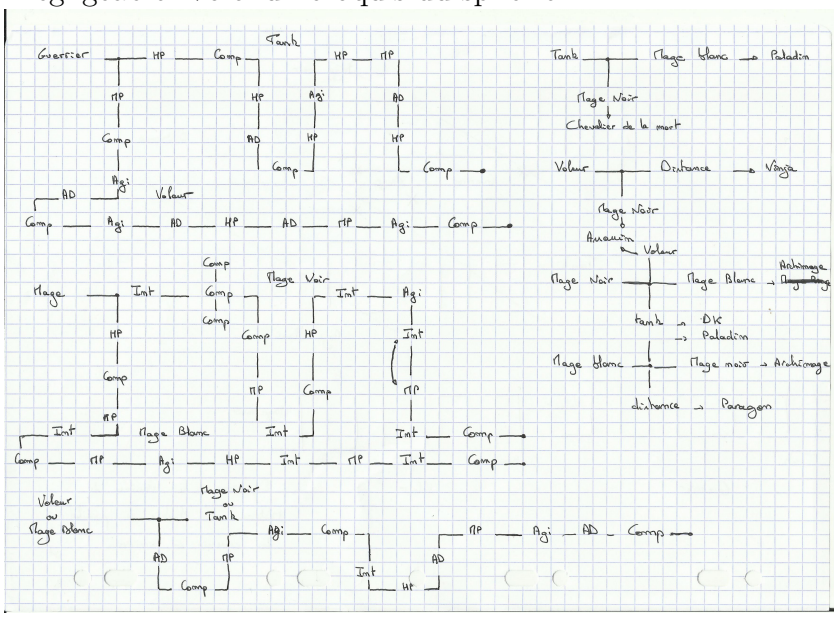
7.1.4 Demande d'utilisateur 16
7.1.5 Demande de tous les utilisateurs 16
7.1.6 Suppression d'un utilisateur 17

Chapitre 1

Objectif

1.1 Présentation générale

Le jeu est basé sur l'archétype de Final Fantasy X, se distinguant par un système de combat au tour par tour ainsi que son emblématique "sphérier" correspondant à un arbre de talent apportant une liberté de personnalisation non négligeable. Voici un croquis du sphérier :



A la différence du véritable jeu, il n'y aura pas de déplacement libre en temps réel ; le gameplay se résume à un déplacement de points d'intérêts en points d'intérêts, poussé par une histoire typique de cet archétype, où se déroulent différents évènements/combats.

1.2 Règles du jeu

Le coeur du jeu sont les deux personnages auquel le joueur (si il joue seul) a accès. Chacun commence avec une spécialisation particulière de mage et de guerrier choisissant directement de se diriger vers différentes spécialisations. Via une carte du monde, le joueur avance de point en point et déclenche différentes rencontres. Les combats se déroulent en tour par tour et si l'un des joueurs meurt, il revient à la vie avec 1 point de vie. La mort des deux joueurs entraine un retour au dernier point de sauvegarde automatique. Gagner un combat donne de l'expérience, qui donne des niveaux, qui permettent de progresser dans le sphérier.

Il existe également différents objets permettant de rendre de la vie ou d'avoir d'autres effets. De plus, un système d'équipement existe pour rendre le joueur plus fort au cours de l'aventure. En effet, un personnage possède des caractéristiques (Force, Agilité, Intelligence, Points de vie (HP) et Points de magie (MP)) qu'il peut améliorer.

Le jeu se finit lorsque l'aventure est terminée !

1.3 Conception Logiciel

Voici les packages de notre projet :

Package state. Package central qui gère l'état du jeu.

Package engine. Package qui modifie l'état de jeu et stocke les commandes.

