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

Projet League Of Dofus

Ludovic NOEL et Paul BUGUET

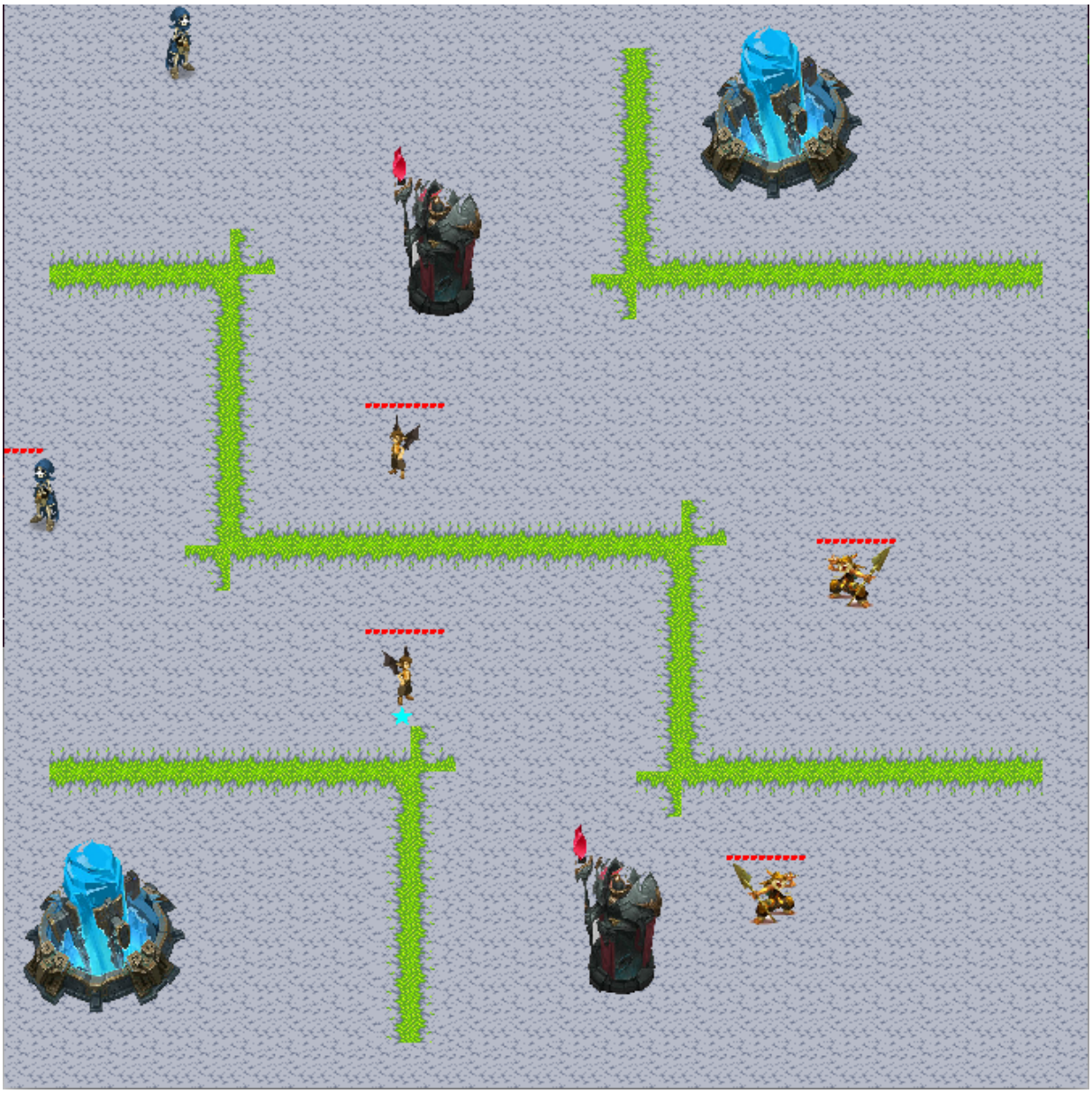


Figure 1 : Exemple du mapping du jeu League Of Dofus

Table des matières

Introduction

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.

