Projet Logiciel Transversal

Elodie FOREAU – Gabriel HARANG

Vamp & Wolf

1 Objectif

1.1 Présentation générale

Notre projet consiste à adapter sur ordinateur le jeu de rôle Dungeons and Dragons (D&D). Nous allons nous inspirer de Baldur's gate en le rendant tour par tour ayant des règles plus simples. Ce jeu s'appellera Vamp & Wolf (V&W).

1.2 Règles du jeu

Le but du jeu est de faire évoluer son personnage à travers des quêtes.

- Univers

L'univers est une carte 2D contemporaine et fantastique en vue du dessus. Elle est composée de zones urbaines et forestières, d'obstacles et des personnages qui se déplacent dessus.

1: Exemple de vue de dessus
(tiré du jeu Pokémon Rouge)

- Personnage

Les personnages intéragissent avec l'univers, d'autres personnages et des objets. Ils possèdent un inventaire, des compétences, une profession et une race.

- Inventaire

L'inventaire est un outil permettant de regrouper et de ranger les objets de défense et d'attaque d'un personnage. Il peut également contenir les objets de quête et les objets récupérés sur la carte. L'inventaire est limité : pour ajouter un objet lorsque l'inventaire est plein, il faut enlever un objet de celui-ci.

2: Exemple d'inventaire (tiré du jeu Baldur's Gate)

- Niveau

Les niveaux sont les différentes évolutions du personnage. Pour faire évoluer son personnage, le joueur doit acquérir de l'expérience jusqu'à un seuil qui lui permettra de débloquer le niveau supérieur et des compétences plus puissantes.

- Compétence

Les compétences sont le déplacement, l'attaque et la défense, la discussion, le ramassage des objets, le vol, la rage et les sorts. Par défaut, un personnage peut parler, attaquer, se défendre, se déplacer et ramasser des objets.

- Profession

La profession correspond au métier du personnage, elle définit ces spécificités. Les différents types de profession sont soldat, sorcier et voleur.

- Race

La race correspond aux avantages et inconvénient du personnage. Les différents types de races sont humain, loup-garou, vampire et monstre.

- Objet

Les objets sont des armes, des potions et des objets de quête. Un système monétaire peut être envisagé.

- Quête

C'est une suite d'événements demandés au joueur qui sera récompensé par de l'expérience et/ou des objets, et donc l'augmentation du niveau du personnage.

2 Description et conception des états

2.1 Description des états

Notre jeu est séparé en deux parties : l'état déplacement sur la carte et l'état combat. L'état déplacement sur la carte est constitué d'éléments fixes et d'éléments mobiles. Tous les éléments possèdent les propriétés suivantes :

- Coordonnées (x,y),
- Identifiant de type élément : ce nombre indique la nature de l'élément (classe).

2.1.1 Etat éléments fixes de la carte

Le principal élément fixe est la carte qui sera représenté par une grille (qui aura une taille fixée). Elle comprend des éléments franchissables et infranchissables.

Éléments infranchissables :
Les éléments infranchissables ne sont pas franchis par les éléments mobiles, ils sont considérés comme des obstacles. Dans cette catégorie, on considère les arbres, les montagnes, les bâtiments et les coffres (qui contiendront les objets de quête).

Éléments franchissables :
Les éléments franchissables sont franchis par les éléments mobiles. Ces éléments sont les prairies, les routes, les chemins. L'emplacement initial des éléments mobiles se situe sur un élément franchissable.

2.1.2 État éléments mobiles de la carte

Les éléments mobiles possèdent une direction (immobile, gauche, droite, haut ou bas) une vitesse et une position. Notre élément mobile est le personnage.
On considère que le personnage est :

- différent selon les deux états du jeu, c'est-à-dire que le même personnage existe sur la carte et peut être en train de combattre,
- et identique dans le sens où si le personnage meurt en combat il n'existe plus sur la carte.

Le personnage joueur est un personnage dirigé par le joueur qui commande la direction de celui-ci. Il peut se déplacer, discuter et ramasser des objets.
Il possède donc un inventaire qui contient des objets, des armes, etc. Il a également des points de vie et des manas.

Nous avons trois catégories de personnage : ami, ennemi ou neutre

2.1.3 État général de la carte

Sur la carte, nous rajoutons les propriétés suivantes :

- Liste des quêtes : permet de savoir si le personnage peut augmenter de niveau,
- Liste des coffres : permet de connaître si la quête est terminée,
- Liste des éléments infranchissables : permet de savoir si le personnage peut se déplacer.

