ENSEA
Ecole Nationale
Supérieure
de l'Electronique
et de ses Applications

# Projet League Of Dofus

**Ludovic NOEL et Paul BUGUET**

Figure 1 : Exemple du mapping du jeu League Of Dofus

# Table des matières

Introduction---------- 3

**1 Présentation Générale**---------- 3

1.1 Archétype---------- 3

1.2 Règles du jeu---------- 3

1.3 Ressources---------- 4

**2 Description et conception des états**---------- 7

2.1 Description des états---------- 7

2.1.1 Etat éléments fixes---------- 7

2.1.2 Etat éléments mobiles---------- 8

2.1.3 Etat général---------- 9

2.2 Conception Logiciel---------- 11

**3. Rendu : Stratégie et conception** ..... 11

3.1 Stratégie de rendu d'un état ..... 11

3.1.1 Les Textures ..... 11

3.1.2 Affichage de la Map ..... 11

# Introduction

Ce rapport présente le projet « League of Dofus ». Il s'agit d'un jeu vidéo tour par tour développé dans un objectif d'apprentissage de la programmation logicielle. Plusieurs domaines rentrent en compte comme : la conception, la programmation, l'optimisation et le service réseau. Il sera la mise en œuvre de savoirs provenant de plusieurs matières enseignées telles que : le génie logiciel, l'algorithme, la programmation parallèle. Un total de 112 heures de travaux encadrés sera nécessaire pour produire un produit fini. Grâce à une bonne organisation, ce jeu sera ouvert à des mises à jour et d'éventuelles modifications.

# 1 Présentation Générale

## 1.1 Archétype

L'objectif de ce projet est de réaliser un jeu tour par tour en partant de certains jeux de bases. Les jeux choisit sont Dofus et League Of Legends. Dofus est un RPG en vue isométrique qui propose aux joueurs de se déplacer librement sur des cartes qui composent le monde entier. Sa particularité est de proposer des combats au tour par tour : chaque personnage (joueur ou monstre) a ainsi une quantité de point de mouvement pour se déplacer et de point d'action pour effectuer des attaques à chaque tour.
League Of Legends quant-à lui est un RTS dans lequel 2 équipes de joueurs s'affrontent pour détruire le QG adverse : ils sont accompagnés de sbires contrôlés par l'IA qui se déplacent automatiquement vers les positons adverses.
Dans le cadre de notre projet, les règles et décors seront simplifiés : cf. figure 1 (attention le fait que les personnages prennent 4 carreaux et les sbires seulement 1 est anecdotique. Cela a pour but de montrer l'importance des personnages et non pas la taille des personnages durant les combats).

## 1.2 Règles du jeu

Deux équipes de 5 héros s'affrontent pour la destruction du QG adverse. La première équipe qui détruit le QG adverse gagne. Les héros ont la possibilité de capturer des tours de commandement qui génèrent des sbires. A chaque tour, les héros peuvent se déplacer et effectuer des actions. Le QG crée automatiquement des sbires à chaque tour, qui avancent vers le QG adverse en attaquant toute cible hostile à portée.

## 1.3 Ressources

L'affichage global de ce jeu va reposer sur 4 textures de 128 par 128 pixels. Nous avons décidé de prendre des textures de vues de dessus afin de faciliter le jeu. L'utilisation de la 3D aurait été nécessaire pour créer le mouvement des personnages. Concernant les personnages, les tourelles et les petits monstres (sbires), nous avons récupérés les sprites suivants :

Figure 2 : différents types de tourelles (QG et tour de commandement) utilisées dans le jeu.