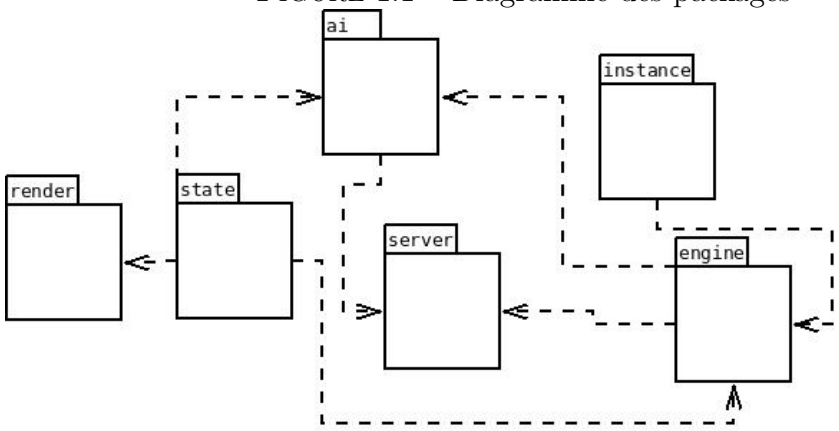
Package ai. Package qui gère le contrôle par l'ordinateur des monstres et, potentiellement, un personnage principal.

Package server. Package contenant la gestion de l'API du jeu, que ce soit par réseau ou localement.

Package instance. Package contenant l'Interface Homme-Machine.

Package render. Package contenant la gestion du rendu.

FIGURE 1.1 – Diagramme des packages



Chapitre 2

Description et conception des états

2.1 Description des états

Un état du jeu est formé par un ensemble d'éléments vivants et non vivants ainsi que la situation dont se trouve le joueur.

2.1.1 Etat éléments vivants

Les éléments vivants sont des entités ayant tous des caractéristiques propres : santé max, santé actuelle, mana max, mana actuel, force, agilité et intelligence. On distingue deux types d'éléments vivants :

Character. Ce sont les héros du jeu. Ils pourront s'équiper d'une arme et d'une protection ajoutant des bonus d'attaque et de défence. Ils évolueront grâce à un système de gain d'expérience et de personalisation du joueur.

Monster. Comme le nom l'indique, ce sont les monstres du jeu. Ils ne peuvent pas gagner d'expérience et leurs caractéristiques sont générées avec le niveau actuel des héros. Leurs compétences seront fixées suivant le type du monstre (élémentaire, boss ect...). Ils ne portent pas d'équipement. Seul les boss auront une capacité spéciale (comme les héros) appelé "Overdrive".

2.1.2 Etat éléments non vivants

Les éléments non vivants sont au nombre de trois et ne portent aucune caractéristiques communes.

Item. Ce sont les objets utilisables par les héros. Ils peuvent changer leurs attributs (augmentation d'une caractéristique ect...).

SphereGrid. C'est une table des compétences. Chaque niveaux supplémentaires permettra au character d'apprendre de nouvelles compétences et d'augmenter ces caractéristiques. Il sera possible au joueur de choisir la personnalisation de son personnage car plusieurs table sont possible au cours du jeu.

Node. Ce sont les points clefs du jeu. Les héros pourront se déplacer sur une carte de noeud en noeud. Chaque noeud comporte des évènements aléatoires et non aléatoires. Les éléments aléatoires sont des combats contre des monstres aléatoires tandis que les non aléatoires sont des élèments de l'histoire. Cela peut être un simple dialogue ou un combat contre un boss. Il est possible uniquement de ce déplacer au noeud suivant ou au noeud précédent. Si ce noeud a déjà été visité, l'évènement non aléatoire n'aura plus lieu.

2.1.3 Situation du joueur

Les situations possibles dans lequelles se trouve le joueur sont au nombre de quatre. Elles représentent la ligne directive du jeu.

Déplacement sur un noeud. Le joueur se déplace dans un nouvel endroit qui va générer des évènements.

Evènement aléatoire. Cet évènement se traduit la plus part du temps par un combat contre des monstres aléatoires.

Aubergiste. En arrivant dans un nouveau lieu, le joueur a accès à un menu lui permettant d'acheter des objets utilisables en combat et de se préparer à l'évènement non aléatoire.

Evénement non aléatoire. Il dépend de l'histoire du jeu.

2.2 Conception logiciel

Dans cette section, nous expliciterons le diagrammes des classes pour les états présenté en fin de chapitre.