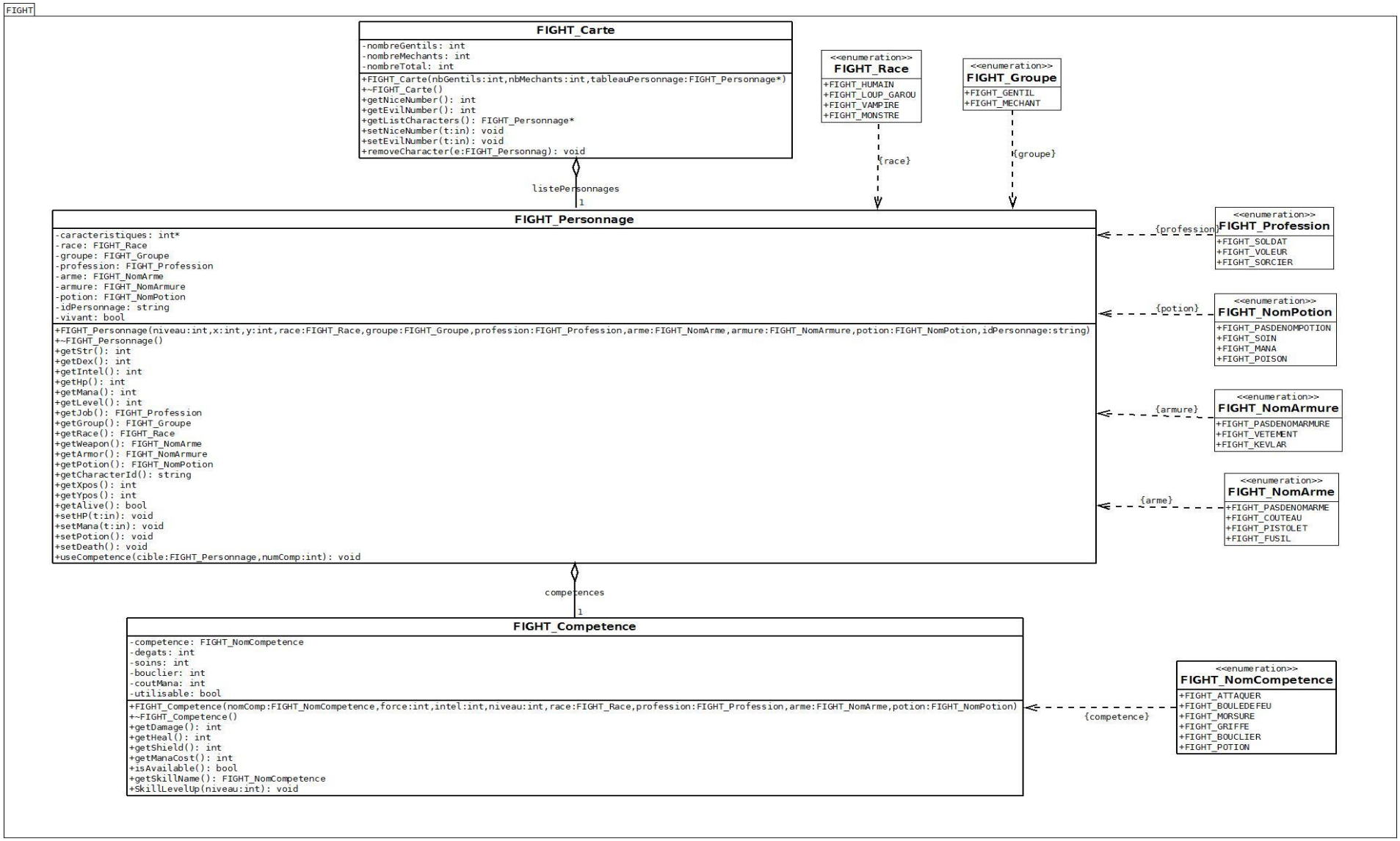
2.1.4 État combat

Cet état est constitué que d'éléments fixes c'est-à-dire que les personnages en combat ne se déplacent pas. Ils lancent des sorts et des attaques, ils reçoivent des dégâts. Le personnage a des compétences qui lui permettent d'attaquer. Il possède comme le personnage sur la carte des points de vie et des manas.