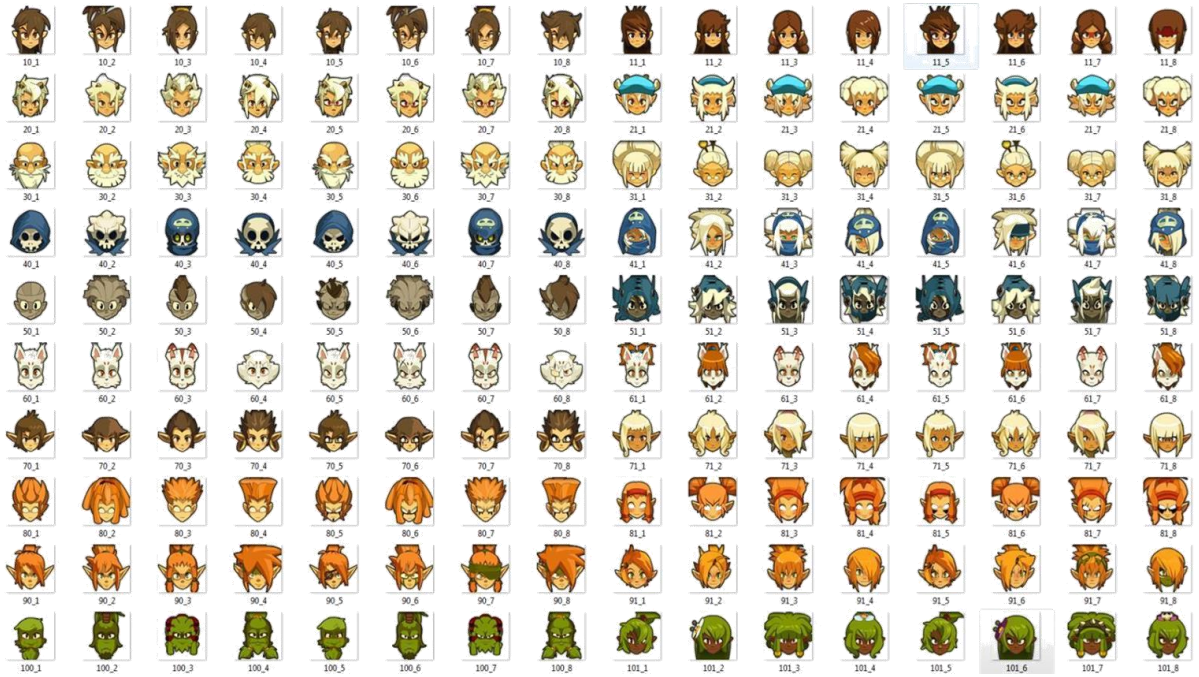


Figure 3 : Textures pour les personnages.

Chaque ligne correspond à un personnage, on a 16 textures par catégorie de personnage dont 8 pour le sexe masculin et 8 pour le sexe féminin. Pour commencer nous prenons 5 héros. Le nombre de héros sera augmenté assez facilement prochainement.
Les 5 types de personnages sont : Sram, Enu, Crâ, Iop, Sacrieur.

Cependant d'autres textures pour les tests des personnages seront utilisées, les voici :

Sram éniripsa Enu Iop Sacrieur

**Remarques** : ces sprites ne sont pas tous de la même taille. C'est pourquoi il y aura intervention d'une échelle pour chaque sprite.