1.2 Règles du jeu

Deux équipes de 3 héros s'affrontent pour la destruction du QG adverse. La première équipe qui détruit le QG adverse gagne. Chaque équipe détient une Tour qui peut attaquer et peut être détruite. 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

L'affichage glob ser sur plusieurs textures de 28 par 28 pixels. Nous avons décidé d s des jeux d'origine. L'utilisation de la 3D aurait été nécessaire pou des personnages.
Concernant les ourelles et les petits monstres (sbires), nous avons récupérés les sprites suivants :

Figure 2 : Tour de commandement utilisée dans le jeu.

Figure 3 : QG utilisé dans le jeu (à détruire pour déterminer l'équipe gagnante)

Remarques : Le fait de mettre un filtre de couleur bleu ou rouge permettra de distinguer les 2 équipes.

Figure 4 : Textures pour les personnages et le cas échéant de leur mort.

Chaque ligne correspond à un personnage. Le premier personnage est le SRAM, ensuite l'ENI, puis l'ENUTROF et le IOP et pour finir le SACRI.
On a donc 11 textures dans la catégorie personnage. 2 textures par personnage, déplacement gauche, déplacement droit et une texture pour représenter leur mort.

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

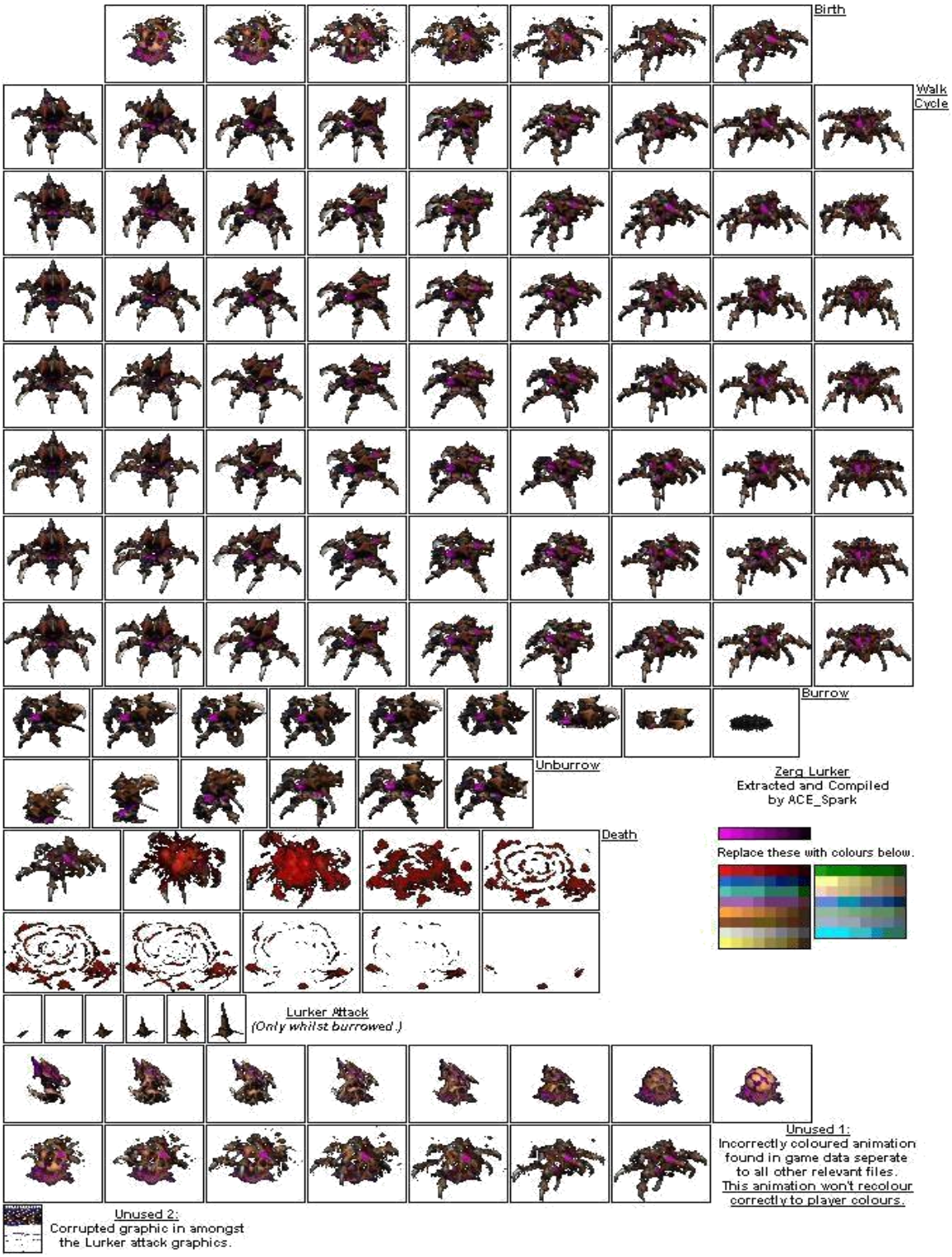


Figure 5 : Texture pour les monstres (sbires)

Figure 6 : Textures pour améliorer le rendu visuel du jeu

Le cœur permet d'afficher le nombre de points de vie restant à tous les éléments du jeu (héros, minions, tours, QG).

L'étoile permet de montrer à qui est le tour.
Si le Sram de l'équipe 1 doit jouer, il se verra afficher cette texture sous le personnage.

La dernière texture permet d'afficher les cases disponibles à chaque héros pour se déplacer et/ou pour attaquer.
La case rouge représente les déplacements disponibles et la case bleu la distance d'attaque possible.

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 24 cellules x 24 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). Chaque équipe détient une tour, qui aura la couleur de son équipe. Elles peuvent attaquer des héros adverses. Cependant ces dernières peuvent aussi être détruites.

- 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 « spawn » qui définissent la position initiale de chaque personnage en début de partie, des QG et des sbires.

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 mort jusqu'à la fin de la partie.

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 sbire.

- 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 :