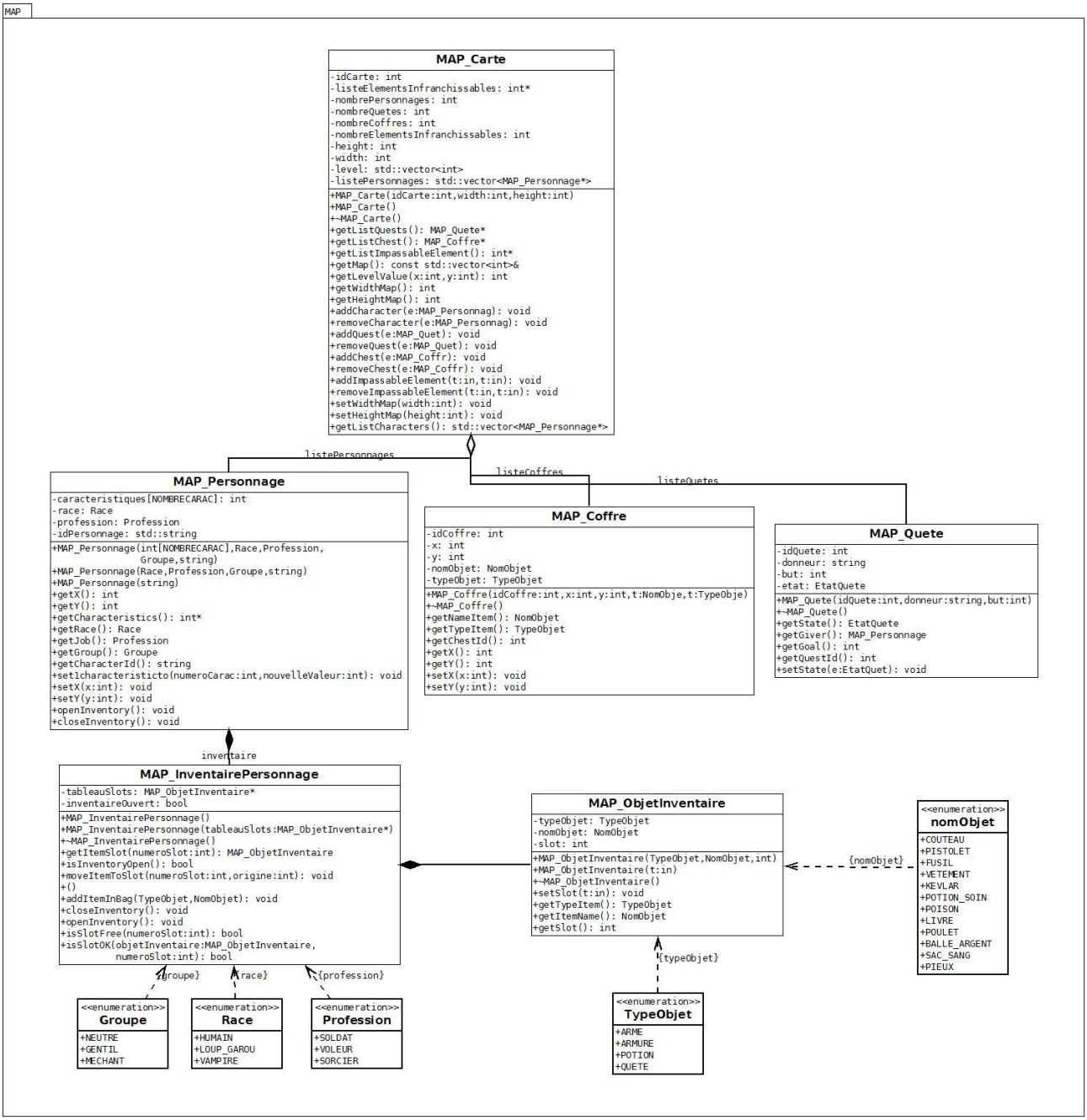
2.1.5 État général

Les deux états du jeu possèdent :

- Une époque : représente l'heure correspondant à l'état c'est-à-dire le tic de l'horloge globale depuis le début de la partie,
- La vitesse : le nombre d'époque par seconde, c'est-à-dire la vitesse à laquelle l'état du jeu est mis à jour,
- Liste de personnages : le nombre de personnages présents sur la carte ou vivants en combat (cette liste se situe au niveau de la carte pour chaque partie du jeu).



3: Diagramme des classes d'état pour la partie combat



4: Diagramme des classes d'état pour la partie carte

3 Rendu : Stratégie et conception

3.1 Stratégie de rendu d'un état

Pour le rendu d'un état, nous avons choisi d'utiliser une stratégie bas niveau. En effet, le CPU prépare les éléments à rendre au sein de structures élémentaires avant de tout envoyer au GPU.

Nous découpons la scène à rendre selon trois plans : un plan pour le niveau (herbe, arbre, chemin, etc), un plan pour les éléments mobiles (personnages) et un plan pour les informations (point de vie, mana, scores, etc).

En ce qui concerne la synchronisation, nous avons deux horloges : une pour les changements d'état et une pour la mise à jour du rendu à l'écran. L'horloge des changements d'états sera plus lente que celle des rendus. En conséquence, il faut interpoler entre deux changements d'états pour pouvoir obtenir un rendu lisse :

- Pour les animations qui n'ont aucun lien avec l'état, nous avons choisi de définir une fréquence pour chaque animation. Les animations tournent donc en boucle, sans corrélation avec les deux horloges.
- Pour les animations qui ont un lien avec l'état (comme le déplacement des personnages), on produit un rendu équivalent un à sous-état fictif, produit d'une horloge fictive des changements d'états synchronisée avec celle du rendu. Par exemple, pour le déplacement d'un élément mobile, on calcule la position intermédiaire entre celle de l'état courant, et celle de l'état à venir.

3.2 Conception logicielle

Le diagramme des classes pour le rendu général, indépendante de toute librairie graphique (SFML), est présenté en Illustration 7.

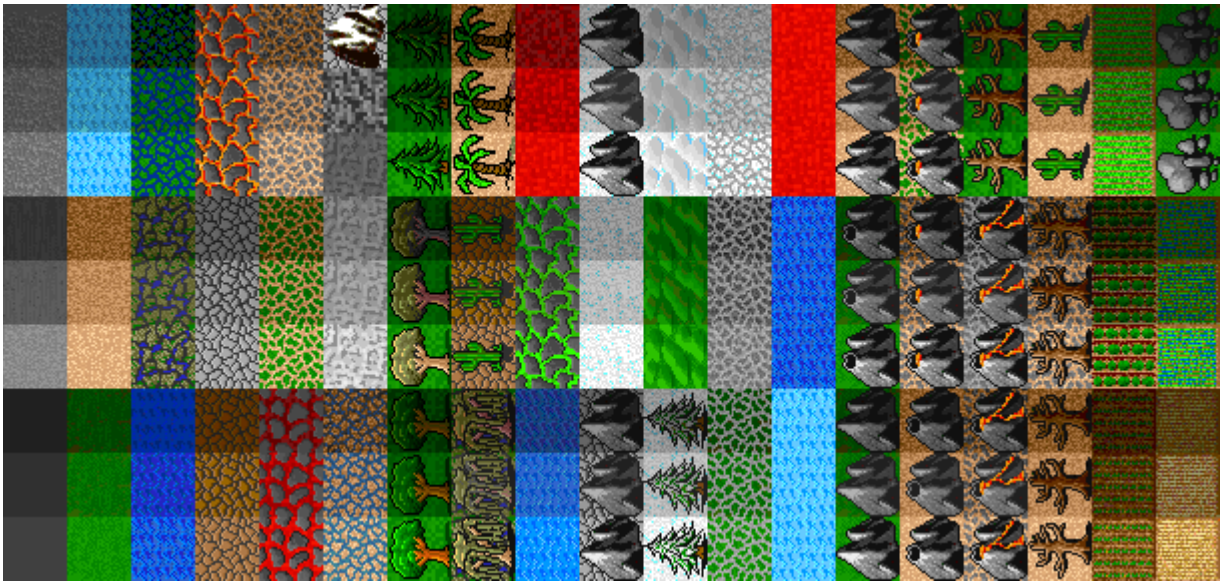
« **TileMapNew** ». Cette classe ne contient qu'une méthode permettant de découper une tuile de 32 par 32 (voir illustration 5).

« **Sprite** ». Cette classe permet de créer un sprite pour les objets (personnages pour le moment), de leur donner des coordonnées et une direction. Le sprite est seulement une représentation de l'objet.

« **Fenêtre** ». Cette classe permet d'afficher un élément contenu dans RENDU_ElementGraphique.