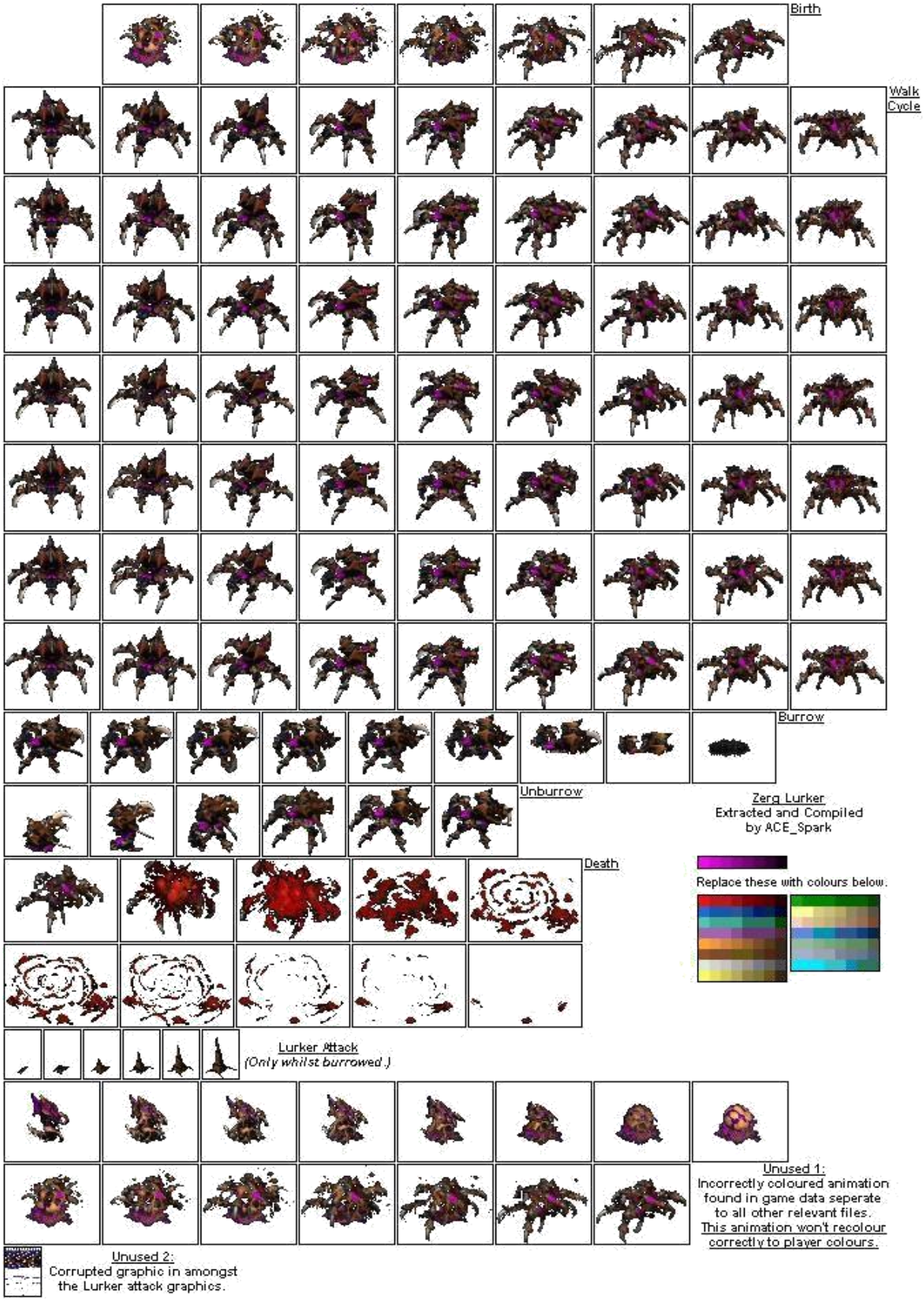



Figure 5 : Texture pour les monstres (sbires)

Possibilité de modifier la texture du sbire afin d'animer les combats.

# 2 Description et conception des états

## 2.1 Description des états

Le jeu est constitué par un seul état. Un état est formé par des éléments fixes et certains mobiles. Tous les éléments possèdent les propriétés suivantes :

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

### 2.1.1 Etat éléments fixes

La carte de combat est composée d'une grille d'éléments nommés « cases ». La taille de cette grille est fixée au démarrage de la partie (par défaut 30 cellules x 30 cellules). Cependant, il existe plusieurs types de « cases » :

**Cases « Obstacles ».** Les cases « obstacles » sont des éléments infranchissables pour les éléments mobiles. L'ensemble des cases « obstacles » seront représentés par :

- Le « rocher » qui recouvrent une case et qui bloquent également les lignes de vue. Il est donc impossible pour le joueur d'attaquer un adversaire situé derrière le rocher.
- Les « points d'eau », qui recouvrent également une ou plusieurs cases mais qui ne bloquent pas les lignes de vue. Contrairement aux rochers, les joueurs peuvent attaquer un adversaire situé derrière un point d'eau.

**Cases « bâtiment ».** Les cases « bâtiment » sont des éléments infranchissables pour les éléments mobiles qui seront représentés par :

- Les « Tours de Commandement» (TC). Ces tours sont de couleurs grises (neutre) au début de la partie. Une fois capturées, celles-ci deviennent de la couleur de l'équipe et génèrent alors des sbires pour cette dernière. Ces bâtiments ne peuvent pas être détruits.

- Les « QG » qui définissent la position du monument à détruire afin de remporter la partie. Chaque équipe a son propre « QG », de sa propre couleur (rouge ou bleu) qui génère ses propres sbires. Les QG attaquent les ennemis à portée.

**Cases « déplacement».** Les cases « déplacement » sont les éléments franchissables par les éléments mobiles. Voici les différents types de cases « déplacement » :

- Les espaces « vide », où les éléments mobiles peuvent se déplacer librement et se positionner.
- Les espaces « capture », où les éléments mobiles doivent rester un tour pour capturer les Tour de Commandement.
- Les espaces « spawn » qui définissent la position initiale de chaque personnage en début de partie, des QG et des sbires ; ainsi que le lieu de réapparition après la mort.

### 2.1.2 Etat éléments mobiles

Les éléments mobiles possèdent une direction (aucune, gauche, droite, haut ou bas), un nombre de points de mouvements (PM), des points de vie, une attaque, une portée et une position. Les éléments sont bien positionnés si leurs coordonnées sont entières. Chaque élément mobile se déplace où il le souhaite en fonction du nombre de PM dont il dispose et à condition que l'espace soit accessible. Un PM permet de se déplacer d'une case entière. Chaque élément mobile est un obstacle infranchissable qui bloque les lignes de vues pour les éléments mobiles adverses. Le fait de passer à travers un allié est autorisé du moment que deux éléments mobiles ne terminent pas leur mouvement sur la même case. L'attaque, la portée et les points de vie dépendent du type de personnage. Tous les minions auront les même caractéristiques mais les 5 types de héros auront des statistiques différentes.