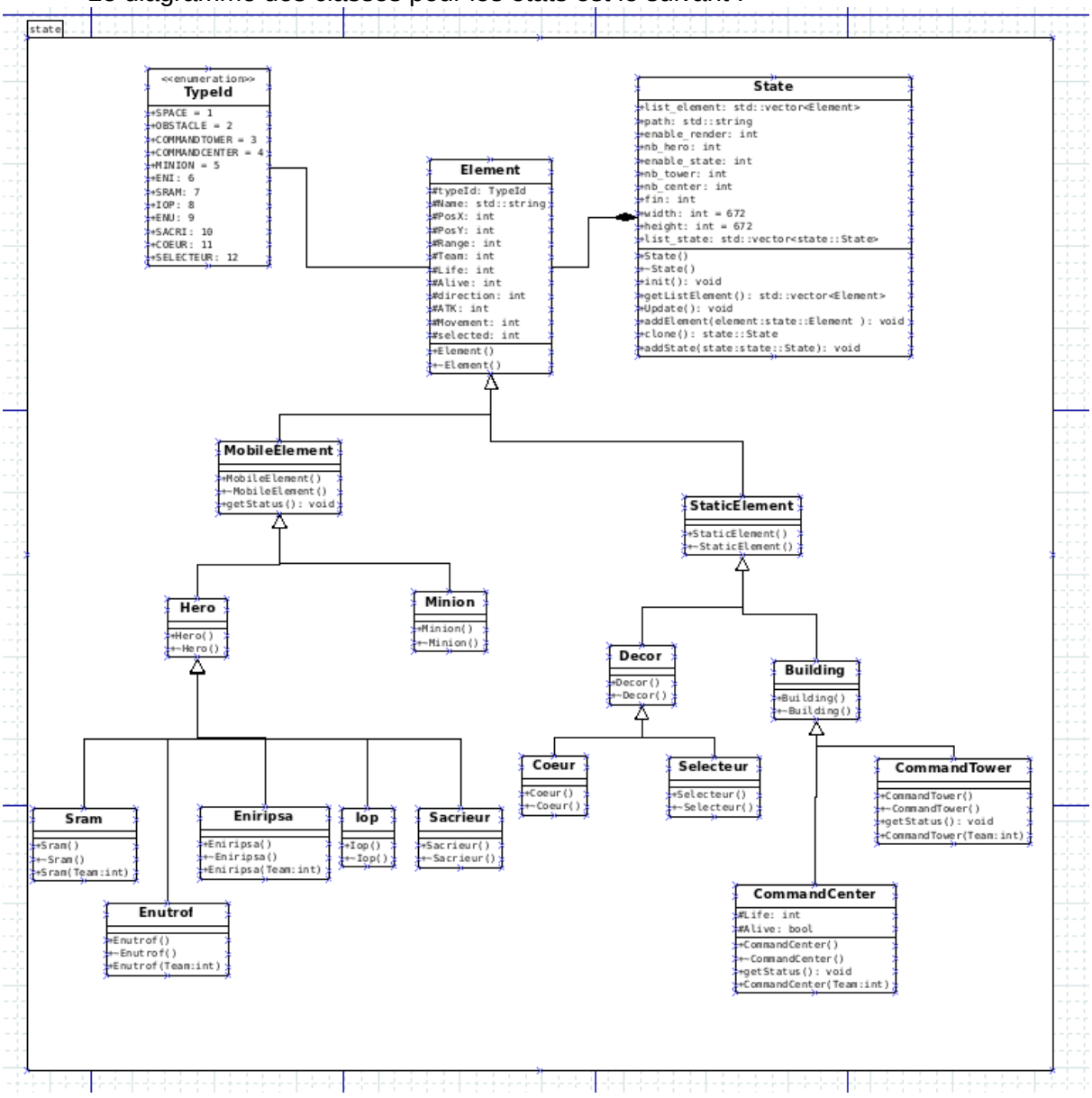


Figure 7 : diagramme des classes State.

- 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.

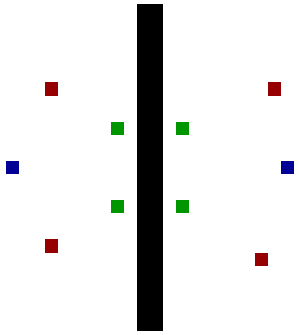
Notre map est définie sur 30 pixels de largeur et 30 pixels de hauteur.

