



Projet Logiciel Transversal

Elodie FOREAU – Gabriel HARANG

Vamp & Wolf

1	Objectif.....	3
1.1	Présentation générale.....	3
1.2	Règles du jeu	3
2	Description et conception des états	5
2.1	Description des états.....	5
2.1.1	Etat éléments fixes de la carte	5
2.1.2	État éléments mobiles de la carte	5
2.1.3	État général de la carte	5
2.1.4	État combat	6
2.1.5	État général	6
3	Rendu : Stratégie et conception	9
3.1	Stratégie de rendu d'un état.....	9
3.2	Conception logicielle.....	9
3.3	Conception logicielle : extension pour les animations	10
3.4	Exemple de rendu	11

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 interagissent 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 débloquent 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.

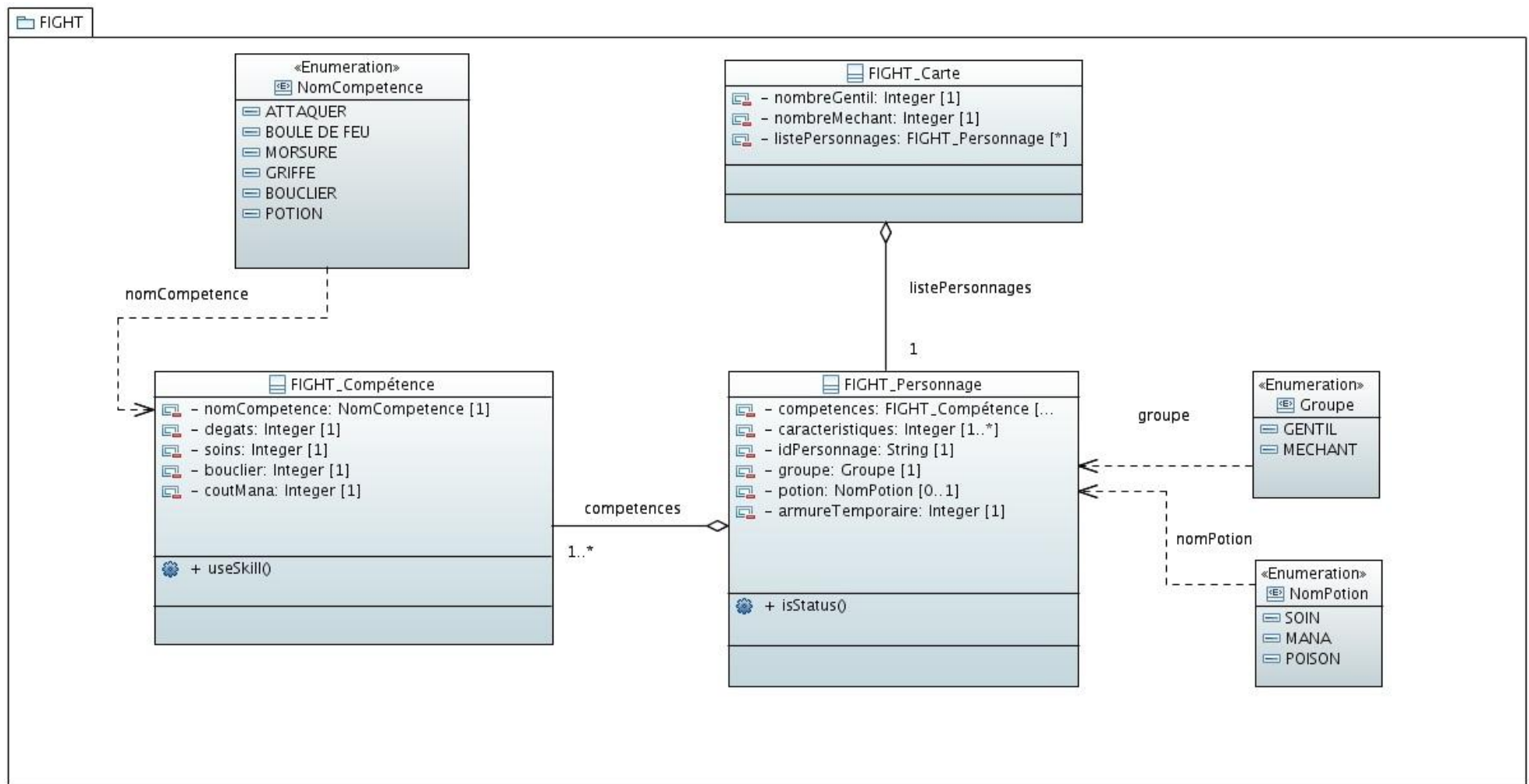
2.1.4 État combat

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.