**Elément mobile « Personnage ».** Cet élément est dirigé par le joueur, qui commande la direction de son personnage, ou par l'IA. On a ensuite la propriété « status » qui prend les valeurs suivantes :

- Status « vivant » : le personnage se déplace normalement et peut attaquer.
- Status « mort » : le personnage est en période de résurrection pendant un certain nombre de tours puis réapparait aléatoirement sur une case libre adjacente à son QG.

Les personnages sont représentés par des visages qui définissent leur classe (le sexe n'a pas d'importance).

**Eléments mobiles « sbires ».** Ces éléments sont commandés par IA et se dirige vers le QG adverse en attaquant tout ennemi à portée.
Ces éléments peuvent se déplacer comme les personnages et ont 2 status :

- Status « normal » : cas le plus courant où le sbire peut avancer et attaquer.
- Status « mort » : où le sbire a été tué. Il respawn au tour suivant à sa position initiale (QG ou TC).

Les sbires sont représentés par un modèle qui n'a pas d'importance autre qu'esthétique.

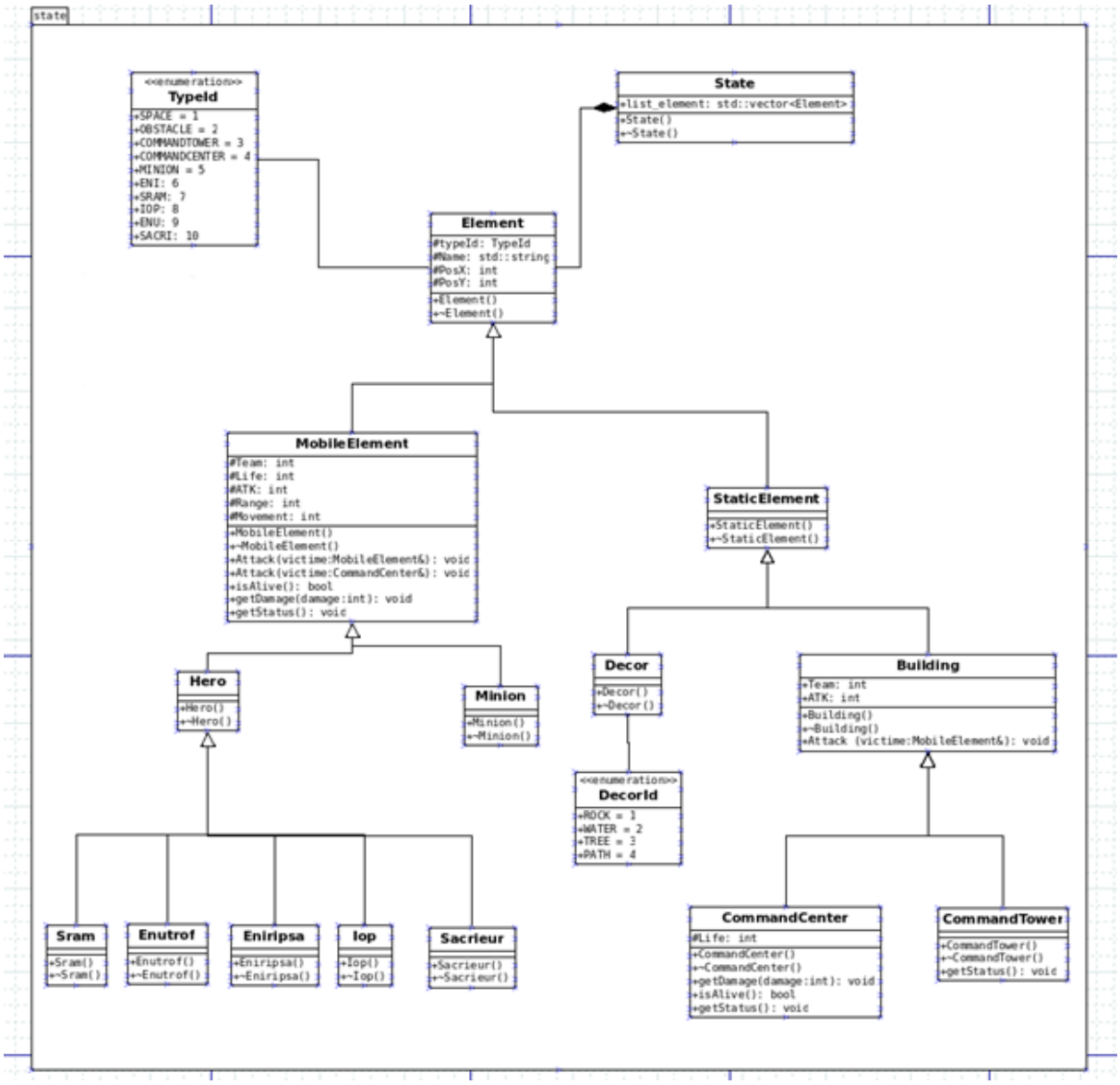
### 2.1.3 Etat général

A l'ensemble des éléments statiques et mobiles, nous rajoutons les propriétés suivantes :