Classes Element. Cette classe et ses classes filles contiennent tout les éléments vivants du jeu. On distingue deux classes filles : Character et Monster. La classe Character possède des dépendances avec des classes comme SphereGrid et Item qui représentent l'évolution des personnages ainsi que les objets dont ils pourront faire l'usage.

Liste d'éléments. Elle va contenir toute les informations sur l'ensemble des éléments présent dans le jeu.

Classes Node. Cette classe permet de faire le lien entre l'apparition d'évènements et les éléments. C'est une liste chainée dont le déplacement est bidirectionnel.

Classe State. Cette classe contiendra toutes les informations liées au noeud où se trouve le joueur et aux éléments vivants / non vivants encore présent.

Observer. Cette classe permettra de relever les changements d'états du jeu et de le transmettre aux autres packages.

Chapitre 3

Rendu : Stratégie et Conception

3.1 Stratégie de rendu d'un état

Notre jeu étant à temps discret, le rendu d'un état sera assez simple. Le tout étant de distinguer entre le rendu et l'IHM qui correspond au menu.

Pour se faire, nous introduisons le concept d'instance, offrant une classe par contexte (ie. Combat, Carte du monde, Menu d'intro...) gérant l'IHM pour chaque environnement. Le rendu sera fait par les classes du package render en associant un Renderer à chaque contexte.

3.2 Conception logiciel

Renderer. Cette classe contient toutes les informations communes entre chaque contexte. Elle est néanmoins abstraite et ses classes filles (WorldMapRender, FightRender ...) doivent ainsi redéfinir ces méthodes afin de différencier correctement chaque contexte et chaque état du jeu.

TextureSetter. C'est dans cette classe que l'on va instancier toutes les textures des éléments vivants et non vivants du jeu. Ce tableau sera utilisé par les classes filles de Renderer.

Chapitre 4

Règle de changement d'état

Le jeu est à temps discret donc les changements d'états ne se feront uniquement lors d'appuis sur des touches du clavier par l'utilisateur. Chaque contexte offre des choix différents à l'utilisateur.

4.1 Conception logiciel

Screen. C'est la classe mère du package. Elle fait le lien entre la classe Application, ses classes filles et le package state afin de lier l'état et le rendu.

Application. Cette classe fait le lien entre la bibliothèque SFML et les différents états du jeu. En effet à chaque nouvel état nous aurons à instancier de nouveaux sprites.

Info, Worldmap, Fight, Inn. Les classes filles de Screen nous permette de mieux personaliser les contextes suivant les choix de l'utilisateur et sa position dans l'histoire du jeu.

Chapitre 5

IA

5.1 Comportement

L'intelligence artificielle peut gérer les personnages comme les monstres du jeux. Son comportement est dicté par la classe Rules (présent dans le package engine) avec les deux variables booléennes AICharneeded et AIMonsterneeded. Dans notre cas, l'intelligence artificielle doit choisir une action à effectuer sur une cible. Une première i.a. devra choisir aléatoirement une action ainsi que sa cible, une seconde devra effectuer une action aléatoirement mais sur la meilleure cible et enfin une troisième devra choisir l'action la plus rentable sur la meilleure des cibles. Pour cela nous utiliserons deux classes types : AI et ChoiceList.

