le cnam

Licence ACSID en alternance 2015-2016

Rapport de projet

LUMSAAR

Alternants :

GRAFFION Iliès - DUMANOIR Dhivya - LESBROS Ulysse - MARY Killian

Enseignant :

AUBONNET Tatiana

Sommaire

1. Introduction

L'objectif de ce projet est de développer un jeu-vidéo stratégie sur plateau en 2D isométrique en Java du même type que « **Dofus** » ou « **Age of Empire**». Le jeu nommé **Lumsaar** consiste à faire jouer un utilisateur contre un autre joueur ou bien une Intelligence Artificielle. L'ambiance de ce jeu sera médiévale et fantastique.

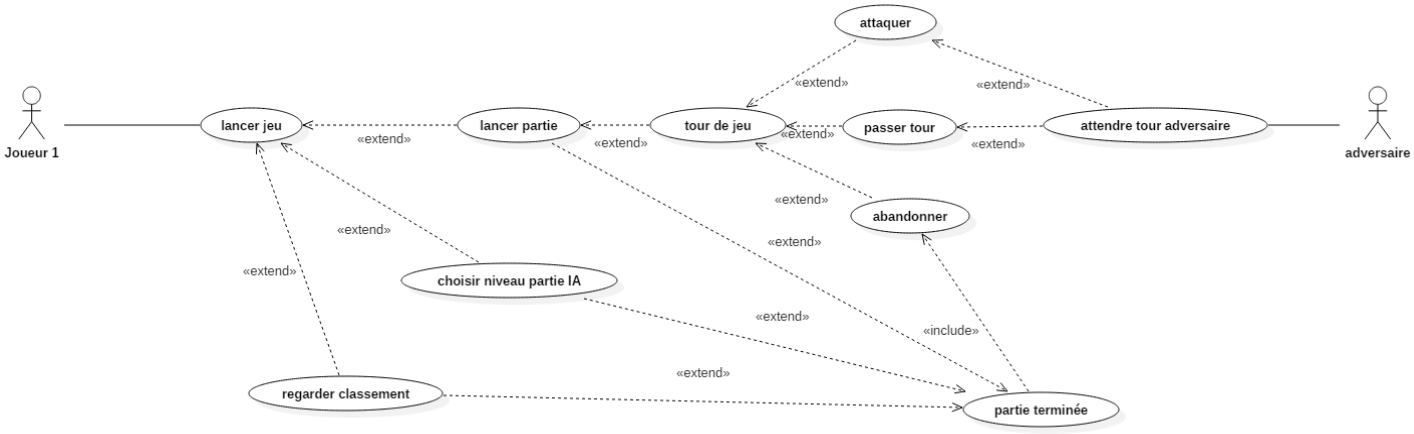
Ce projet suivra principalement les points suivants:

- Concept du jeu d'échec
- Tour par tour
- Chaque pion a une propriété spécifique (Corps à corps, longue portée, kamikaze etc...)
- Chaque action coûte des points d'actions (déplacement, frapper un pion adverse etc…)
- Lors d'un nouveau tour, il est possible de voir des bonus apparaître sur le plateau (Soins, dommages augmentés, défense etc…)
- Chaque joueur joue à son tour et déplace le pion de son choix
- Lorsqu'un des deux joueurs n'a plus de pion, la partie est terminée.