- compteur résurrection : le nombre de tours restants avant la résurrection d'un personnage ou d'un sbire.
- Compteur de niveau : qui permet d'afficher le niveau de chaque personnage.
- Compteur de Point de vie: affiche la vie de chaque personnage et QG afin de connaitre l'avancée du jeu.
- l'état « State » contient tous les éléments présent à l'instant t du jeu. Cette classe dépendra donc de l'avancement de la partie.

## 2.2 Conception Logiciel

Le diagramme des classes pour les états est le suivant :



- La classe State contient une liste contenant tous les éléments du jeu. Elle peut aussi créer et détruire un état. Cette classe permet de savoir dans quel état du jeu, on se trouve.
- La classe Element contient les attributs de bases de chaque Element. Les éléments possèdents des Ids qui permettent de les différencier entre eux ainsi qu'une position et un nom. Elle possède également deux classes filles : les MobileElement et les StaticElement.
- Les MobileElement désignent les éléments mobiles, c'est-à-dire les héros et les minions. Elle possède les attributs communs à ces éléments ainsi que des methodes leur permettant d'attaquer d'autres MobileElement et des CommandCenter (classe que nous détaillerons plus par la suite).
- Les Heros sont divisés en 5 sous-classes, qui correspondent aux différentes classes de héros qu'il est possible de jouer : chaque classe à un Id RaceId (rem : dans Dofus il n'y avait pas, au début, de classes à proprement parlé, mais uniquement0 des races).
- Les StaticElements représentent les éléments qui ne peuvent pas se déplacer. Il s'agit du décor et des Building.
- La classe Decor contient les Id des différents type de décor (eau, rocher, arbre)
- La classe Building contient les CommandCenter (le bâtiment à abattre pour remporter la partie) et les CommandTower (les tours à capturer pour générer plus de minions). Ces 2 bâtiments peuvent attaquer les ennemis à portée, mais seuls les CommandCenter peuvent recevoir des dégâts.

# 3 Rendu : Stratégie et conception

## 3.1 Stratégie de rendu d'un état

Dans l'état du jeu, le joueur doit pouvoir être informé de l'ensemble des variables. Le rendu graphique permet d'afficher les variables de l'état faisant ainsi le lien entre les données et l'affichage. On vient donc charger un rendu graphique qui dépend entièrement de l'état. A chaque déplacement ou mort de personnage, on charge le rendu graphique correspond à ces actions.

Chaque personnage peut voir la totalité de la « carte » et l'emplacement des joueurs, des « Towers », des « minions » et des « Center ». Le joueur peut donc réfléchir à ses actions bien avant son tour.

### 3.1.1 Les Textures

Pour afficher les éléments, on vient tout d'abord charger leur texture via la classe Textures. C'est à cette étape qu'interviennent les sprites définis précédemment.

-Classe « Textures » : on vient charger la texture d'un élément en lui faisant correspondre un Sprite suivant son statut, sa position et son « TypeId ». On obtient alors le Sprite de l'élément. Un élément peut donc avoir jusqu'à 4 sprites. Les héros auront un sprite pour chacun de leur déplacements (haut, bas, droite, gauche).

Une fois que les textures sont chargées, il nous reste à pouvoir les afficher, ceci est réaliser dans la classe « View ».

### 3.1.2 Affichage de la Map

L'élément central du jeu est la carte, parce qu'elle permet aux joueurs de définir ses actions et sa stratégie. Pour créer la carte, nous utilisons le logiciel « Paint.net » qui nous permet d'éditer une image. L'image au format PNG est ensuite traitée par la classe Tile qui est ensuite utilisé par la classe TileMap (voir View.cpp) qui permet de dessiner la carte en tant que Background.

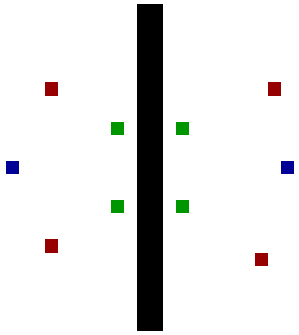
Notre map est définie sur 30 pixels de largeur et 30 pixels de hauteur.

