

ASD2

Tower Defense

Mise en œuvre d'un graphe interactif

Création d'une variante du célèbre jeu « Tower Defense » dans le cadre d'un cours d'algorithmique.



Sommaire

1	Analyse préliminaire	4
1.1	Introduction	4
1.2	Organisation.....	4
1.3	Objectifs	5
1.4	Planification initiale	6
2	Analyse.....	7
2.1	Concepts algorithmiques	7
2.1.1	Outils algorithmiques	7
2.1.2	Complexité	9
2.2	Etude concurrentielle et rétrospective.....	10
2.2.1	Bref historique.....	10
	Détails des plus célèbres jeux « Tower Defense »	11
2.2.2	ESSOR actuel sur l'Internet et avenir	13
2.3	Etude de faisabilité	14
2.3.1	Risques techniques.....	14
2.3.2	Risques concernant le planning & ressources humaines.....	15
2.3.3	Risques concernant le budget.....	15
3	Conception.....	16
3.1	Dossier de conception	16
3.1.1	Systèmes d'exploitation	16
3.1.2	Outils logiciels.....	16
3.1.3	Librairies externes	16
3.1.4	Architecture de l'application.....	16
3.1.5	Schémas UML.....	17
3.1.6	Interface graphique.....	20
3.1.7	Architecture de l'application.....	22
3.1.8	Gestion de la concurrence	25
4	Réalisation	26
4.1	Dossier de réalisation.....	26
4.1.1	Résultat de l'interface graphique.....	26
4.1.2	Résultat de l'implémentation du maillage	28
4.2	Description des tests effectués.....	32
4.2.1	Génération du maillage (graphe associé) avec <i>JGraphT</i>	32
4.2.2	Recherche du chemin le plus court.....	35
4.2.3	Enregistrement des scores et leur tri.....	37
4.2.4	Test sur un autre environnement	40
4.2.5	Test global de l'application	40
4.3	Erreurs restantes.....	42
5	Conclusions.....	43
5.1	Objectifs atteints / non-atteints	43
5.2	Points positifs / négatifs	44
5.2.1	Points positifs	44

5.2.2	Points négatifs.....	44
5.3	Difficultés particulières	44
5.4	Avenir du projet	44
6	Annexes	46
6.1	Sources – Bibliographie.....	46
6.2	Journal de bord de chaque participant.....	46
6.3	Manuel d'Utilisation.....	46
6.3.1	Lancer le jeu	46
6.3.2	Choisir un terrain.....	47
6.3.3	Jouer	47
6.3.4	Raccourcis clavier	47
6.3.5	Couper le son.....	47
6.3.6	Fin du jeu	47
6.3.7	Consulter les scores.....	47
6.3.8	Quitter le jeu	48
6.4	Archives du projet.....	48
6.4.1	CD	48
6.4.2	Internet.....	48

1 Analyse préliminaire

1.1 Introduction

Ce projet de fin de semestre consiste à créer une application ludique mettant en œuvre des algorithmes et structures de données étudiées en cours. Nous avons choisi pour cela de créer une variante du célèbre jeu « Tower Defense », dans lequel des personnages se déplacent d'un point A à un point B selon un chemin optimal. En effet, on aura besoin pour cela d'une structure de graphe ainsi que les algorithmes associés, ce qui colle parfaitement avec la contrainte de départ car nous les avons étudiés en cours.

Le but est également ici de créer une application « didacticiel » (il s'agit d'un logiciel interactif destiné à l'apprentissage de savoirs) qui sera présentée dans les futurs cours de cette unité d'enseignement. Ce projet permettra en effet de montrer une application réelle de l'utilisation d'algorithmes associés à des graphes (en particulier celui de recherche du chemin le plus court entre deux nœuds).

Nous commencerons par effectuer une analyse du projet, notamment en ce qui concerne l'organisation, les objectifs ainsi que la planification initiale. Puis nous détaillerons les étapes de conception et de réalisation. Enfin, avant de conclure, nous présenterons les fonctionnalités de l'application finale sous forme de mode d'emploi et nous présenterons des captures d'écran.

1.2 Organisation

Les membres participant à ce projet sont :

Etudiant 1 : Aurélien Da campo, *aurelien.dacampo@heig-vd.ch*

Etudiant 2 : Pierre-Dominique Putallaz, *pierre-dominique.putallaz@heig-vd.ch*

Etudiant 3 (responsable de projet) : Lazhar Farjallah, *lazhar.farjallah@heig-vd.ch*

Nous prévoyons de nous répartir les tâches de manière suivante entre chaque entité du groupe :

1. Responsable de projet

- a. Rédactions, administrations
- b. Suivi des rendus (*deadlines*)
- c. Surveillance et coordination
- d. Développement

2. Etudiant 1

- a. Création de l'interface graphique
- b. Gestion de l'affichage
- c. Rendu graphique
- d. Interaction avec l'utilisateur