2. Conception UML

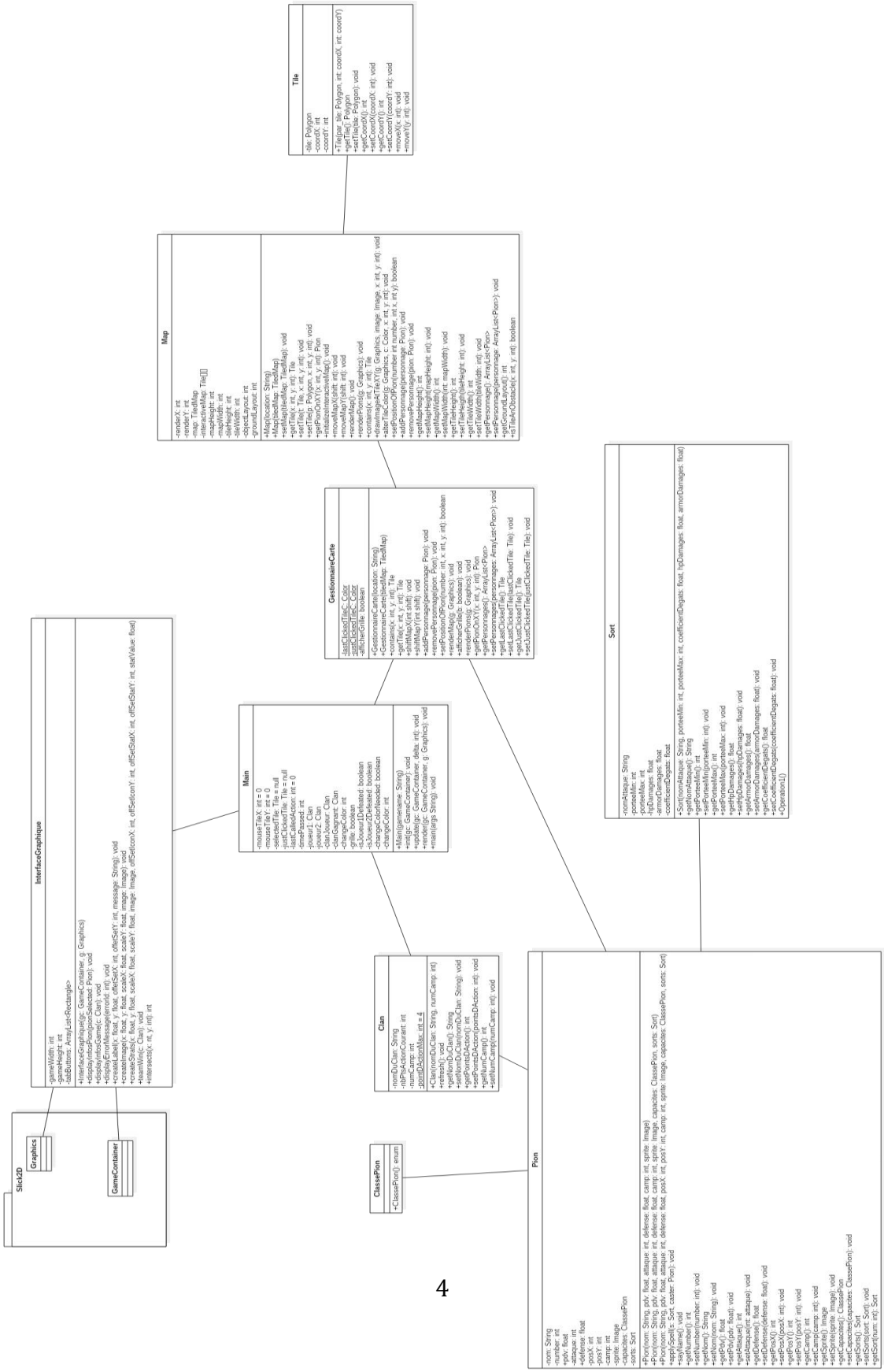
2.1 Diagramme de cas d'utilisation

A l'aide de l'outil Star UML, on obtient le diagramme de cas d'utilisation suivant. Arrivé sur le menu du jeu vidéo, l'utilisateur peut soit lancer une nouvelle partie, choisir le niveau (de l'IA) de la partie ou afficher le classement des meilleurs joueurs.



2.2 Diagramme de classe

Voici le diagramme de classe de notre jeu vidéo, agrémenté des quelques classes de la librairie Slick2D utilisées.



3. Implémentation

3.1 Liste des technologies utilisées

- Partie programmation : **Java**

  Java est un langage de programmation orienté objet appris et approfondis durant notre cursus au sein du CNAM.

- Interface graphique : **Slick2D**

  Slick2D est une librairie open source qui permet de créer très simplement un jeu 2D en java. Différentes fonctions intégrées à la libraire peuvent aider à développer rapidement les interfaces graphiques.
  Sources : http://slick.ninjacave.com/

- Partie illustration : **Paint.net**

  Pour cette partie, nous avons utilisé Paint.net qui est un logiciel libre et gratuit. Il nous a servi à modéliser des illustrations ou icones visibles dans le jeu.

- Partie modélisation : **StarUML**

  Nous avons eu recours au logiciel StarUML pour la modélisation, qui est un également logiciel libre. C'est un logiciel très complet qui répond à nos besoins en termes de modélisation et qui permet d'avoir une bonne vision globale du projet.