**-**Classe « Tiles » : avec un argument <<sf :: Image>>. Elle permet de créer un pointeur sur un tableau à deux dimensions. Ce tableau correspond aux numéros des tuiles associées à chaque pixel de l'image.

Les tuiles sont positionnées suivant le placement des frontières. Il nous faut des frontières de deux pixels soit deux tuiles pour créer celle-ci. Une autre fonction de Tiles renverra à partir de la même image, la position des nombres permettant d'afficher les héros, les Tours et les minions. Le fait d'utiliser une fonction traitant une image permet de modifier la carte sans avoir à modifier tout le code source. Cela permet une facilité et une rapidité d'exécution. On récupère par la même occasion les positions des tours et des héros ainsi que leurs TypeId.

-classe « TileMap » (issu de View.cpp): est un élément généré à partir du tableau de Tiles (transmis précédemment). Cette fonction permet de spécifier la texture de la map. Cet élément est drawable afin de pouvoir l'afficher via la librairie SFML. On peut alors modifier la texture de la map avec facilité. Il suffit de créer une image png avec des bords et

un centre à nos souhaits et de l'intégrer à la fonction TileMap. La classe TileMap nous a été donnée de base, on a alors testé et validé. Cette classe n'est pas sur l'UML car elle est indispensable pour la bonne utilisation du rendu graphique.