3. Etudiant 2

- a. Algorithmique
- b. Implémentation des algorithmes de graphe
- c. Fournir les briques logicielles pour permettre la construction de la partie fonctionnelle de l'application

1.3 Objectifs

Les objectifs de ce projet sont les suivants :

- Illustrer le concept de graphe de manière ludique et interactive.
- Acquérir de l'expérience dans la planification et l'accomplissement d'un projet conséquent.
- Utiliser et découvrir des librairies existantes implémentant le concept de graphe
- Apprendre à mettre en œuvre une interface graphique en Java.
- Séparer le travail en plusieurs niveaux d'abstraction pour faciliter l'élaboration et l'évolutivité de ce projet.
- Comprendre la nécessité d'utiliser des algorithmes complexes dans les applications informatiques.
- Mettre en œuvre un algorithme de recherche de chemin le plus court (ACPC).
- Respecter le design pattern MVC (*Model – View – Controller*) qui structure un programme en trois couches principales.

1.4 Planification initiale

Le projet se déroulera du **18 novembre 2009 au 15 janvier 2010**, ce qui représente un total de 25 périodes en classe (par personne). Nous prévoyons également de passer un total d'environ au moins 25 périodes par personne en dehors des heures encadrées. **Au total, c'est environ 150 périodes de travail** que nous allons planifier comme suit :

Tâches	nov.09			déc.09				janv.10		
	18	25	2	9	16	23	30	6	13	15
Analyse										
Conception										
Réalisation										
Tests										
Documentation										
Planification										

Remarque : Vue l'ampleur du projet et le temps disponible pour celui-ci, nous avons décidé de ne réaliser que la planification initiale pour le projet et de ne pas détailler plus profondément les durées des sous-tâches. Les sous-tâches en question correspondent aux chapitres de ce dossier que vous pourrez découvrir dans les pages suivantes. Bien évidemment chaque chapitre ne nécessite pas la même implication dans tous les cas, c'est pourquoi nous avons décidé qu'il était du devoir des membres de gérer leur emploi du temps afin de bien répartir leur temps de travail sur toutes les parties qui leurs sont demandés.

2 Analyse

2.1 Concepts algorithmiques

Le principal concept algorithmique utilisé dans ce projet est celui de graphes. En effet, ce concept, bien connu dans le domaine algorithmique, constitue un point clé dans la réalisation de notre cahier des charges. Il intervient principalement dans le problème de la recherche d'un chemin (le plus court) entre deux points donnés A et B.

Si on regarde à un niveau plus abstrait, le but premier de notre application est de lancer des objets (créatures) à partir d'un point A pour qu'ils se dirigent vers un point B selon un chemin optimal, calculé à la volée en fonction d'un graphe qui ne cesse de se modifier (dynamique).

Nous avons également besoin d'un algorithme de tri afin de trier les meilleures scores du jeu obtenus et de les afficher dans l'ordre.

Cette simple vision est tout à fait suffisante pour comprendre les outils algorithmiques dont nous devons s'armer. Nous allons maintenant développer plus en détails quels sont les structures et algorithmes de graphes dont nous avons besoin.

2.1.1 Outils algorithmiques

Partant du constat du paragraphe 2.1, on comprend assez facilement que nous aurons besoin des structures suivantes :

- un graphe pondéré non orienté
- des nœuds composant le graphe
- des arcs pour relier les différents nœuds du graphe
- un algorithme de recherche d'un chemin optimal entre deux nœuds (connu sous l'appellation « algorithme de Dijkstra ») appliqué à un graphe pondéré non orienté
- un algorithme de tri afin de trier les meilleures scores des joueurs

2.1.1.1 Graphe pondéré non orienté

Nous avons décidé d'utiliser un tel graphe car il correspond parfaitement à ce dont nous avons besoin. En effet, chaque arc possède un poids (pondéré), car une créature doit savoir à partir de n'importe quel nœud du graphe quel est la longueur du chemin jusqu'au prochain nœud. De plus, nous avons choisi d'utiliser un graphe non orienté, c'est-à-dire que les arcs n'ont pas de sens (si un nœud A est voisin d'un nœud B alors B est aussi un voisin de A). Les nœuds sont donc traversables en double sens, ce qui se prête bien à nos besoins car il se peut très bien qu'une créature doive tout d'un coup reculer et revenir sur ces pas.

2.1.1.2 Nœuds composants le graphe

Les nœuds de notre graphe contiennent des informations telles que la position x et y (en fonction du repère cartésien du jeu) afin de connaître le parcours à suivre en coordonnées cartésiennes, mais aussi un « drapeau » (*flag* en anglais) permettant de savoir si un tel nœud est actif ou non.

2.1.1.3 Arcs pour relier les nœuds du graphe

Les arcs reliant les nœuds du graphe sont pondérés, c'est-à-dire qu'ils possèdent une valeur associée (un nombre) strictement positif déterminant la longueur (tout simplement) de l'arc en question. Cette longueur est calculée selon la position cartésienne des extrémités de l'arc (autrement dit à partir des 2 nœuds reliés par l'arc) grâce au théorème de Pythagore. Ces arcs sont également non orientés, c'est-à-dire qu'ils ne possèdent pas de sens (chaque arc peut être parcouru dans n'importe quel sens).