-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.

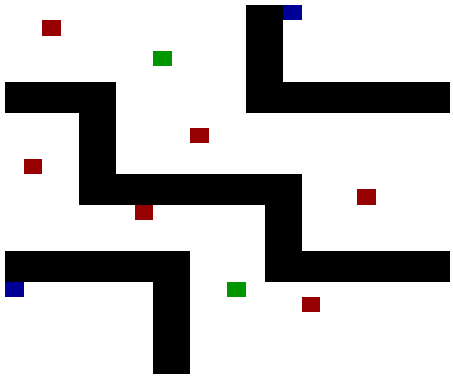


Carte du niveau envoyée par paint .net

Rendu du niveau crée sur Paint.

Test numéro 2 :

Carte du niveau 2 envoyée par paint .net



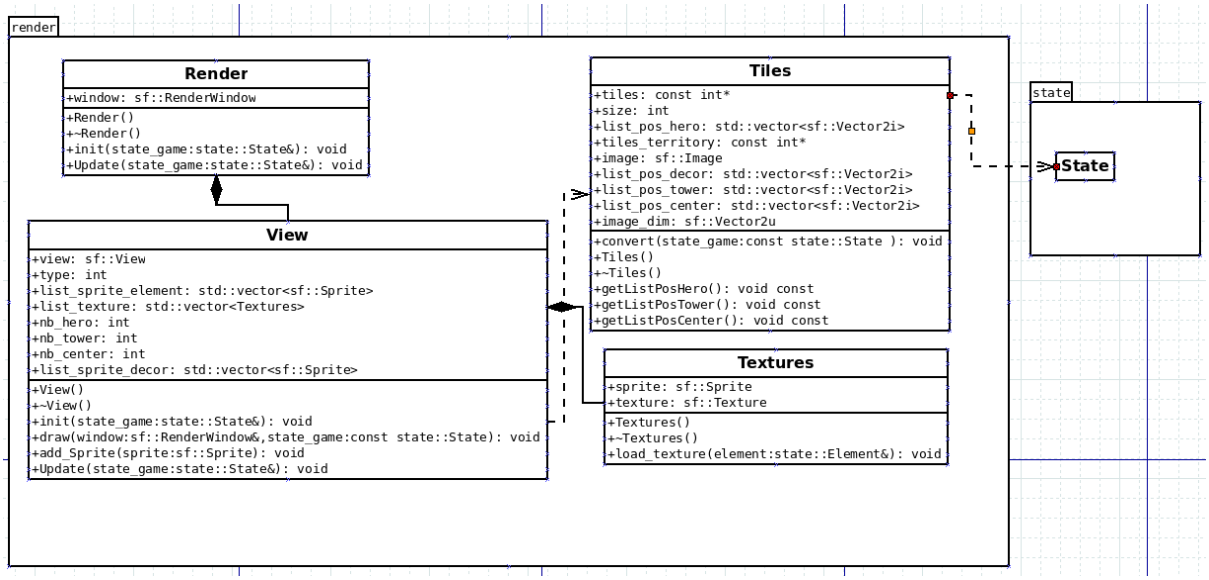
Rendu du niveau 2 crée sur Paint.