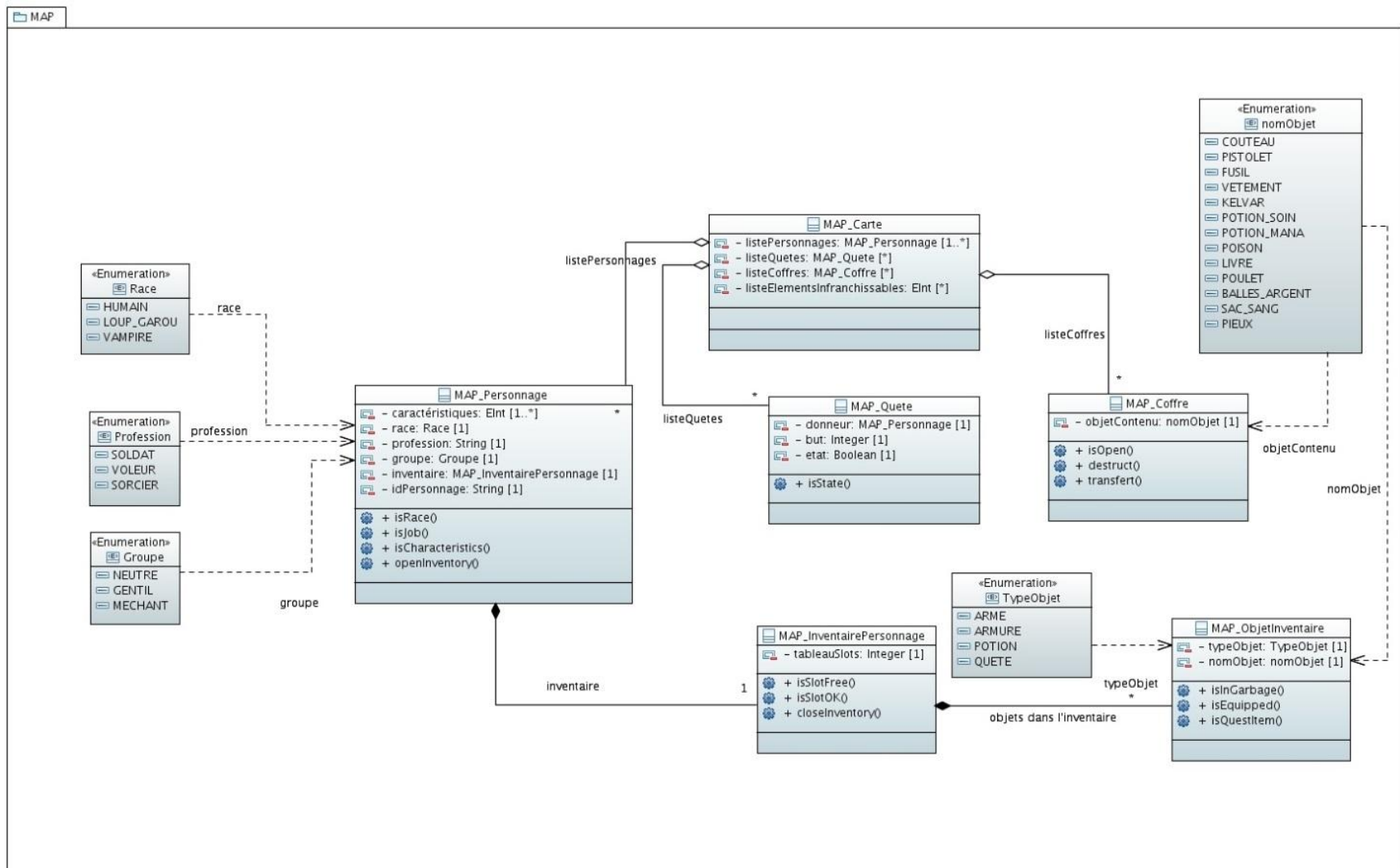
2.1.5 État général

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). Chaque plan contiendra deux informations bas-niveau qui seront transmises à la carte graphique : une unique texture contenant les tuiles et une unique matrice avec la position des éléments et les coordonnées dans la texture. En conséquence, chaque plan ne pourra rendre que les éléments dont les tuiles sont présentes dans la texture associée.

Pour la formation de ces informations bas-niveau, la première idée est d'observer l'état à rendre, et de réagir lorsqu'un changement se produit. Si le changement dans l'état donne lieu à un changement permanent dans le rendu, on met à jour le morceau de la matrice du plan correspondant. Pour les changements non permanent, comme les animations et/ou les éléments mobiles, nous tiendrons à jour une liste d'éléments visuels à mettre à jour (modification de la matrice du plan) automatiquement à chaque rendu d'une nouvelle frame.

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 général, l'horloge des changements d'états sera dans les 4-12Hz alors que celle des rendus dans les 30-60Hz. 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, est présenté en Illustration 4.