3.2 Conception et affichage du menu

Dans cette partie, nous verrons comment nous avons implémenter la partie graphique des menus et de l'interface globale du jeu.

```
package vue;

    import gestionnairePersonnages.Clan;


public class InterfaceGraphique {
    //Slick2D objects
    private Graphics g;
    private GameContainer gc;
    //Useful variables
    private int gameWidth,gameHeight;

    // Image/Rectangle tab
    private ArrayList<Rectangle> tabButtons;

    /**
     * Constructeur de l'interface graphique
     * @param gc
     * @param g
     */
    public InterfaceGraphique(GameContainer gc,Graphics g){
        this.gc = gc;
        this.g = g;
        gameWidth = gc.getWidth();
        gameHeight  = gc.getHeight();
        tabButtons = new ArrayList<Rectangle>();
    }
```

Nous avons créé une classe **InterfaceGraphique** dont le constructeur attend en paramètre deux objets de la librairies Slick2D :

1. **GameContainer** : comme son nom l'indique il s'agit de la fenêtre du jeu, contenant toutes les informations nécessaires comme la largeur et la longueur de la fenêtre.
2. **Graphics** : est un objet qui contient des méthodes pour afficher ou dessiner des images, ainsi que des labels à des coordonnées données.

Puis nous avons implémenter des méthodes dans cette classe qui permettent d'afficher les informations d'un pion allié ou ennemi sur lequel on a cliqué par exemple.

```
/**
 * Permet d'afficher les infos d'un pion
 * @param pionSelected
 * @throws SlickException
 */
public void displayInfosPion(Pion pionSelected) throws SlickException{

    boolean isEnnemi = false;

    // Container des infos
    createImage((float) (isEnnemi ? (gameWidth*0.65) : (gameWidth*0.05)),(float) (gameHeight*0.80),(float) (gameW
    // Nom du pion
    createLabel((float) (isEnnemi ? (gameWidth*0.65) : (gameWidth*0.05)),(float) (gameHeight*0.80), 0, -20, pionS
    // Avatar du pion
    createImage((float) (isEnnemi ? (gameWidth*0.65) : (gameWidth*0.05)),(float) (gameHeight*0.80),(float) (gameW

    // Infos de la vie du pion
    createStats((float) (isEnnemi ? (gameWidth*0.65) : (gameWidth*0.05)),(float) (gameHeight*0.80),(float) (gameW
    // Infos de l'attaque du pion
    createStats((float) (isEnnemi ? (gameWidth*0.65) : (gameWidth*0.05)),(float) (gameHeight*0.80),(float) (gameW

    // Infos de la défense du pion
    createStats((float) (isEnnemi ? (gameWidth*0.65) : (gameWidth*0.05)),(float) (gameHeight*0.80),(float) (gameW
    // Infos du déplacement du pion
    createStats((float) (isEnnemi ? (gameWidth*0.65) : (gameWidth*0.05)),(float) (gameHeight*0.80),(float) (gameW
}
```

Selon le camp du pion (allié ou ennemi) les informations vont être placé à des coordonnées différentes pour plus de visibilité.

On peut voir ici les différentes caractéristiques d'un pion ainsi qu'une petite icône le représentant.

Cette classe sert aussi à afficher les points d'actions restants d'un joueur mais aussi qui est le joueur qui doit jouer.

3.3 Contrôle du personnage

Cette partie sert à expliquer comment sont contrôlés les pions. Le principe est simple, pour déplacer un pion, il suffit de le sélectionner en cliquant dessus, puis de cliquer sur la case de destination dans la limite des points de mouvement d'un pion.

Une fonction de contrôle a été créé dans le but de s'assurer qu'un pion puisse se déplacer sur la case cliquée, c'est la fonction **setPositionOfPion** de la classe **GestionnaireCarte** :

```
public boolean setPositionOfPion(int number,int x,int y){
    //Get the player which has to be moved.
    Pion p = personnages.get(number);
    //Get the pos around the player
    boolean bBottomRight = p.getPosX()==x && p.getPosY() ==y+1;
    boolean bUpperRight = p.getPosX()==x+1 && p.getPosY() ==y;
    boolean bBottomLeft = p.getPosX()==x-1 && p.getPosY() ==y;
    boolean bUpperLeft = p.getPosX()==x && p.getPosY() ==y-1;
    //Verify that the pos targeted is next to the player
    if( !(bBottomRight || bUpperRight || bBottomLeft || bUpperLeft)){
        System.out.println("This tile is too far, you can't go there.");
        return false;
    }
    //Check if the tile isn't an obstacle.
    if( map.isTileAnObstacle(x, y) ){
        System.out.println("This is an obstacle, you can't go there.");
        return false ;
    }

    for(Pion allPions : personnages){
        if(allPions.getPosX() == x && allPions.getPosY()==y){
            System.out.println("This is an entity here, you can't go there.");
            return false;
        }
    }

    p.setPosX(x);
    p.setPosY(y);
    return true;
}
```

Cette fonction prend en paramètre **l'id** d'un pion dans **l'ArrayList** des pions du plateau de jeu, ainsi que les **coordonnées** de la case de destination.