2.1.1.4 Algorithme de dijkstra (sur un graphe pondéré non orienté)

Voici, en pseudo-code, l'algorithme de recherche de tous les chemins les plus courts à partir d'un sommet initial vers tout autre sommet du graphe (un sommet est un synonyme de nœud) que nous utilisons :

```
(partie générale)
marquer tous les sommets comme non visités
visiter le sommet initial

(partie spécifique à un sommet : visiter le sommet)
si le sommet est non visité alors
    déposer le sommet avec priorité nulle dans la queue de
    priorité
    tant que la queue de priorité n'est pas vide boucler
        prélever le sommet de tête (il devient le sommet courant)
        marquer le sommet courant comme visité
        traiter le sommet courant
        pour chaque sommet voisin du sommet courant boucler
            si le sommet voisin est non visité alors
                calculer la priorité du sommet voisin (c'est-à-dire
                prendre la somme de l'attribut de l'arc, entre le
                sommet courant et lui-même, et de la priorité du
                sommet courant)
                déposer le sommet voisin et sa priorité dans la
                queue de priorité
            fin si
        fin boucler
    fin boucler
fin si
```

Algorithme 2.1 : recherche des chemins les plus courts dans un graphe (Dijkstra).

2.1.1.5 Algorithme de tri pour le tri des meilleurs scores des joueurs

Pour trier les meilleurs scores obtenus par le joueurs, nous utilisons la méthode `sort()` disponible dans l'API Java, applicable sur un objet de type `ArrayList` (une collection). En effet, Java utilise pour ce faire le tri Merge Sort (tri fusion) dont voici le pseudo-code :

```

procedure triFusionI(entier[] tab)
    entier[] tmp <- tableau de taille N;
    entier i <- 1;
    entier debut <- 1;
    entier fin <- debut + i + i - 1;
    tant que (i < N) faire
        debut <- 1;
        tant que (debut + i - 1 < N) faire
            fin <- debut + i + i - 1;
            si (fin > N) alors
                fin <- N;
            fusion(tab, tmp, debut, debut + i - 1, fin);
            debut <- debut + i + i;
        fin tant que
        i <- i + i;
    fin tant que
fin procedure

```

Algorithme 2.2 : tri fusion sur un tableau d'entiers.

2.1.1.6 Algorithme pour la génération de la santé des créatures

2.1.2 Complexité

2.1.2.1 Algorithme de Dijkstra

Si le graphe possède m arcs et n noeuds, en supposant que les comparaisons des poids d'arcs soient à temps constant, et que le tas soit binomial, alors la complexité de l'algorithme est $O((m + n) \cdot \log(n))$.

2.1.2.2 Tri fusion

On voit que la procédure de fusion nécessite un tableau intermédiaire aussi grand que le nombre d'éléments à trier. C'est en effet le principal inconvénient du tri fusion, car sa complexité est dans tous les cas en $O(n \cdot \log(n))$, (prix du tableau annexe aussi grand que le tableau initial).

2.2 Etude concurrentielle et rétrospective

Dans ce chapitre, nous souhaiterions vous présenter divers jeux qui ont été inspirés par le concept innovant de « Tower defense ». Nous commencerons ce chapitre avec une petite présentation des origines de ce genre de jeux et détaillerons les plus célèbres d'entre eux. Pour finir, nous aborderont encore un sujet d'actualité, il s'agit du grand essor que subissent actuellement ce genre de jeux sur l'Internet et de leur avenir sur la toile.

2.2.1 Bref historique

De nos jours, Il existe une multitude de jeux vidéo qui exploitent ce concept qui ne date pas d'hier. En effet, les premiers jeux vidéo de stratégie en temps réel comme « Age of Empire » sorti en 1997 sont réellement des précurseurs en la matière.



C'est en juillet 2002 qu'une boîte de développement nommée « **Blizzard Entertainment** » lance son troisième opus de la série **Warcraft** et officialise sous la forme d'un « mod¹ » la première version officielle du concept des jeux appelés « Tower Defense ».

La première version de ce mod fut un succès et était déjà très aboutit. En effet, elle offrait déjà énormément de possibilités et fonctionnalités qui seront en majorité reprises par les jeux dérivés. Notons qu'il existe fondamentalement deux types de Tower Defense et que ces deux ont été imaginés dans la version de Blizzard. La différence entre ces deux types repose sur le positionnement des tours. On les appelle, avec ou sans labyrinthe (with / without mazing). Voici une brève description de chacun d'eux :

- Dans une version à labyrinthe, les tours ne peuvent être placées que **le long de l'itinéraire des ennemis (donc chemin des créatures statique)**. Le but est alors de trouver le placement optimal et la meilleure combinaison de tours.
- Dans une version sans labyrinthe, le joueur peut placer ses **tours sur l'itinéraire des ennemis qui les contournent (implémentation obligatoire d'un algorithme de recherche de chemin)**. La stratégie est alors de créer des chemins qui forcent les vagues d'ennemis à rester le plus longtemps possible sous le feu des tours.