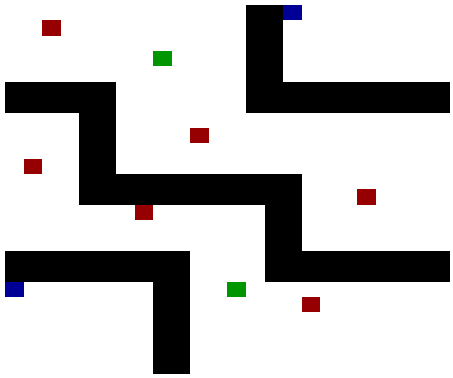


**Carte du niveau envoyée par paint .net**


**Rendu du niveau crée sur Paint.**

## Test numéro 2 :

**Carte du niveau 2 envoyée par paint .net**



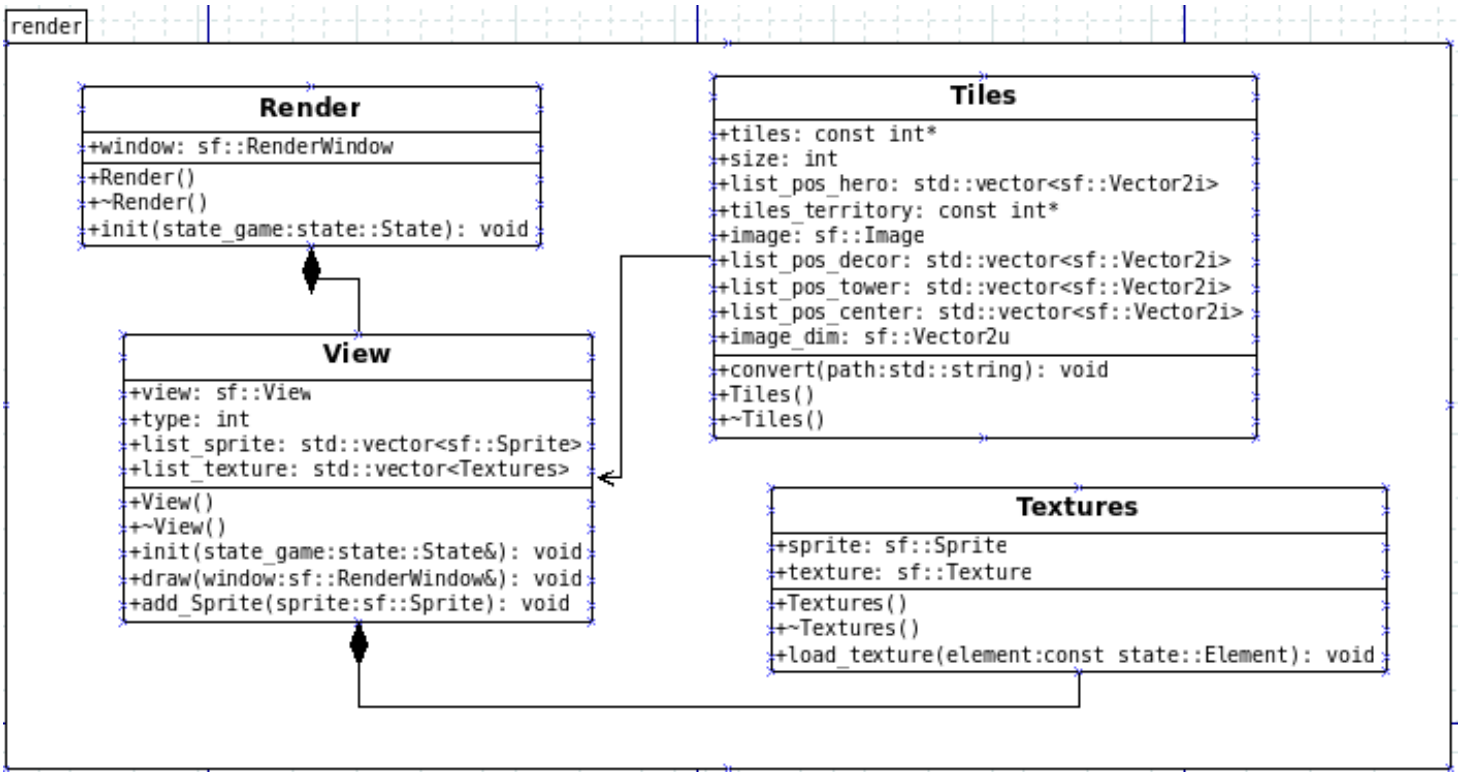
**Rendu du niveau 2 crée sur Paint.**

### 3.1.3 Les views

Les views permettent de divisé l'écran et donc de séparer un affichage. Elles sont générées à partir de la classe View. Elles correspondent à des couches qu'on superpose.

La méthode add_sprite() permet d'ajouter à l'affichage les sprites sélectionnés. Cela peut concerner tous les éléments mobiles et statiques.
La méthode draw permet « d'écrire » à l'écran pour avoir un rendu visuel des Sprite en question.
La méthode init permet d'initialiser la « couche », la remettre à « zéro ».
Cette classe a comme attributs : une liste de sprite, une liste de texture un type et un view.
Les listes permettent d'aller choisir le sprite correspondant à l'élément en question.

Dia du Package Render



# 4 Règles de changements d'état et moteur de jeu :

## 4.1 Changements extérieurs :

Les changements extérieurs sont provoqués par des commandes extérieures, comme l'appui sur une touche ou l'utilisation de la souris :

- Cliquer sur « lancer la partie » permet de fabriquer un état initial à partir d'un fichier.
- Commande « Déplacement » : cette commande prend en paramètre un personnage et la direction associée. Un personnage contrôlé par l'IA ne se déplace que d'une case à la fois, mais peut réutiliser cette commande tant qu'il lui reste des points de mouvement ( voir plus tard).
- Commande « Attaquer » : cette commande prend en paramètre un personnage ou un bâtiment allié et un autre élément (personnage adverse ou QG adverse) et le nombre dégât à ,elle ne peut être utilisée qu'une seule fois par tour par entité.

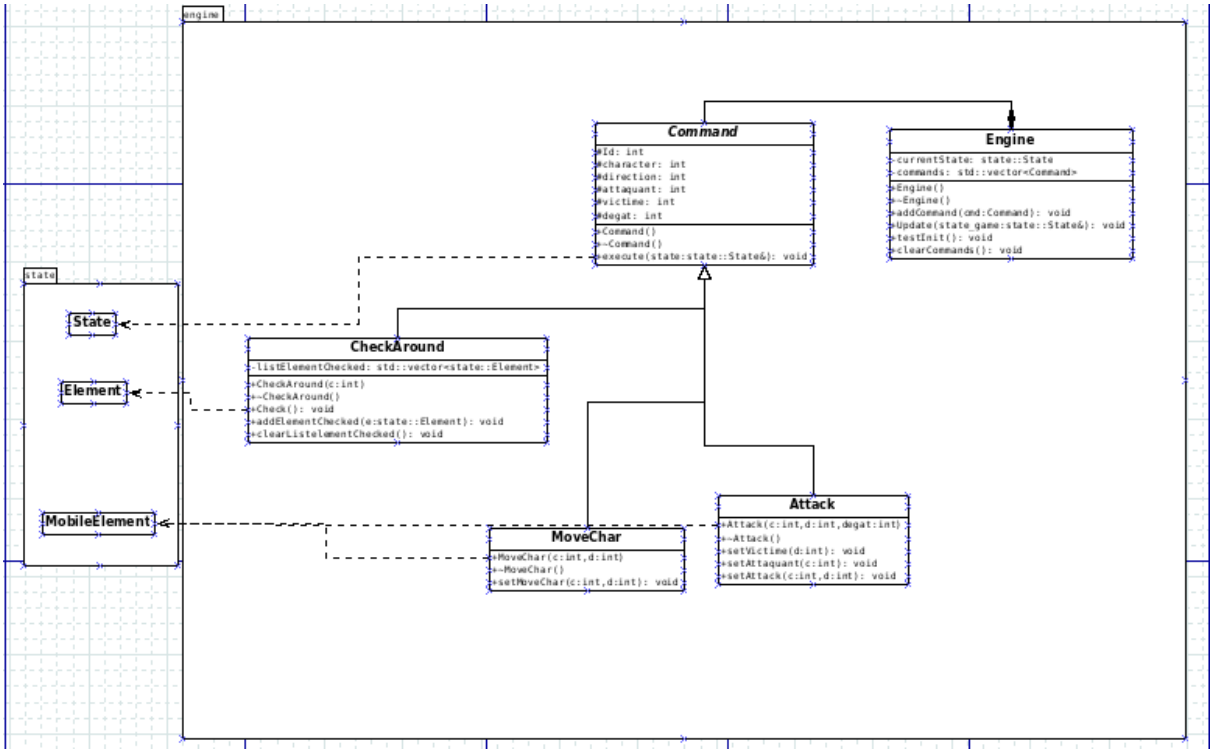
## 4.2 Changements autonomes :