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 :

- 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.
- 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 de dégât à infliger. Elle ne peut être utilisée qu'une seule fois par tour par entité.

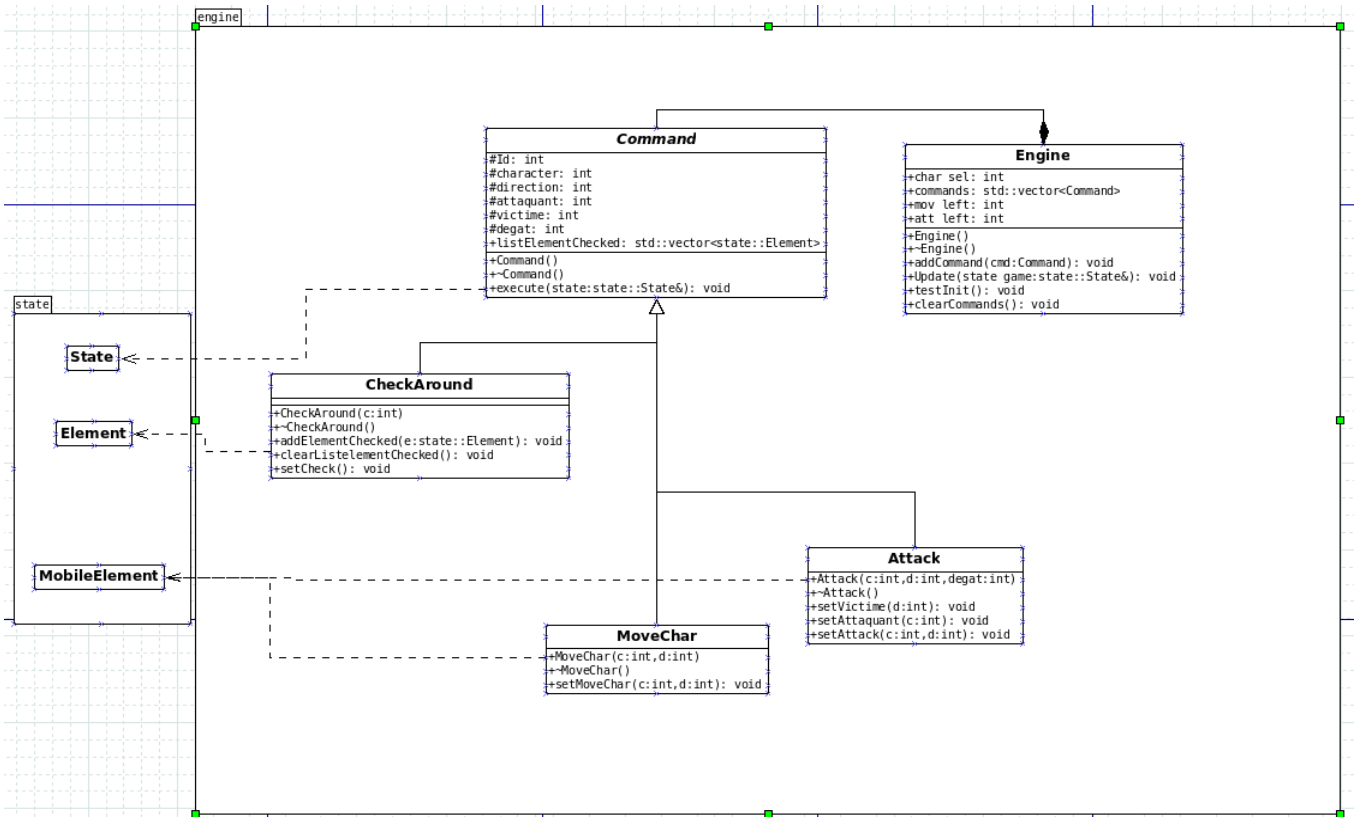
4.2 Changements autonomes

Chaque équipe est constituée de 3 héros. Ces héros sont définis par leur Id puis inséré dans une liste. 3 héros appartiennent à l'équipe 0 et 3 autres à l'équipe 1. Pour faire simple, les 3 héros les plus proches du QG en haut sont dans l'équipe 0 et les trois autres dans l'équipe 1.
Les changements autonomes représentent le déroulement d'un tour :

1. Chaque héros a le droit à 3 déplacements et 1 attaque par tour.
2. Appliquer les règles de déplacement sur le premier héros de la liste :
    a. Le héros bouge d'une case
    b. Test de collision
    c. Si le test est positif, le héros ne bouge pas, sinon celui-ci obtient sa nouvelle position
    d. On réduit le nombre de déplacement de 1.
    e. Envoie d'une notification à l'état afin d'autoriser la mise à jour de celui-ci (Pattern Observer)
    f. Mise à jour du rendu via l'état et le Pattern Observer
    g. Tests sur le nombre de déplacement restants.
    h. Si le nombre de déplacement restant est nul, on passe à l'étape 3 sinon on retourne à l'étape 2.
3. Appliquer les règles d'attaque sur le premier héros de la liste.
    a. Le héros attaque
    b. Test d'attaque
    c. Si le héros attaque un partenaire ou lui-même, cela ne fait rien, sinon le héros attaque un adversaire et lui réduit ses points de vie.
    d. Réduction du nombre d'attaque de 1. Le joueur n'a plus le droit d'attaquer pour ce tour.
    e. Envoie d'une notification à l'état afin d'autoriser la mise à jour de celui-ci (Pattern Observer)
    f. Mise à jour du rendu via l'état et le Pattern Observer
4. Test si l'adversaire a encore des points de vie, sinon celui-ci meurt.
5. Un héros mort ne peut plus jouer jusqu'à la fin de la partie.
6. Fin du tour du premier héros de la liste.