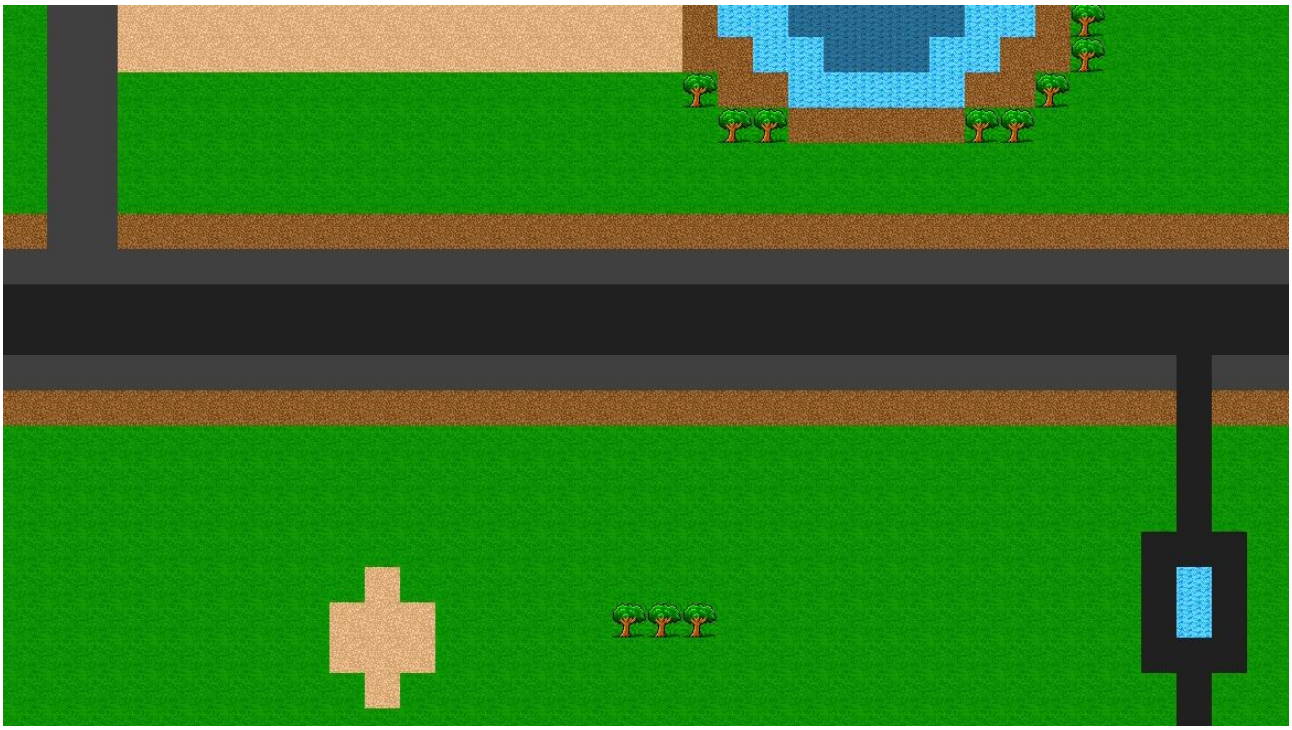
3.3 Exemple de rendu

Voici un exemple de texture pour le mode carte :