¹⁾ Un mod est un jeu vidéo créé à partir d'un autre jeu vidéo, ou une modification du jeu original, sous la forme d'une greffe qui se rajoute à l'original, le transformant parfois complètement.

Détails des plus célèbres jeux « Tower Defense »

Warcraft 3 (le mod)

La plus célèbre version du jeu reste sans nul doute la version originale développée par la célèbre boîte de Los Angeles appelée « Blizzard Entertainment ».

Avec son mode multi joueurs permettant de défendre un labyrinthe commun à plus de 8 joueurs, cette version a encore énormément de charme auprès des joueurs sur l'Internet et dans les réseaux locaux de jeux (LAN).



Figure 1 TD de Warcraft III

Points forts :

- + multi joueurs
- + énormément de type de tours et créatures
- + en 3D
- + mode avec et sans labyrinthe
- + outils de création de maps
- + et j'en passe...

Flash Element TD

Flash Element TD est un jeu de type Tower Defense inspiré par le désormais célèbre jeu de stratégie en temps réel Warcraft 3.

Développé et mis à jour actuellement plus ou moins quotidiennement par David Scott, le jeu est très vite devenu un gros succès avec 250 000 visiteurs journaliers, 10 jours à peine après sa sortie (janvier 2007).

Points forts :

- + flash (pas d'installation)
- + Respect du design de Warcraft 3 en 2D
- + On retrouve les éléments originaux de Warcraft 3
- + Ambiance

Flash Element TD



Figure 2 jeu Element TD

Desktop Tower Defense

Une version très récente du concept et une des plus jouée actuellement sur l'Internet.

Ce qui rend cette version différente des autres jeux de type « Tower Defense » est que cette dernière se déroule sur une carte totalement vierge! C'est à vous de créer entièrement votre propre labyrinthe.

Lien : <http://www.handdrawinggames.com/DesktopTD/game.asp>

Points forts :

- + flash (pas d'installation)
- + mode sans labyrinthe
- + très complet



Figure 3 Jeu Desktop Tower Defense

Onslaught 2

Onslaught 2 est un autre classique des jeux de Tower Defense déniché sur l'Internet. Ici c'est une version avec un labyrinthe.

Lien : <http://onslaught.playr.co.uk/>

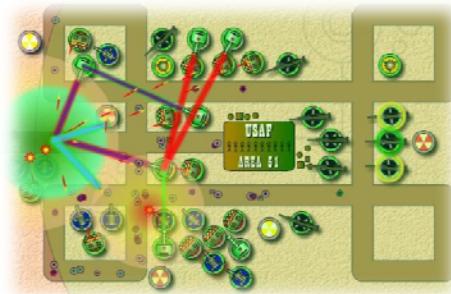


Figure 4 Jeu Onslaught 2

Points forts :

- + flash (pas d'installation)
- + grand choix de tour
- + 4 niveaux de difficulté
- + possibilité de créer ses propres cartes

Vector Tower Defense

Un tower defense avec un design futuriste.

Lien : <http://www.candystand.com/play/vector-td>

Points forts :

- + flash (pas d'installation)
- + Design et Maps originales
- + Ambiance générale



Figure 5 Jeu Vector Tower Defense

Epic Tower Defense

Voici un premier essai avec Unity tournant autour de la licence Warcraft : Epic Tower Defense. Ce n'est pas encore tout à fait au point mais ça risque de donner de très belles choses dans le futur.

Lien :

<http://www.shockwave.com/gamelanding/epictowerdefense.jsp>



Figure 6 Jeu Epic Tower Defense

Points forts :

- + **Unity !**
- + en 3D
- + licence Warcraft 3
- + Projet très ambitieux

2.2.2 Essor actuel sur l'Internet et avenir

Les « **Tower Defense** » rencontrent actuellement un grand succès sur l'Internet, notamment grâce à la technologie Flash qui est utilisée dans la majorité des cas.

Mais ce n'est qu'une première étape car avec l'arrivée des environnements riche, de la 3D dans les navigateurs, nous risquons de voir bientôt débarquer de bien belles choses. Notamment en provenance d'un environnement du type Unity (<http://unity3d.com/unity/>).

2.3 Etude de faisabilité

Ce chapitre permet de prendre le temps d'analyser et cerner les points délicats qui pourraient rendre le projet impossible. En parallèle de ce dernier il est possible que les membres du projet commencent déjà à mettre en place quelques implémentations afin de répondre concrètement à cette étude. Pour chacun des sujets abordés nous nous posons une question et tentons d'y répondre le mieux possible pour évaluer les risques qui pourraient survenir. Nous décomposons cette étude en trois-sous chapitres présentés ci-dessous.