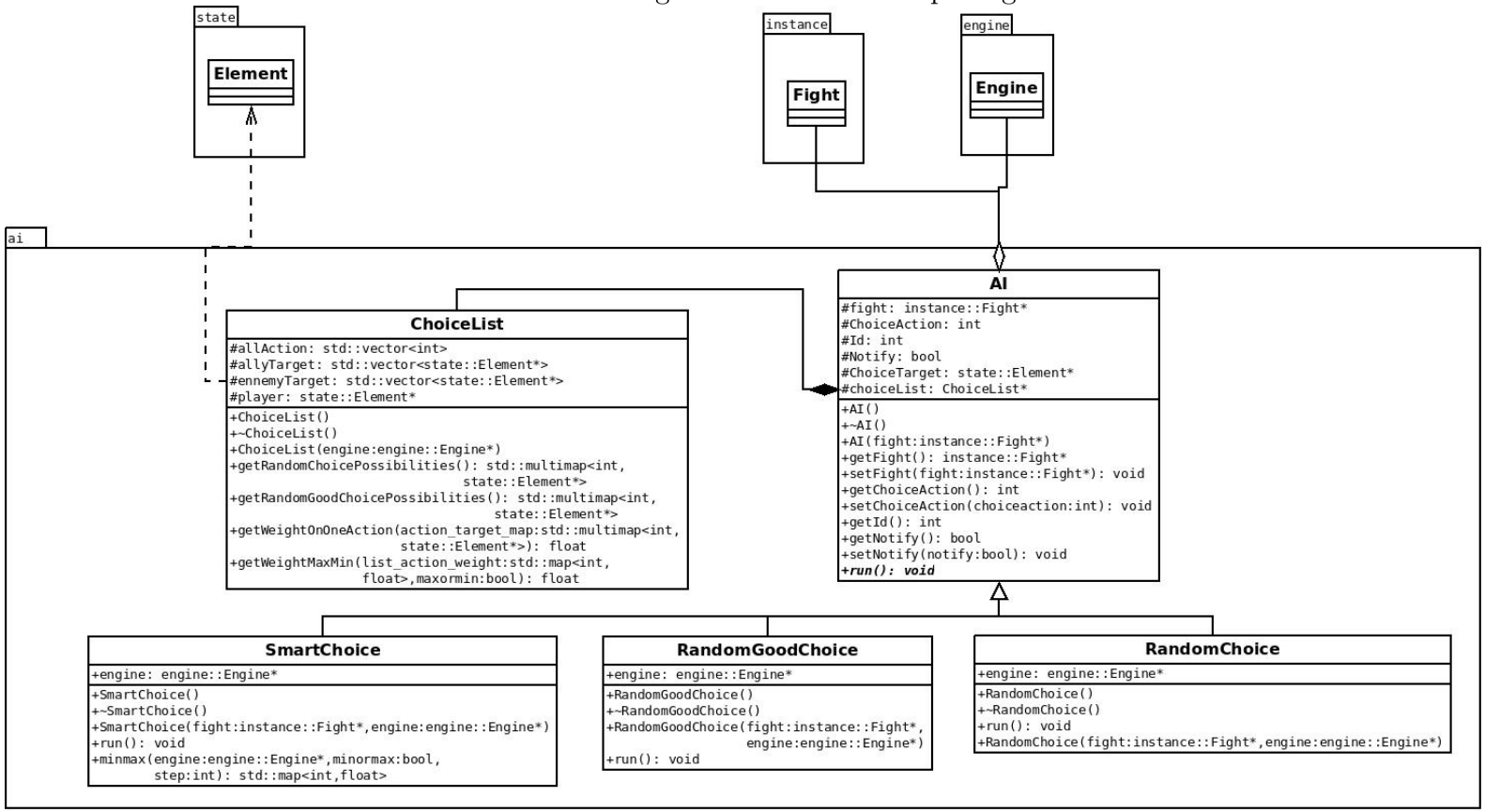
AI. Cette classe contient toutes les informations dont à besoin une intelligence artificielle pour fonctionner. Les variables ChoiceTarget et ChoiceAction sont récupérées par l'engine qui devra actualiser l'état du jeu. Les sous classes RandomChoice et RandomGoodChoice sont les intelligences artificielles premières et secondes citées plus haut. La classe SmartChoice est ainsi la troisième.

ChoiceList. Cette classe permet de créer un tableau avec les divers actions possibles sur n'importe quelles cibles. Ce tableau sera plus moins restreint suivant les besoins de l'intelligence artificielle utilisée. Par exemple, l'i.a. RandomChoice aura besoin de toutes les combinaisons possibles d'action et de cible alors que RandomGoodChoice fera un premier tri sur les cibles suivant l'action executée. La restriction du tableau suivant les différentes cibles possibles pour une action se base sur des paramètres logiques. Ainsi les actions défensives ne pourront pas être lancées sur des ennemies.