7. On refait ces 6 étapes pour les autres héros restant.
8. On regarde les status des héros de la liste
    a. Si tous les héros d'une équipe sont mort alors l'adverse gagne et fin de la partie.
    b. Sinon on retourne à l'étape 1.

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.
- 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.

5 Intelligence Artificielle

5.1 Stratégies

5.1.1 Intelligence aléatoire

La stratégie de jeu utilisée ici est la même pour les 6 héros : un personnage a le droit à 3 déplacements et 1 attaque.

On passe en revue chaque personnage de la première équipe, puis de la seconde. Chaque personnage se déplace 3 fois aléatoirement puis attaque.
Cependant des tests de collision sont faits afin que le personnage ne s'évade pas de la map du jeu. En effet si le test de collision est positif, le personnage va perdre un point de mouvement et ne bougera pas.

Après avoir effectué tous ses déplacements, le personnage attaque aléatoirement un autre héros.
Si l'attaque se dirige vers quelqu'un de son équipe ou lui-même, l'attaque n'aboutit pas. En revanche si l'attaque se dirige vers un adversaire, celui-ci l'attaque et réduit ses points de vie.
Lorsqu'un héros n'a plus de points de vie, celui-ci meurt et ne joue plus avant la fin de la partie.

5.1.2 Intelligence basée sur des heuristiques

La stratégie de jeu reprend ici les bases de la stratégie pour l'intelligence aléatoire, en y ajoutant un ensemble d'heuristiques pour offrir un comportement meilleur que le hasard pur.

Tous les personnages suivent le comportement suivant :
- Si le personnage n'a pas attaqué et a tous ses points de mouvements, il va se déplacer vers l'adversaire le plus proche.
- S'il le personnage a utilisé tous ses points de mouvements et est trop éloigné d'un adversaire, celui-ci n'attaque pas.
- Si l'adversaire est accessible, le personnage va l'attaquer.
- Si le personnage a déjà attaqué, il va chercher à s'éloigner du personnage adverse le plus proche.
- Si plusieurs ennemis sont à portée, le personnage va privilégier celui qui est le plus proche.