2.3.1 Risques techniques

Nous décrivons ici les risques techniques liés principalement au développement de l'application.

- **Graphe & algorithmes**

Question : Arriverons-nous à implémenter les algorithmes nécessaires au bon fonctionnement du jeu ?

Réponse : Nous avons eu un cours dédié à l'apprentissage des graphes et aux algorithmes qui nous permettent de comprendre leur implémentation.

Question : Devrons-nous créer notre propre TDA ou trouverons-nous une librairie ou un TDA déjà réalisée et fonctionnelle afin de nous éviter de devoir tout refaire ?

Réponse : Nous avons trouvé une librairie gratuite et libre qui nous offre exactement ce que nous voulions. Celle-ci s'appelle jGraphT (<http://jgrapht.sourceforge.net/>)

- **Interface Java**

Question : Arriverons-nous à réaliser l'interface graphique avec Java ?

Réponse : Les membres du projet ont déjà de bonnes connaissances en interface graphique avec Java et se sentent capable de réaliser cela. De plus nous suivons en parallèle un cours de Java et nous aborderons les notions d'interface graphique.

- **Architecture de logiciel**

Question : Serons-nous capable de faire cohabiter tous les éléments du jeu dans un ensemble d'une façon logique ?

Réponse : Plusieurs membres du projet ont déjà réalisé des jeux vidéo durant leur formation et savent plus ou moins comment structurer une telle architecture. De plus avec les notions actuelles de « programmation orienté objet », nous sommes capable de concevoir cela sans trop de problème.

Question : Arriverons nous à réalisé l' « inverse engineering » basé sur les versions du jeu existantes sur l'Internet ?

Réponse : L'étude concurrentielle et nos expériences nous ont permis de bien comprendre les éléments clefs du concept. Après quelques schémas d'essais nous avons réussi à cerner les objets et leur interaction entre eux.

2.3.2 Risques concernant le planning & ressources humaines

Nous abordons ici les risques liés au travail des membres et de leur communication.

- **Délais**

Question : Arriverons-nous à fournir un logiciel fonctionnel dans les temps ?

Réponse : Le temps pour le projet est court. Ce pourquoi nous allons dans un premier temps nous concentrer sur l'essentiel et éviter les choses superflues comme la gestion des scores, les musiques et choses similaires. De plus, comme le sujet est très intéressant nous allons prendre beaucoup de notre temps libre pour réaliser ce travail. S'il nous reste du temps, nous n'éditerons pas à implémenter ces points annexes

- **Répartition des tâches et travail d'équipe**

Question : Partagerons-nous suffisamment bien les tâches de façon équitable entre les membres du projet ?

Réponse : Nous avons décomposé le projet en plusieurs couches bien distinctes (chef de projet, gestion de maillage et des algorithmes, interface graphique, conception du jeu) et avons distribué chacune des aux membres. Nous avons également prévue des tâches optionnelles (musique, design, gestion des scores, etc.). Si un membre termine complètement sa partie il pourra alors se concentrer sur les parties annexes.

- **« Versionnage » & mise en commun**

Question : Comment allons-nous gérer les versions du jeu et la mise en commun des parties ?

Réponse : plusieurs membres du projet connaissent très bien des applications de gestion du « versionnage » et gestion de projet de développement. Les membres qui ne sont pas encore familiarisés avec ces notions apprendront à les utiliser. Ces genres de logiciels sont extrêmement puissants et permettent un gain de temps énorme.

2.3.3 Risques concernant le budget.

Aucun budget n'est nécessaire à la réalisation de ce projet.

3 Conception

3.1 Dossier de conception

3.1.1 Systèmes d'exploitation

Le système d'exploitation sur lequel nous travaillons est Windows (XP et Seven). Cependant, grâce au choix qui a été fait d'utiliser un encodage de type UTF-8 ainsi que celui du langage portable Java, nous pouvons sans soucis travailler sur un environnement Linux ou Mac par exemple.

3.1.2 Outils logiciels

Nous développons avec l'IDE Eclipse (version 3.4 et supérieure) intégrant tous les outils nécessaires au développement d'applications Java. Ce logiciel est très largement répandu dans le monde des développeurs et est très utilisé. Il possède de nombreuses fonctions spécialement conçues pour augmenter la productivité des développeurs et leur simplifier la vie (comme le *refactoring* par exemple). De plus, cette plateforme nous permet d'ajouter toute une série de plugins qui peuvent ajouter des fonctionnalités, telles que SVN (logiciel de gestion des versions du code). En ce qui concerne la génération des diagrammes de classe UML, nous utilisons le plugin **eUML 2.0** de chez **Soyatec**. Finalement, pour la gestion des versions en fonction de l'avancement du projet, nous utilisons le fameux logiciel **svn** ainsi que le service **GoogleCode** afin de garder une trace de notre code à chaque nouvelle étape du projet, le tout étant sauvegardé sur un serveur.