5.2 Intelligence artificielle haut niveau

L'intelligence artificielle dite de haut niveau (intitulé SmartChoice) s'appuit sur le parcours d'arbre de possibilités. En partant du tableau utilisé par l'i.a. RandomGoodChoice (qui conserve la meilleure cible pour une action donnée), elle applique un algorithme appelé algorithme de MinMax. Cette algorithme a pour rôle de retourner une map avec comme clef les différentes actions possibles et comme valeur liée à cette clef son gain maximum. Le calcul de gain pour chaque action se fait suivant une fonction de pondération présente dans la classe ChoiceList et le gain maximum est donc la somme du gain d'une action avec le gain le plus grand ou le plus faible de l'action du joueur suivant s'il est un allié ou un ennemi. Avec cette map, l'i.a. décide de jouer l'action ayant le gain "max" le plus élevé sur la meilleure cible, selectionné préalablement avec le tableau de l'i.a. RandomGoodChoice.

FIGURE 5.1 – Diagramme de classe du package ai



Chapitre 6

Engine et modularisation

L'engine a pour but de stocker les commandes générées par l'utilisateur ou l'IA avant qu'un thread parallèle au principal vide le tableau de commandes et les exécute.

6.1 Stockage de commandes

Les commandes envoyées par l'IHM sont de deux types différents : MoveInUI correspondant au changement d'état associé á un déplacement dans le menu et Action correspondant aux différentes actions lors d'un combat (attaque, objet, soin, etc..). Ces commandes sont stockées dans un tableau de l'engine en attendant que le thread secondaire les exécute.

6.2 Exécution des commandes

Le thread secondaire est pris dans une boucle infinie dans laquelle il transfère les commandes dans un tableau secondaire avant de le lock et d'exécuter toutes les commandes. Le tableau secondaire permet de ne pas locker le tableau principal, ce qui agirait comme un bottleneck dans l'ajout de commandes par l'IHM.

Chapitre 7

Server : conception de l'API

7.1 API REST

La partie serveur de l'API est instaurée via la librairie Pistache tandis que la partie client est instaurée via Chilkat.

Le but de l'API est de rassembler les différentes commandes générées sur un serveur distant avant que le client ne les récupère pour les exécuter mais aussi d'offrir les services CRUD sur une base de données d'utilisateurs.

7.1.1 Envoi des commandes au serveur

Requête : PUT /cmd
Données requête :

```
type: "object",
properties: {
    "unixtime": { type:number },
    "cmdtype": { type:string, pattern: "(action|moveinui)" },
    "action": { type:number, minimum:0, maximum:16 },
    "casterindex": { type:number, minimum:0},
    "targetindex": { type:number, minimum:0},
required: [ "unixtime", "cmdtype", "action", "casterindex", "targetindex" ]
```

Réponse : STATUS 200 si OK, STATUS 500 si une erreur est survenue.

7.1.2 Récupération des commandes

Requête : GET /cmd
Réponse : STATUS 200 si OK, STATUS 404 si aucune commande n'est sur le serveur.

Données Réponse :

```
type: "object",
properties: {
    "command": { type:object, minItems:0 },
required: []
```

7.1.3 Ajout/modification d'utilisateur

Requête : PUT /user/<id>
Données requête :

```
type: "object",
properties: {
    "id": { type:number },
    "nom": { type:string }
required: [ "id", "nom" ]
```

Crée un utilisateur si il n'existe pas, le modifie sinon.
Réponse : STATUS 200 si OK, STATUS 500 si une erreur est survenue.

7.1.4 Demande d'utilisateur

Requête : GET /user/<id>
Réponse : STATUS 200 si OK, STATUS 404 si l'utilisateur n'existe pas.
Données réponse :

```
type: "object",
properties: {
    "id": { type:number },
    "nom": { type:string }
required: [ "id", "nom" ]
```

7.1.5 Demande de tous les utilisateurs

Requête : GET /user/all
Réponse : STATUS 200 si OK, STATUS 500 si il y a eu un problème.
Données réponse :

```
type: "object",
properties: {
    "object": { type:number }
required: [ "object" ]
```

7.1.6 Suppression d'un utilisateur

Requête : DELETE /user/<id>
Réponse : STATUS 200 si OK, STATUS 404 si l'utilisateur n'existe pas.