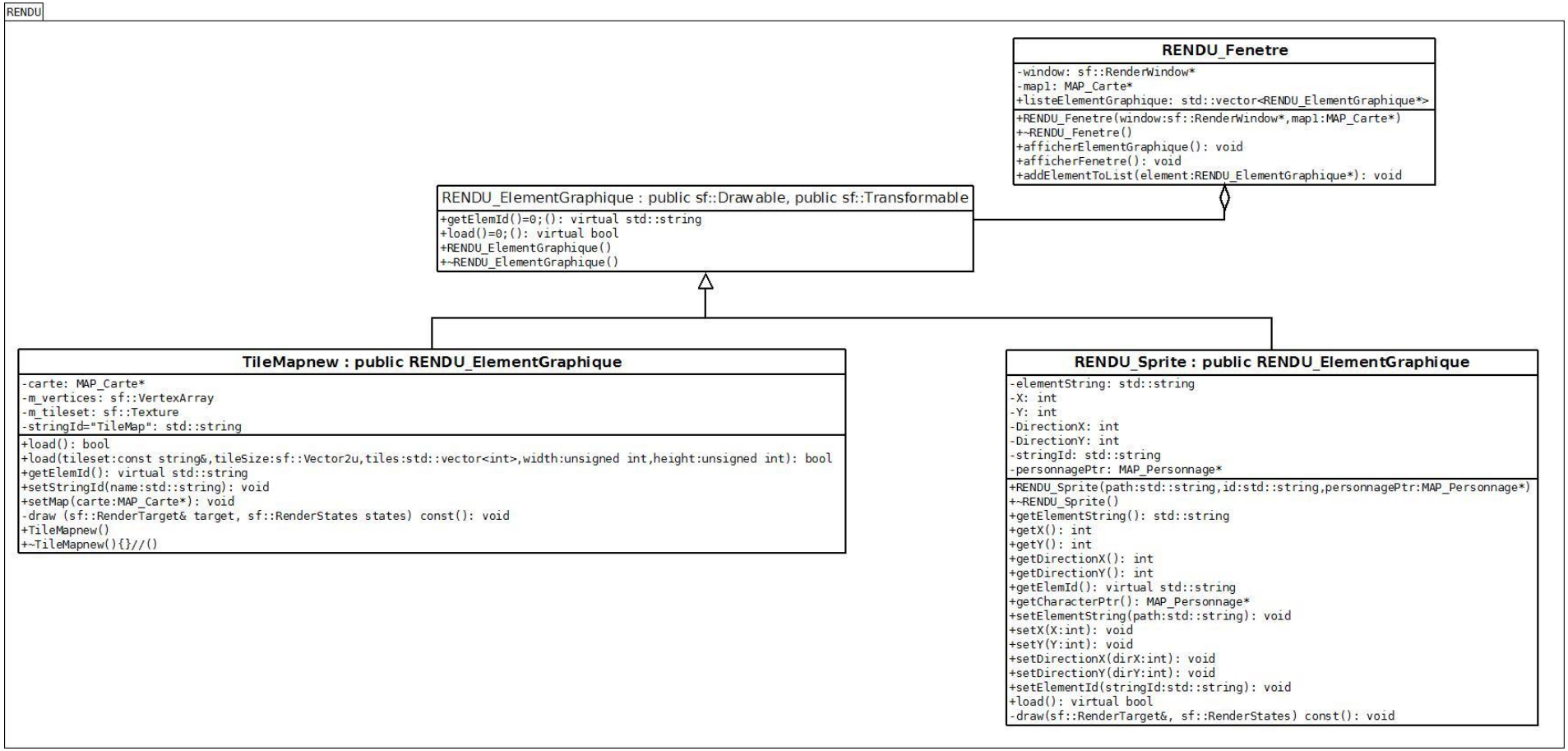


5:Exemple de texture – Mode carte

Voici un exemple de rendu du jeu pour la carte sans personnages pour le moment.



6: Exemple de rendu – Mode carte



7: Diagramme de classes pour le rendu

4 Règles de changement d'états et moteur de jeu

4.1 Horloge globale

Les changements d'état suivent une horloge globale, de manière régulière, on passe directement d'un état à un autre. Il n'y a pas de notion d'état intermédiaire. Ces changements sont calibrés sur le temps qu'il faut, à un élément mobile à vitesse maximale, pour passer d'une case à une autre. En conséquence, tous les mouvements auront une vitesse fonction de cet élément temporel unitaire. Notons bien que cela est décorrélé de la vitesse d'affichage : l'utilisateur doit avoir l'impression que tout est continu, bien que dans les faits les choses évoluent de manière discrète.

4.2 Changements extérieurs

Les changements extérieurs sont provoqués par des commandes extérieures telles que l'appui sur une touche du clavier ou un clic de la souris, mais peut également provenir d'un ordre du réseau. Nous avons trois types d'actions :

- Actions en mode carte :
  → Charger un niveau : on fabrique un état initial à partir d'un fichier ;
  → Nouvelle partie : on remet à l'état initial ;

- Commandes Modes : on modifie le mode actuel du jeu comme par exemple « en pause » ou « rejouer la partie » ;

- Commandes Direction Personnage : la direction du personnage est modifiée si cela est possible.

4.3 Changements autonomes

Les changements autonomes sont appliqués à chaque création ou mise à jour d'un état après les changements extérieurs.

Les règles sont pour la carte :

1. Appliquer les règles de déplacement du personnage.
2. Si le personnage rencontre un autre personnage, il peut discuter avec.
3. Si le personnage est sur une case contenant un objet, il peut le ramasser s'il a de la place dans son inventaire.
4. Suppression d'un objet dans l'inventaire si le joueur le souhaite

Les règles pour le combat :

1. Appliquer les règles d'attaque du personnage
2. Selon les dégâts corps à corps reçus par l'adversaire, appliquer les règles de défense du personnage. (Perte de point de vie).
3. Recommencer à 1. Tant qu'un membre de chaque équipe est vivant.

4.4 Conception logiciel

Le diagramme des classes pour le moteur du jeu est présenté en Illustration 7.

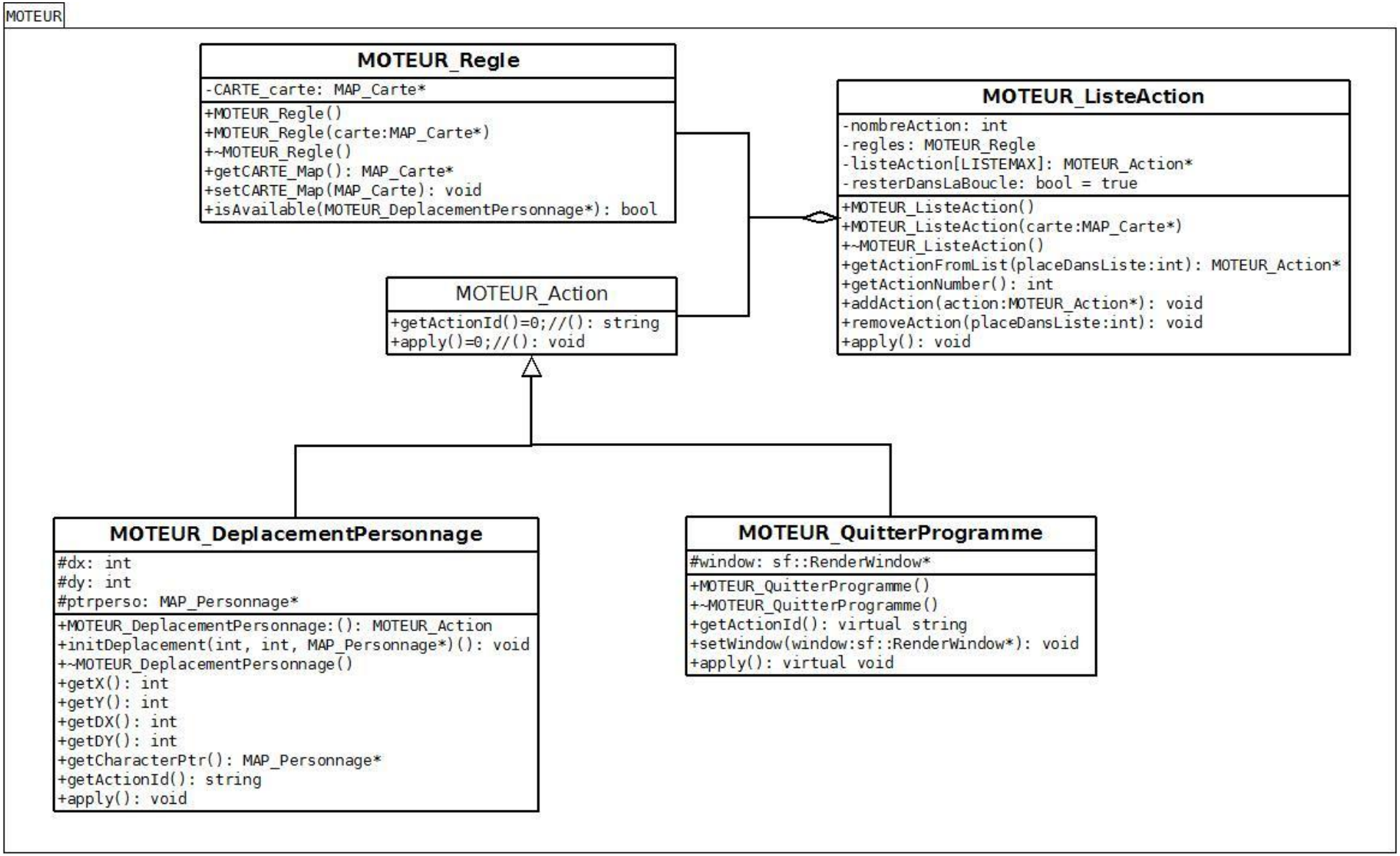
« **Regles** ». Cette classe permet de vérifier si l'action est possible. Par exemple, pour le déplacement d'un personnage, il faut vérifier qu'il ne traverse pas un arbre, c'est donc le rôle de la méthode *isAvailable()* qui gère les collisions et les contours de l'écran.