3.1.3 Librairies externes

3.1.3.1 JGraphT

Nous utilisons la librairie (libre) externe **JGraphT** codée en Java. En effet, cette librairie possède toutes les briques logicielles nécessaires à la création de graphes (de tout type). Nous l'utilisons comme une boîte noire sans se soucier de son implémentation. Le choix de cette librairie n'a pas été sans réflexion. En effet, après quelques recherches effectuées sur Internet ainsi qu'une discussion avec le professeur, cette librairie nous paraît correcte.

3.1.3.2 JLayer

Nous utilisons la librairie (libre) externe **JLayer** codée en Java. Cette librairie nous permet de jouer des musiques codées en divers formats tel que le mp3 par exemple (ou le wav). Elle sert uniquement à jouer de la musique, ce qui apporte un petit plus au projet.

3.1.4 Architecture de l'application

Nous avons choisi comme architecture du programme le fameux design pattern MVC (Modèle – Vue – Contrôleur).

3.1.5 Schémas UML

3.1.5.1 Maillage

Voici un petit schéma UML du maillage, avec les classes associées.

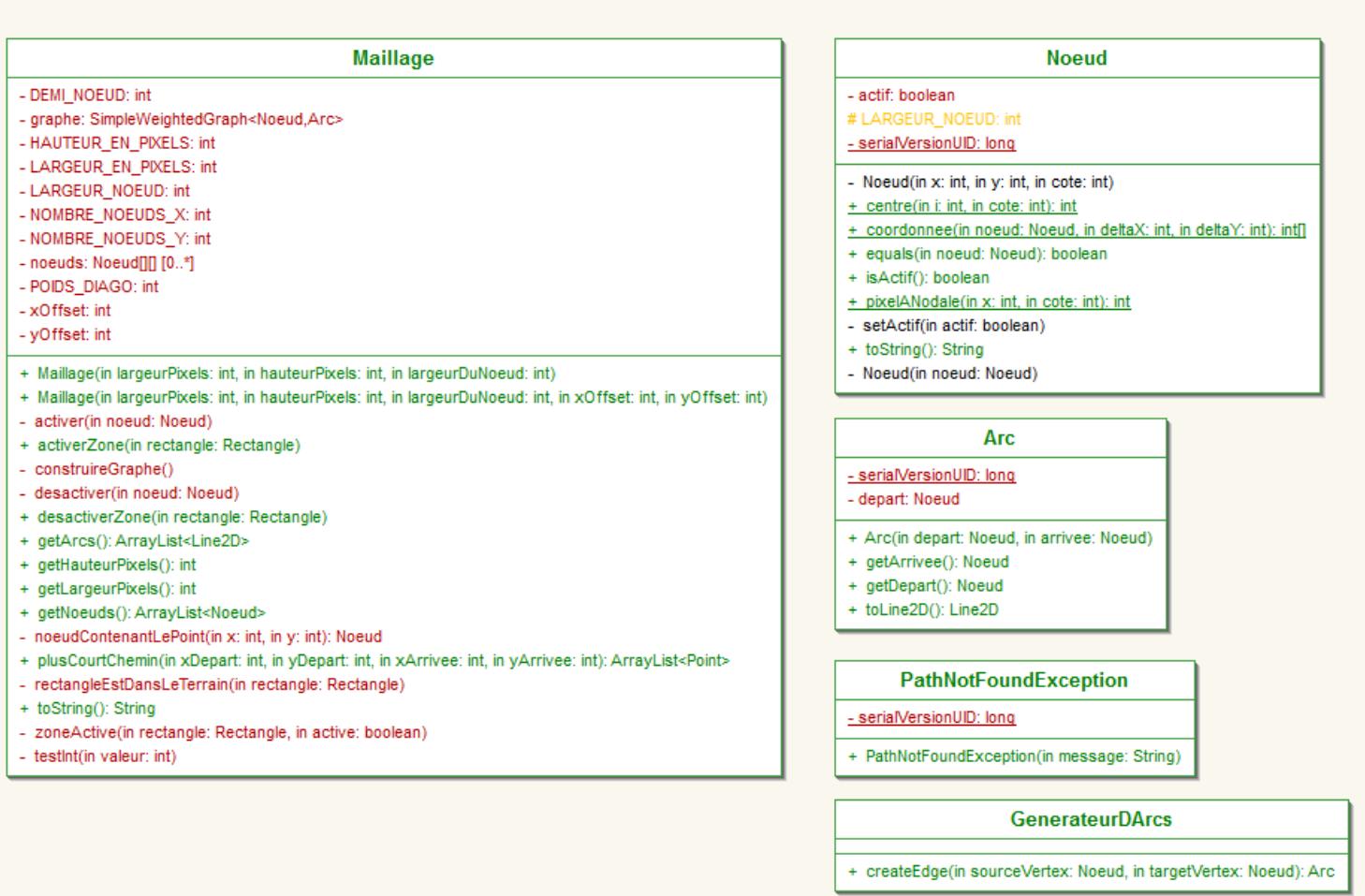


Figure 7 : Schéma UML du TDA maillage

Une description exacte des méthodes les plus importantes se trouve dans la section 4.1.2, notamment concernant les méthodes publiques utilisées hors du package spécifique.