Plan et **Surface** : Le cœur du rendu réside dans le groupe autour de la classe Plan. Le principal objectif des instances de Plan est de donner les informations basiques pour former les éléments bas-niveau à transmettre à la carte graphique. Ces informations sont données à une implantation de Surface. Cette implantation non représentée dans le diagramme, dépendra de la librairie graphique

choisie. L'ensemble classe Surface avec ses implantations suivent donc un patron de conception de type Adapter. La première information donnée est la texture du plan, via la méthode *loadTexture()*. Les informations qui permettront à l'implantation de Surface de former la matrice des positions seront données via la méthode *setSprite()*. Pour plus d'efficacité, nous indexons tous les éléments graphiques (sprites). Ainsi, la surface sait qu'il faut gérer un nombre fixe de sprites, chacun identifié par un numéro unique. La classe Plan est le cas général, que l'on peut spécialiser avec des instances de EtatPlan et ListElementPlan, chacun étant capable de réagir à des notifications de changement d'état via le mécanisme d'Observer.

Scène: Tous les plans sont regroupés au sein d'une instance de Scene. Les instances de cette classe seront liées à un état particulier. Une implantation pour une librairie graphique particulière fournira des surfaces via les méthodes *setSurface()*. Notons bien que les instances ont pour rôle de remplir les surfaces, mais pas de les rendre : cela restera le travail de la librairie choisie.