« **Actions** ». Le rôle des classes MOTEUR_Action et MOTEUR_ListeActions est de représenter une modification particulière d'un état du jeu mais également de représenter une commande extérieure, provenant par exemple d'une touche clavier (ou de la souris). Notons bien que ces classes ne gèrent absolument pas l'origine des commandes, ce sont d'autres éléments en dehors du moteur de jeu qui fabriqueront les instances de ces classes (Classes de l'IHM). De plus, Elles ne gèrent pas non plus les règles du jeu : chaque instance de ces classes applique la modification qu'elle contient, sans se demander si cela a un sens. On ajoute donc des actions dans une liste actions à exécuter.

La classe MOTEUR_DeplacementPersonnage permet de créer l'action associée au déplacement d'un personnage.

4.5 Conception logiciel : extension pour la parallélisation

La classe MOTEUR_QuitterProgramme permet d'arrêté le thread relatif au moteur de jeu.



8: Diagramme de classes pour le moteur de jeu

5 Intelligence artificielle

5.1 Stratégies

5.1.1 Intelligence minimale

Nous proposons une intelligence extrêmement simple, basée sur les principes suivants :

Pour le mode carte :

- Tant que c'est possible on avance vers un personnage ou un objet,
- Lorsqu'on arrive devant un obstacle (arbres, eau, etc.), on choisit l'un des côtés qui est disponible. Cela permet de pas rester bloqué dans un coin,
- Lorsque l'on rencontre un personnage, on lui parle puis on le combat.

Pour le mode combat :

- On choisit aléatoirement quelles attaques et défenses utilisées pour le personnage. Cela permet de voir si le personnage va gagner ou perdre et de voir quelles combinaisons sont possibles (attaque de l'adversaire, réponse du personnage en fonction de l'attaque).

5.1.2 Intelligence basée sur des heuristiques

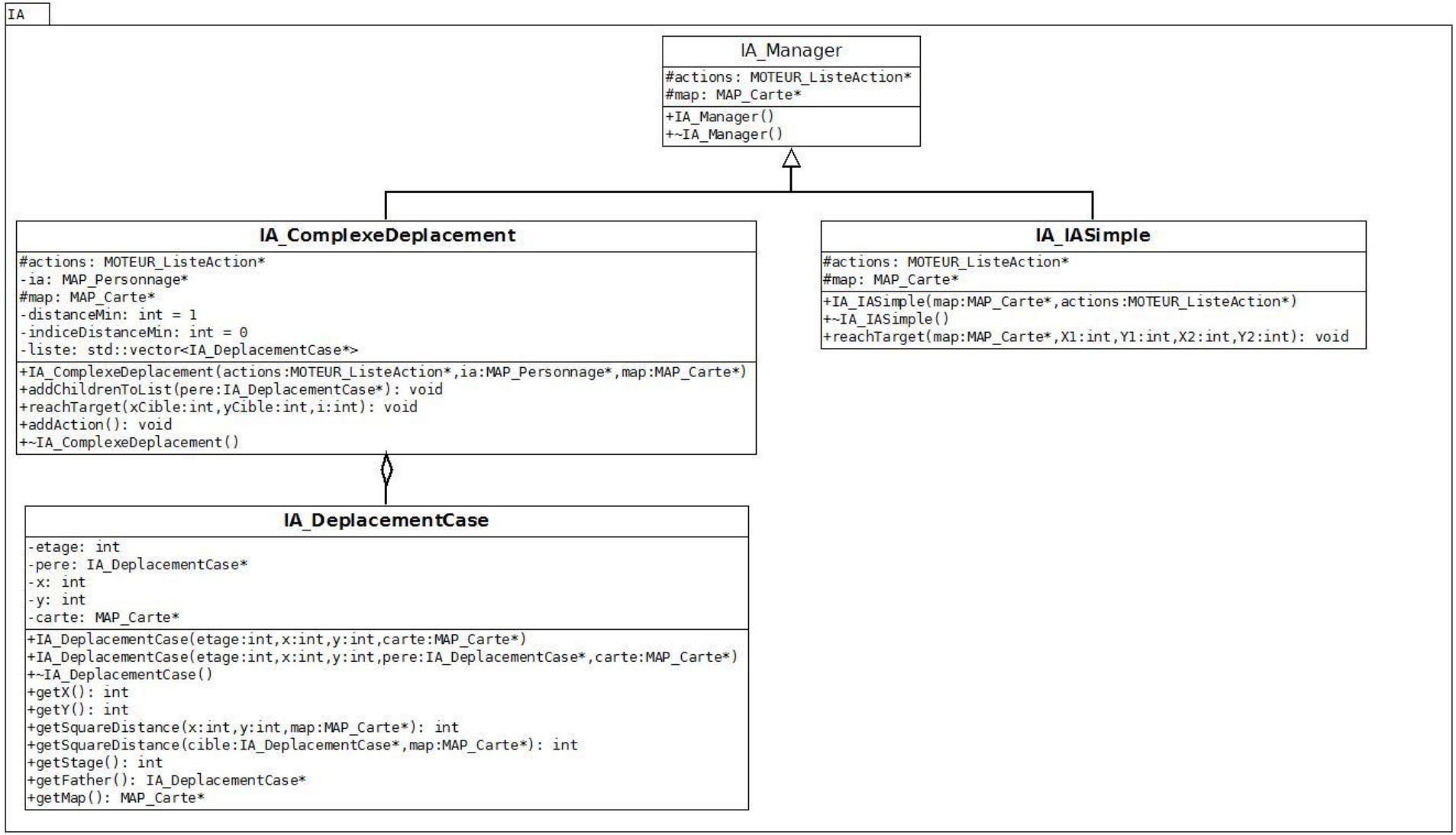
Nous utilisons l'heuristique basée sur la diminution de la distance que ce soit pour l'intelligence artificielle simple ou complexe.