Si la case est trop loin du pion, que celle-ci est un obstacle ou qu'un autre pion s'y trouve déjà, on informe le joueur qu'il est impossible de déplacer le pion à cet endroit.

Si ce n'est pas le cas, le pion est déplacé à cet endroit.

Un autre élément important de contrôle du personnage est la phase d'attaque. Elle est gérée par la fonction **applySpell** qui prend en paramètre le **Sort jeté** et le **Pion attaquant**.

```
public void applySpell(Sort s,Pion caster){
    this.defense = this.defense - (s.getArmorDamages()*caster.getAttaque());
    if(this.defense<s.getHpDamages()*caster.getAttaque()){
        this.pdv = this.pdv + this.defense - (s.getHpDamages()*caster.getAttaque());
    }
}
```

Cette fonction est appelée lorsque le **joueur attaquant** clique sur l'icône en **forme d'épée** sur les **caractéristiques d'un pion**, et qu'il sélectionne le **pion ennemi** à attaquer.

3.4 Ajout d'une carte

Nous allons maintenant parler de l'implémentation de la carte en jeu. Tout d'abord il faut savoir que la carte se comporte comme un **tableau à 2 dimensions** contenant des objets **Tile** (qui hérite de la classe **Polygon** de la librairie **Slick2D**).

Cette carte est issue de l'objet **Map** crée pour l'occasion contenant donc le tableau à 2 dimensions de **Tile**, ainsi que les dimensions des cases et de la carte.

```
public class Map {
    private int renderX=400,renderY=100;
    //Drawable map
    private TiledMap map;
    //Interactive map
    private Tile[][] interactiveMap;
    //Map variables
    private int mapHeight, mapWidth, tileHeight, tileWidth,objectLayout,groundLayout;
    //Personnages présents sur la carte
```

La carte est ensuite initialisée au lancement du jeu grâce à la fonction **initializeInteractiveMap** qui remplit le tableau à 2 dimensions de **Tile** positionnés les uns après les autres grâce aux dimensions et au coordonnées données.

```
//Create the map
private void initializeInteractiveMap(){
    mapHeight = map.getHeight();
    mapWidth = map.getWidth();
    tileHeight = map.getTileHeight();
    tileWidth = map.getTileWidth();
    groundLayout= map.getLayerIndex("ground");
    objectLayout = map.getLayerIndex("obstacles");

    float[] points;
    interactiveMap = new Tile[tileHeight][tileWidth];
    for(int j=0;j<mapHeight;j++){
        for(int i=0;i<mapWidth;i++){
                                //premier X                         ,premier Y
            points= new float[]{ (renderX +(i+1)*tileWidth/2) -j*(tileWidth/2),renderY+
            interactiveMap[j][i] = new Tile(new Polygon(points),j,i);
        }
    }
}
```

Une case sur la carte est un objet instancié de la classe **Tile** qui contient les éléments suivants :

1. Un **polygon** (**cliquable** et pouvant contenir du **vide**, un **pion** ou un **obstacle**)
2. Coordonnée X
3. Coordonnée Y

```
public class Tile {
    private Polygon tile;
    private int coordX,coordY;

    public Tile(Polygon par_tile,int par_coodX,int par_coordY){
        tile=par_tile;
        coordX=par_coodX;
        coordY=par_coordY;
    }
```

4. Conclusion

Pour conclure, voici un listing des différents éléments qui nous ont posé problème durant la phase de conception et de développement du projet.

1. **Gestion du clic sur les cases difficiles**

   Nous avons rencontré des difficultés pour détecter si le clic de la souris est bien compris dans un **Tile**. Nous avons dû créer une fonction de collision qui va vérifier si la souris au moment du clic se trouvait bien dans le **Polygon** du **Tile** grâce aux dimensions et aux coordonnées de ces-derniers.