Les changements autonomes représentent le déroulement d'un tour :

1. Appliquer les règles de capture pour les tours :
    a. Au début de la partie les tours sont neutres
    b. Au début de chaque tour de chaque joueur elles vérifient les personnages présents dans leur zone de capture : s'il n'y a que des personnages du joueur dont c'est le tour, la tour est capturée par cette équipe
    c. Une tour capturée suit immédiatement les règles de génération de minions et d'attaque (cf. étapes suivantes)
    d. Au début de chaque tour du joueur adverse, les tours vérifient les personnages présents dans leur zone de capture : s'il n'y a que des unités adverses elles sont alors capturées et changent donc d'équipe.
    e. Dans le cas où il y a au moins une unité adverse dans la zone de capture pendant le tour de l'équipe possédant la tour, cette dernière passe dans l'état « contestée » : elle ne peut plus générer de minions mais peut toujours attaquer pendant son tour (à noter que dans ce cas de figure, si aucune unité allié ne rejoint la zone de capture ou que l'unité adverse n'est pas éliminée, la tour sera alors capturée au début du prochain tour adverse). Au final les tours peuvent attaquer les éléments mobile présent dans leur zone de tir.
2. Un héros réapparait un tour après sa mort, à sa position initiale de la partie.
3. Si le compteur de sbire de bâtiment n'a pas atteint sa capacité maximale, en rajouter un dans la zone de spawn de ce bâtiment.
4. Appliquer les règles de déplacement pour les héros (gauche , bas , droite , haut , test de collision ).
5. Appliquer les règles d'attaque pour les héros et les bâtiments.
6. Retour à l'étape 4 tant que tous les déplacements et/ou attaques voulues n'ont pas été réalisés (il n'est pas nécessaire d'utiliser tous les points de déplacement ni d'attaquer pendant un tour) créer une façon de finir son tour.

## 4.3 Conception logiciel :

Le diagramme des classes pour le moteur du jeu est représenté dans la figure suivante :




- La classe Engine est le cœur de engine : elle contient une liste de commandes qu'elle exécute à l'aide de la méthode Update().
  Chaque commande est stockée dans la liste « Command ». On retrouve le principe d'utilisation d'une FIFO. La première commande enregistrée dans la liste sera la première exécutée par l'Update de l'état.
- L'enable_render permet d'autoriser le rendu à s'actualiser avec les différentes nouvelles valeurs (position, vie, etc..). Après chaque exécution de commande on met enable_render à 1 ce qui permet d'actualiser le rendu.
- De même pour l'engine avec son enable_engine.
- La méthode Testlinit() permet de créer toutes les commandes qui sont ajoutées à la liste Command et qui seront donc exécutées à chaque appuie de « B ».
  Après l'exécution d'une commande on l'écrase de la liste afin d'exécuter toujours la commande avec l'indice 0 de la liste.
- Chaque classe fille de Command possède son propre Id afin de pouvoir les différencier lors de leur exécution :
  - La classe MoveChar permet de déplacer un personnage
  - La classe Attack permet à un élément d'attaquer.
  - La classe CheckAround permet de vérifier et de lister la présence d'autres éléments autour de l'élément choisi. Cette classe permet donc de limiter ou non les mouvements possible du joueur. On rappelle que les joueurs ne peuvent pas s'arrêter sur la même case, du coup il nous faut prendre en compte les positions des autres joueurs.