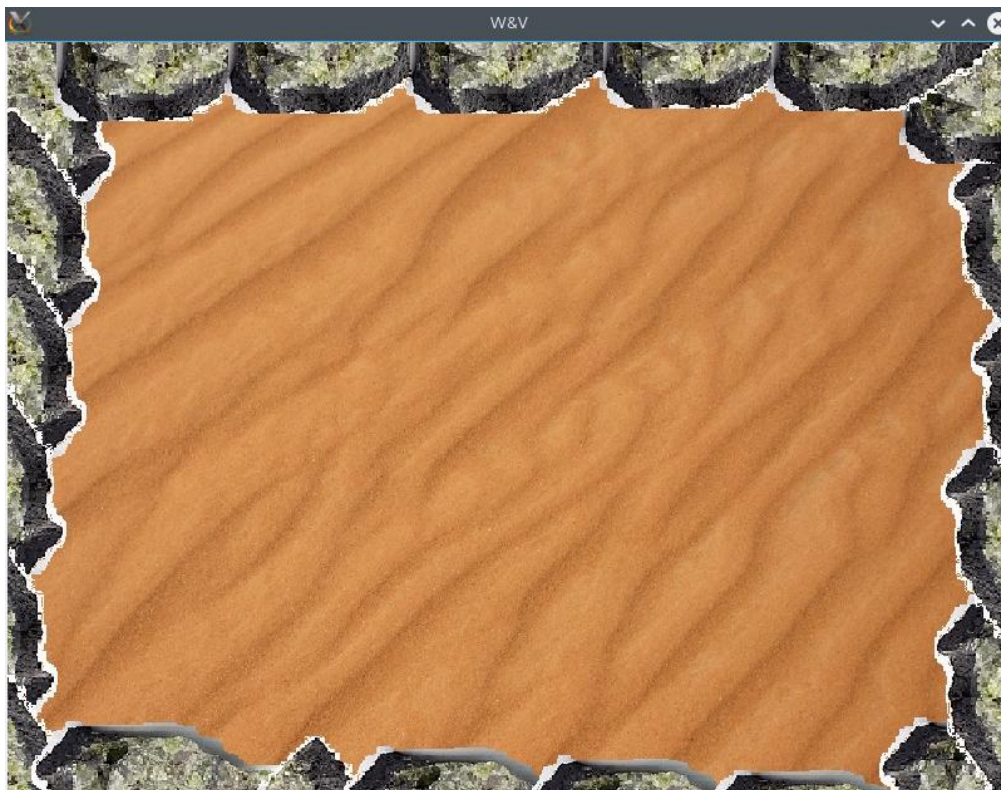
Tuiles: La classe Tuile ainsi que ses filles ont pour rôle la définition de tuiles au sein d'une texture particulière, ainsi que les animations que l'on peut former avec. La classe StaticTuile stocke les coordonnées d'une unique tuile, et la classe AnimatedTuile stocke un ensemble de tuiles. Etant donné qu'AnimatedTuile est à la fois une classe fille de Tuile et un conteneur de celle-ci, nous sommes donc face à un patron de conception de type Composite. Enfin, les implantations de la classe JeuDeTuile stocke l'ensemble des tuiles et animations possibles pour un plan donné. Notons que nous ne présentons pas dans le diagramme des implantations, uniquement la classe abstraite JeuDeTuile. En outre, on peut voir ces classes comme un moyen de définir un thème graphique : lors de la création d'instance de Plan, on pourra choisir n'importe quel jeu de tuile, pourvu qu'il contient des tuiles cohérentes avec le plan considéré.

3.3 Conception logicielle : extension pour les animations

Animations. Les animations sont gérées par les instances de la classe Animation. Chaque plan tient une liste de ces animations, et le client graphique fait appel aux méthodes de mise à jour pour faire évoluer ses surfaces. Nous n'avons qu'un type d'animation, celui du déplacement d'un personnage. Nous avons ajouté toutes les informations permettant d'afficher le déplacement (direction et vitesse) sans dépendre de l'état. Ainsi, lorsqu'une information de mouvement parvient au plan, elle est définie dans une instance d'Animation et peut se prolonger de manière autonome. Cette animation est synchronisée avec les changements d'état grâce à la méthode *synchronisation()*, qui permet de transmettre le moment précis où un état vient de changer. C'est grâce à ces astuces que l'on peut voir les personnages se déplacer à 60 images par secondes même si le jeu évolue bien plus lentement, par exemple à 4 changements d'état par seconde.

3.4 Exemple de rendu

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



5: Exemple de rendu – Carte combat