2. **Un sol type de sort disponible**

   Pour l'instant, par soucis de temps, nous n'avons développé qu'un seul sort pour tous les pions. Normalement un **sort** possède plusieurs caractéristiques comme un nom, une portée, le montant des dégâts et la pénétration d'armure. Ainsi, nous pouvons en créer plusieurs sortes pour diversifier les attaques et rendre les pions uniques comme prévu à la base.

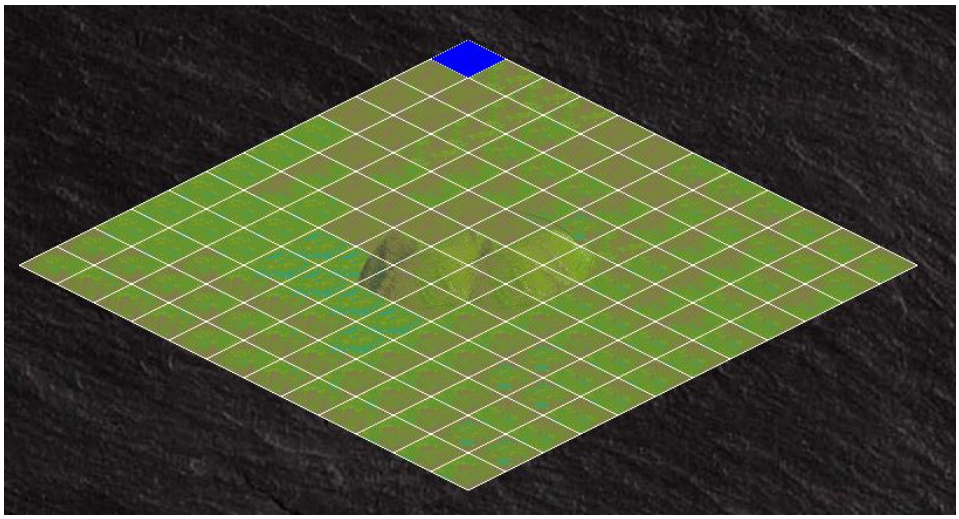
3. **Pas de présence de menu principal**

   Contrairement à ce qui est modélisé dans le diagramme de cas d'utilisation, nous n'avons pas eu le temps également de créer un menu répondant aux besoins.

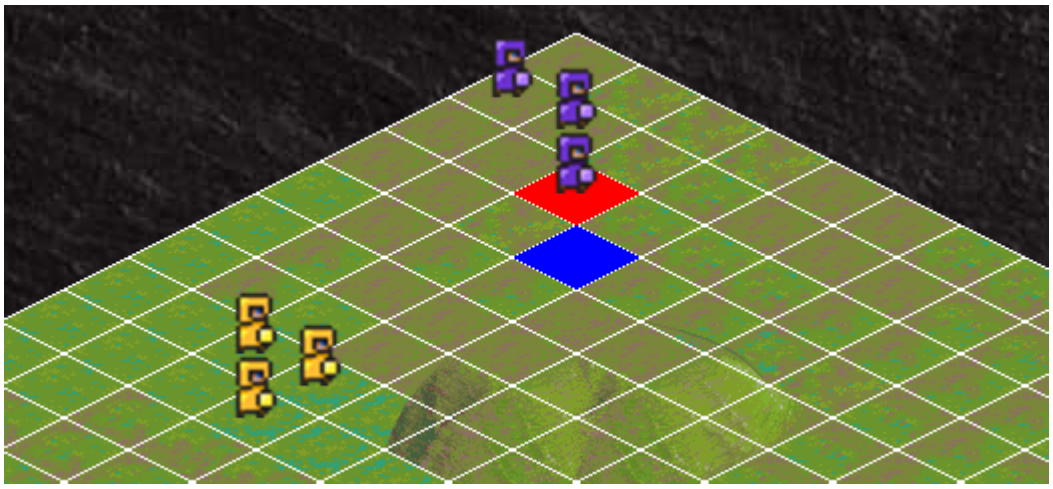
Nous avons pris plaisir à développer ce projet avec les connaissances qu'on a assimilé durant la licence. Malgré les difficultés rencontrées, nous avons su gérer notre temps et développer les bases fonctionnelles de notre projet conformément au cahier des charges et aux diagrammes.

5. Captures d'écran

1. **Carte vierge du jeu**



2. **Sélection d'un pion**



3. **Message d'erreur à cause d'un pion qui ne peut pas se déplacer sur une case trop éloignée**

4. **Affichage des informations d'un pion**