La méthode heuristique proposée est mise en œuvre en utilisant un système de score :
Avant chaque action, des scores sont attribués à chaque action possible en fonction du contexte (attaque ou non, point de mouvement restant, vie et position des personnages ennemis). Ensuite l'action ayant le score le plus important est effectuée.

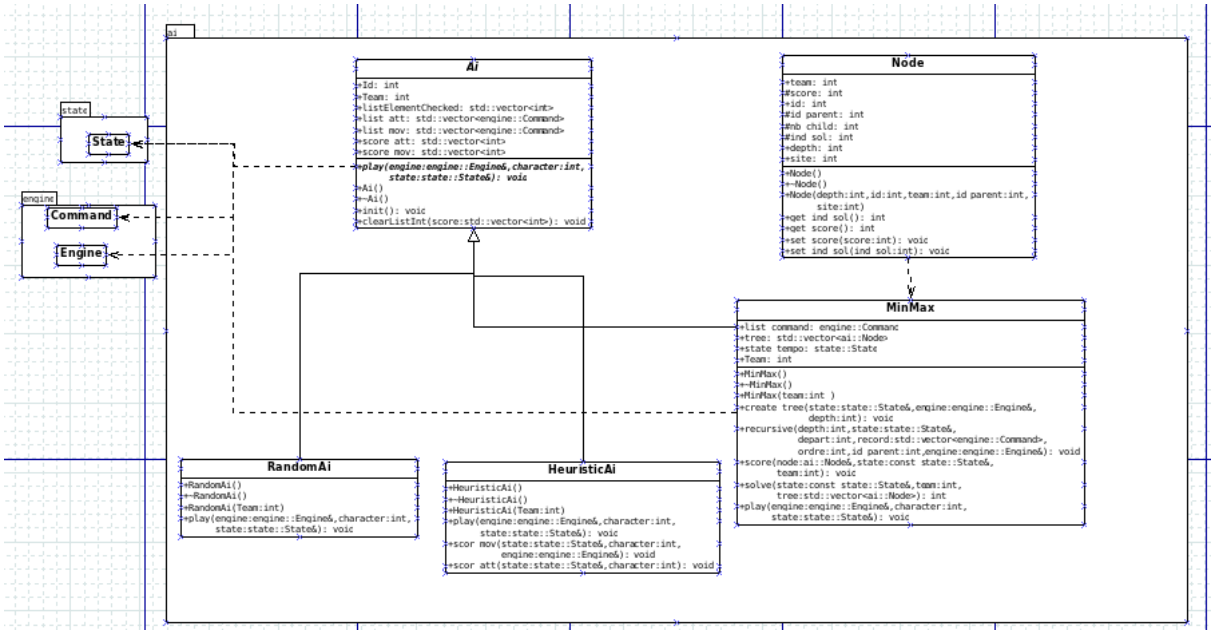
En effet, la priorité sera donnée à l'attaque.
Si l'attaque est encore possible, le personnage se déplace vers l'adversaire le plus proche ou l'attaque si celui-ci est déjà assez proche.
En revanche si l'attaque a déjà été faite, le personnage s'éloigne de l'adversaire afin d'éviter de se faire attaquer.
Dans ces cas ci-dessus le personnage en question cible toujours l'adversaire le plus proche afin de le tuer ou éviter de mourir.

Concernant les mouvements, la priorité sera donnée en fonction des déplacements nécessaire. Si l'adversaire est trop éloigné, notre personnage va s'en rapprocher en donnant un score plus élevé aux déplacements qui l'aideront à se rapprocher.
En revanche, si ce dernier doit s'en fuir, on donnera un score plus élevé aux déplacements qui agrandiront la distance entre lui et son adversaire.