3.1.5.2 Gestionnaire du jeu (model)

Nous avons décidé de fournir un schéma UML simplifié (sans getter/setter et attributs de gestion spécifique) afin de rendre la compréhension plus aisée. Voici comment nous pensons organiser nos classes :

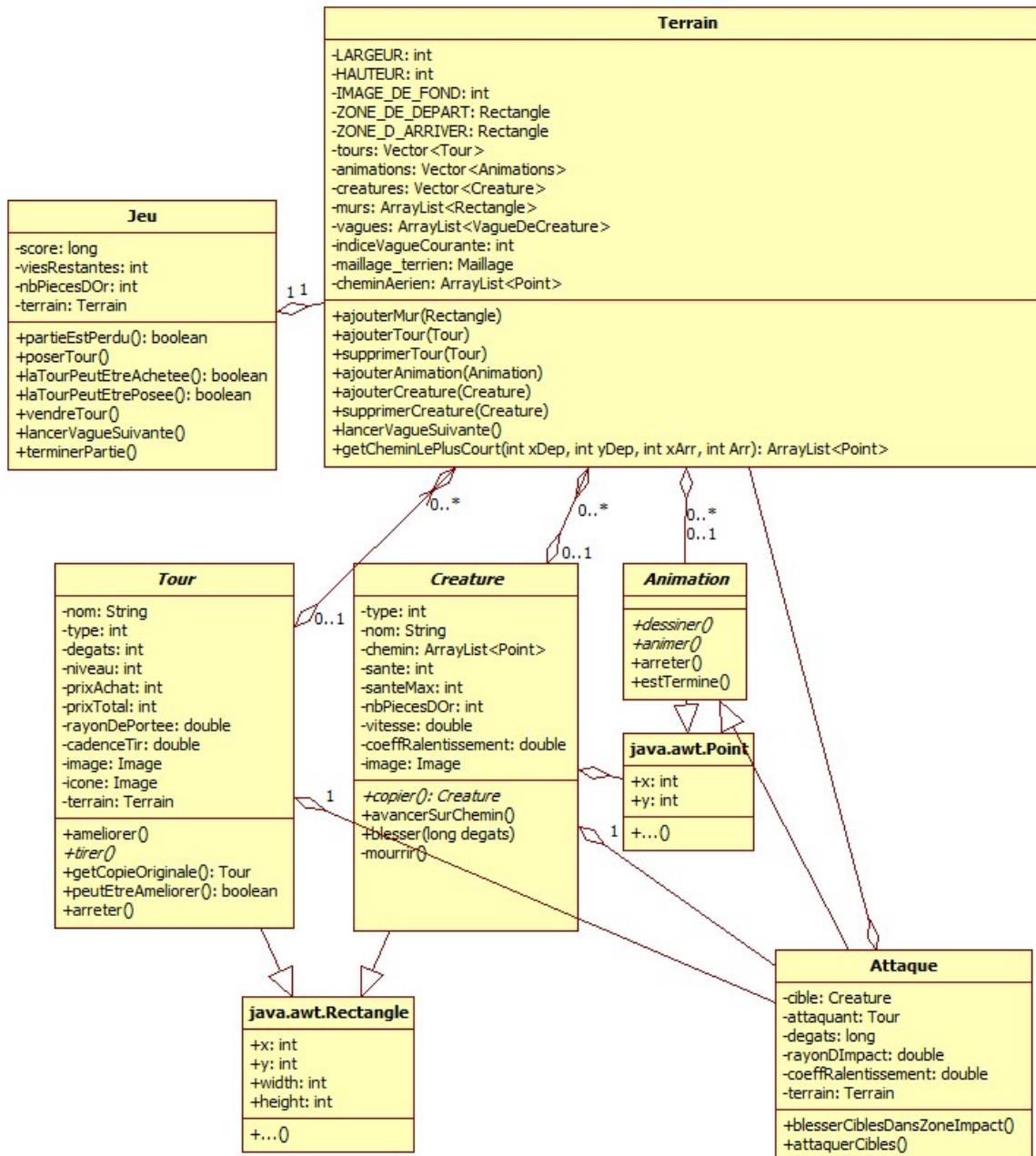


Figure 8 Schéma de classe du jeu

Commençons notre explication par la classe principale que nous nommons **Jeu**. Celle-ci contiendra les données du joueur et de la partie en cours. Elle permettra également d'encapsuler le terrain. La classe **Terrain**, elle, contiendra toutes les collections d'éléments comme les tours, les créatures et les animations. C'est également elle qui encapsulera le maillage et définira les zones de départ et d'arrivée des créatures. Les murs (simple rectangle invisible) du terrain permettront de rendre des zones non accessibles pour les

créatures et les tours (pose de tours impossible). Passons maintenant à la classe **Tour**, cette entité est caractérisée par un prix d'achat, des dégâts, une cadence de tir, un rayon de portée, etc. Celle-ci peut tirer sur des créatures et a besoins du terrain pour savoir où sont les créatures. Dans ce jeu, une **Créature** est caractérisée principalement par sa santé, sa vitesse de déplacement et son nombre de pièces d'or gagné par le joueur lors de son décès. Les tours et les créatures héritent de la classe Rectangle fourni par Java qui nous permet de gérer facilement les collisions. Ensuite, le dernier groupe éléments principales est les **animations**. Une animation sera surtout utilisée d'un point de vue esthétique mais les attaques (qui sont des animations) joueront un rôle important sur les modèles.