5.1.3 Intelligence basée sur les arbres de recherche

Pour avoir une intelligence artificielle améliorée, nous utilisons la recherche en largeur.

5.2 Conception logiciel

« **IA** ». Toutes les formes d'intelligence artificielle implantent la classe abstraite IA_Manager. Le rôle de ces classes est de fournir un ensemble d'actions à transmettre au moteur de jeu. Notons qu'il n'y a pas une instance par personnage, mais qu'une instance doit fournir les actions pour tous les personnages. La classe IA_IASimple implante l'intelligence minimale, telle que présentée ci dessus. De même, la classe IA_ComplexeDeplacement implante la version améliorée.



9: Diagramme de classes pour l'intelligence artificielle

6 Modularisation

6.1 Organisation des modules

6.1.1 Répartition sur différents threads

Le but est de placer le moteur de jeu et l'état sur un thread et le rendu et l'interface homme-machine sur un autre thread. Nous avons deux types d'information qui transite entre les différents modules : les actions et les informations de l'état du jeu.

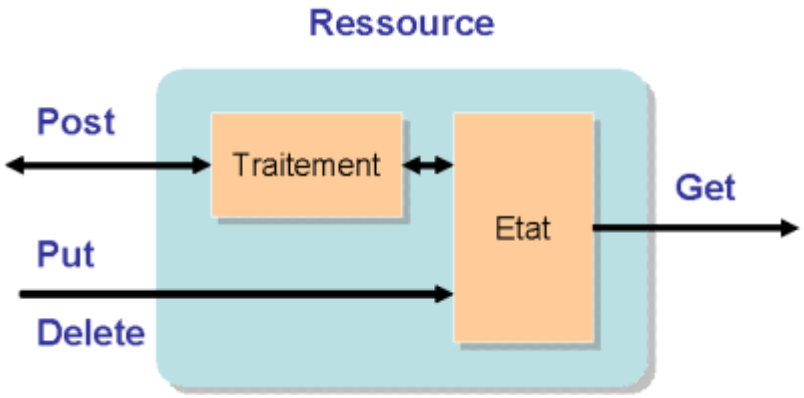
Les actions sont en réalité les touches clavier pressées par le joueur. Celles-ci peuvent arriver à n'importe quel moment. Pour résoudre ce problème, nous comptons utiliser un double tampon d'actions. L'un contiendra les actions actuelles et l'autre les actions à traiter. Ainsi, on permute les deux tampons pour qu'il y en ait toujours un qui puisse accueillir les nouvelles actions.

Les informations de l'état du jeu sont, pour le moment, envoyés au moteur de rendu sans besoin de faire une mise à jour. Le rendu correspond à l'état du jeu.

6.1.2 Répartition sur différentes machines

Nous comptons répartir sur un serveur tout ce qui est relatif au moteur de jeu et à l'état du jeu. Pour la partie client, nous utiliserons les modules de l'interface homme-machine et du moteur de rendu. Pour l'échange de données entre le client et le serveur nous utilisons la technologie REST.

REST (Representational State Transfer) est une architecture orienté ressource qui se pose en alternative à SOAP (Service Oriented Architecture Protocol), permettant la réalisation d'architecture orientée services utilisant des services Web destinés à la communication entre machines. L'application de cette architecture utilise ces quelques principes :
* URI : pour nommer et identifier une ressource,
* HTTP : pour fournir toutes les opérations nécessaires (GET, POST, PUT et DELETE),
* JSON : le format utilisé pour l'échange des données.



10: Caractéristique d'un web service REST

Pourquoi REST et pas un autre web service ?

Si on veut comparer SOAP et REST, ces deux web services sont différents. SOAP est basé sur une architecture orientée opération basé sur XML et REST est basé sur une architecture orientée ressources qui se base uniquement sur l'usage des requêtes du protocole HTTP. SOAP ajoute au protocole de communication une enveloppe XML : des entêtes et un document. Le protocole de communication est la plupart du temps HTTP. REST est basé directement sur le protocole HTTP, Ainsi, REST est plus simple et donc plus rapide que SOAP.

Quel format utilisons-nous ?