5.1.3 Intelligence basée sur la recherche dans un arbre

Pour cette Ai nous allons utiliser un arbre. Pour cela, nous allons devoir travailler dans un état qui n'est pas l'état du jeu, d'où la nécessité de la méthode clone() dans la classe State. Pour faciliter la navigation dans cet arbre, nous utilisons la fonction Undo qui permet de défaire une commande.
L'algorithme MinMax est utilisé afin de résoudre le problème : le résultat recherché est de maximiser les points de l'équipe joueuse, et de minimiser les poitns de vie de l'équipe adverse. Le joueur 1 va donc chercher à maximiser les points de vie de chacun de ses héros, alors que que le joueur 2 va essayer de minimiser cet écart.
En prenant en compte toutes les possibilités, l'arbre va vite devenir beaucoup trop grand pour la machine, c'est pour cela que nous nous limitons donc aux quatres déplacements (haut, bas, gauche et droite) ainsi qu'à l'attaque (la taille de l'arbre est reglable).
Enfin, nous utilisons également la possibilité d'effectuer un rolleback (ie un retour en arrière). Le rolleback se base surtout sur la méthode undo() qui permet d'annuler l'exécution d'une commande, en effectuant l'inverse de la commande : le personnage se déplace à gauche s'il sétati déplacé à droite, rend des points de vie s'il en avait enlevé et ressuscite même un adversaire si ce dernier avait succombé aux attaques précédentes.

5.2 Conception logiciel



Le diagramme des classes pour l'intelligence artificielle est présenté ci-dessous.

La classe AI est composée du constructeur AI et de son destructeur, ainsi que de la méthode init et play.

La méthode Init permet de créer les commandes possibles suivantes :
- 4 commandes pour le déplacement (haut, bas, gauche et droit)
- 6 commandes pour attaquer (héro1, héro2, héro3, héro4, héro5, héro6).

Les commandes déplacements seront mises dans une liste list_mov et les commandes ataques dans la liste list_att.
On a 6 commandes attaques car il y a possibilité d'attaquer 6 héros. La confirmation de l'attaque dépend du héros qui attaque et de son numéro d'équipe.
Que l'attaque soit validée ou non, le héros en question perd son point d'attaque.

La méthode play prend en argument un état, un héros et un engine.
A travers cette méthode nous retrouvons le random qui permet de créer ce sentiment d'IA aléatoire.
La première étape est de tester si les points de mouvement du héros sont nul ou non. Si ses points de mouvements sont non nuls, l'IA choisit aléatoirement une commande dans « mov_list » et l'ajoute à l'engine. On fait cela pour les 3 points de mouvement et nous faisons de même avec l'attaque. Si le héros a un point d'attaque, l'IA choisit une commande dans « list_att » et l'ajoute à l'engine. Sinon le héros a déjà attaqué.

Les méthodes « score_mov » et « score_att » dans la classe HeuristicAi sont les méthodes attribuant des scores aux actions de déplacement et d'attaque en fonction du contexte.

La partie de l'exécution et de l'update sont présentés précédemment.

Les classes filles de la classe AI permettent de créer une AI avec différents niveaux de réflexions que l'on peut appliquer à chaque personnage :

- RandomAI : Intelligence aléatoire
- HeuristicAi : Intelligence heuristique
- DeepAi : une intelligence plus poussée qui utilise la recherche dans un arbre

La recherche dans un arbre étant plus poussée, nous allons la décrire plus en détail :

La construction de l'arbre se fait grâce à la methode create_tree, qui vient dans un premier temps push l'origine de la racine. Nous faisons ensuite appel à une fonction recursive pour créer les différents nœuds. Si l'action n'est pas ou plus possible (comme un déplacement après utilisation de tous les points de mouvement), la création de nœud ne se fait pas, ce qui permet de réduire considérablement la taille de l'arbre. Pour la récursivité, nous créons un nœud en utilisant la commande clone() et nous le scorons à l'aide de la fonction score(), puis le poussons dans l'arbre et recommençons. Tous les nœuds sont donc scorés, score qui correspond à la différence de point de vie entre les deux équipes (la soustraction est inversée en fonction de l'équipe qui joue).
Une fois l'arbre construit il ne reste plus qu'à le résoudre. Cette résolution se fait étape par étape, de haut en bas, à l'aide de la méthode solve() : nous regardons le score de tous les derniers nœuds puis suivant l'équipe dont c'est le tour, nous prenons le min ou le max et nous remontons jusqu'à la profondeur 1 qui va nous donner l'indice du coup à jouer.