JSON (JavaScript Object Notation – Notation Objet issue de JavaScript) est un format léger d'échange de données. Il est facile à lire ou à écrire pour des humains. Il est aisément analysable par des machines. JSON est un format texte complètement indépendant de tout langage. Ces propriétés font de JSON un langage d'échange de données adapté à notre cas.

Un document JSON ne comprend que deux éléments structurels :

- des ensembles de paires (nom / valeur)
- des listes ordonnées de valeurs.

Ces mêmes éléments représentent 3 types de données : des objets, des tableaux et des valeurs génériques de type tableau, objet, booléen, nombre, chaîne ou null.

6.2 Conception logicielle

Le diagramme présente le diagramme des classes utilisées. On peut distinguer plusieurs parties :

- Classes Actions et ActionsDB : permettent de simuler une petite base (ie, dans un cas plus réel, ce serait l'interface vers une base SQL ou noSQL). Les différentes méthodes parlent d'elles-mêmes, et correspondent aux quatre opérations CRUD usuelles (Create/Remove/Update/Delete).
- Classe AbstractService : classe abstraite pour tout service REST de ce sujet.
- Classes ActionServices : permait d'implémenter les opérations (GET/POST/PUT/DELETE).
- Classe ServicesManager : le gestionnaire de service, c'est-à-dire qui sélectionne le bon service et la bonne opération à exécuter en fonction de l'url et de la méthode HTTP.
- Classe ServiceException : permet de jeter une exception à tout moment pour interrompre l'exécution du service, de la manière suivante :

```
throw new ServiceException(<code status HTTP>,<message>);
```

Le code status HTTP est une des valeurs de l'enum HttpStatus, et le message une chaîne de caractère.

En ce qui concerne les opérations GET, POST, PUT et DELETE, voici ce qui est attendu :

- GET

Cette opération permet de renvoyer les informations concernant une action.
Exemple d'url : http://localhost:8080/action/1
Résultat :

```
{
        « id » : « 1 »
}
```

- POST

Cette opération permet de modifier les informations d'une action.
Exemple d'url : curl –X POST –data '{« id » :24}' http://localhost:8080/action/1
Cela modifie l'id de l'action 1 devenant 24. POST ne renvoie aucunes données. Pour vérifier la modification il faut utiliser l'opération GET.

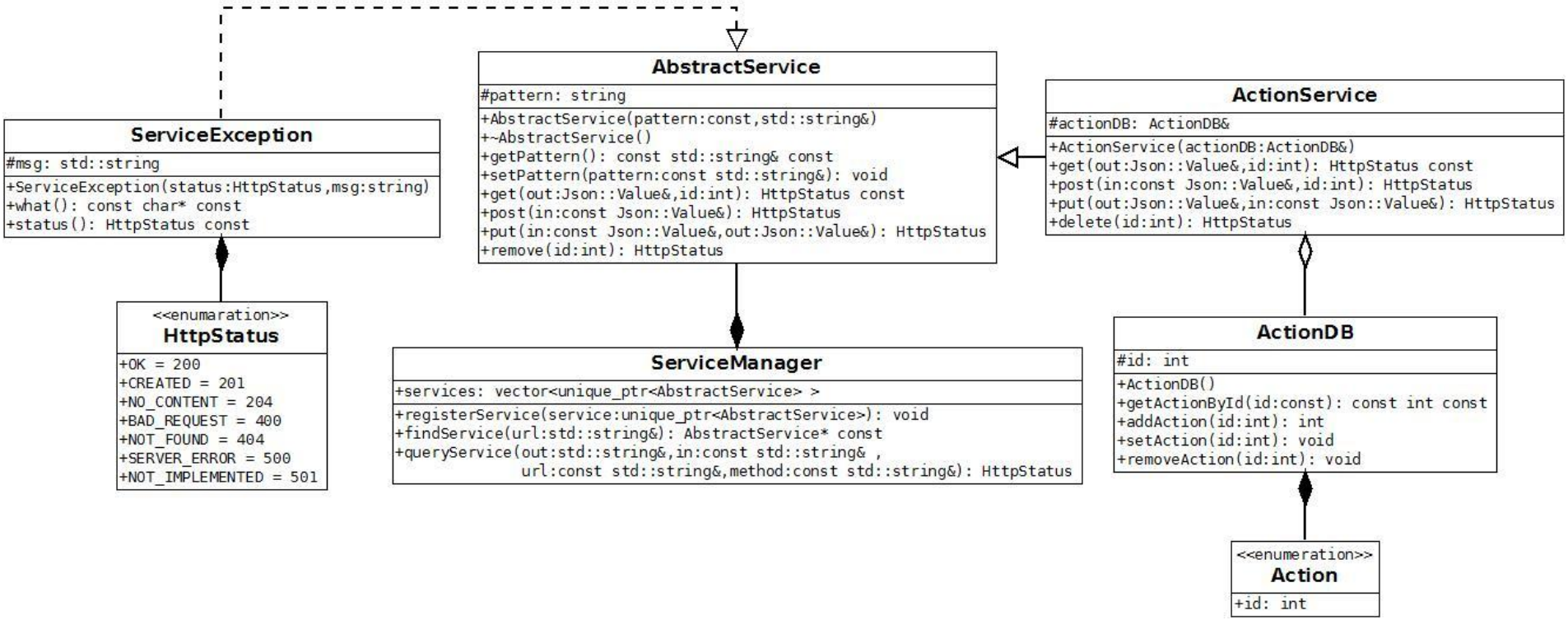
- PUT

Cette opération permet d'ajouter une action.
Exemple d'url : curl –X PUT –data '{« id » : »2 »}' http://localhost:8080/action
Résultat :

```
{
        « id » : « 2 »
}
```

- DELETE

Cette opération permet de supprimer une action désignée par id dans l'url.
Exemple d'url : http://localhost:8080/action/1
Il n'y a pas de données entrantes ni sortantes.



11: Diagramme des classes du web service