Faute de temps nous n'allons pas détailler d'avantage cette partie. Nous répondrons volontiers à vos questions après le projet et nous vous invitons aussi à consulter la « java doc » fournie en annexe qui présente et détail toutes les classes et les membres qui les composent.

3.1.6 Interface graphique

Ce chapitre nous permettra d'imaginer et de penser le résultat final de l'interface graphique. Grace à cette étude, nous pourrons plus facilement élaborer et trouver les classes qui en découlent.

Commençons par lister les actions possibles et informations visibles dans la fenêtre de jeu. I s'agit d'un schéma UML « Use Case » sous la forme textuelle :

- Voir les créatures se déplacer sur le terrain et les tours tirer sur les créatures
- Choisir et poser une tour sur le terrain
- Améliorer ou vendre une tour sélectionnée
- Voir les informations d'une tour (ces caractéristiques)
- Voir les informations d'une créature sélectionnée
- Voir les informations du jeu (score, argent, vies restantes)
- Voir les informations de la vague de créatures suivante et courante
- Lancer la vague de créature suivante
- Quittez le jeu
- Ouvrir la fenêtre « A propos »

Ensuite nous élaborons un schéma d'interface graphique permettant d'implémenter toutes les actions et informations citées ci-dessus. En voici le résultat final :



Figure 9 Schéma de la fenêtre principale

La fenêtre est décomposée en 4 parties :

La première partie, celle au nord, contient le menu permettant de quitter l'application (fichier -> quitter) et d'ouvrir la fenêtre « à propos » (aide -> à propos). Le menu Edition permettra d'offrir d'autres fonctionnalités comme l'affichage du maillage (pour debug).

La deuxième concerne l'affichage du terrain et de l'animation du jeu. C'est ici que l'on verra bouger les créatures et tirer les tours. C'est aussi dans cette partie que le joueur posera ses tours et sélectionnera les tours et les créatures.

La troisième zone est la partie de droite qui est composé de 4 boites :

- « *Infos du jeu* » permettant au joueur de voir l'état de la partie caractérisé par le score, l'argent du joueur et le nombre de vies restantes.
- « *Choix des tours* » offrant la possibilité au joueur de sélectionner une tour à acheter (pour acheter la tour sélectionnée, le joueur doit alors cliquer sur une zone du terrain pour choisir où mettre la tour)
- « *Info d'une tour sélectionnée* » permettant de montrer les caractéristiques d'une tour en particulier (prix, dégâts, cadence de tir, bonus, etc.) et c'est ici que le joueur peut améliorer ou vendre une tour.
- « *Info d'une créature sélectionnée* » qui permet de voir les caractéristiques d'une créature (nom, santé, vitesse, etc.)

Le quatrième cadran permet au joueur de voir des informations sur les vagues suivante ainsi que de lancer la vague suivante d'ennemis. Nous pourrons également diffuser des messages d'erreur ou autre dans la zone d'affichage des infos vagues.

Les autres fenêtres (fenêtre à propos, et menu principal d'accueil) de l'application ne nécessitent pas une étude particulière. En effet, elles seront relativement simples et nous ont demandé uniquement un petit croquis sur papier. Elles ne feront donc pas l'objet d'une décortication dans ce chapitre.

3.1.7 Architecture de l'application

Nous avons choisi d'implémenter une architecture très répandu dans la conception de logiciels, il s'agit du célèbre design pattern **MVC** (Modèle – Vue – Contrôleur). Les étudiants responsables de cette partie s'étaient déjà familiarisés avec cette notion durant d'autres projets d'études et nous l'avons également étudié durant le cours de « programmation orientée objet » qui nous suivons en parallèle de ce projet. Malheureusement nous n'allons pas détailler et décrire le fonctionnement du pattern mais présenter comment nous pensons l'implémenter dans ce jeu vidéo.

MVC dans un jeu vidéo

Dans un jeu vidéo, il est difficile de séparer distinctement les vues des modèles car les éléments des gestions sont parfois intrinsèquement liés à leur affichage. En effet, quasiment tous les éléments comme les tours, les créatures, le terrain, la gestion des sons et les animations (attaques ou autre) sont bien des objets faisant partie du modèle mais d'un autre côté, il est difficile de ne pas les intégrer dans les vues car les interactions entre eux peuvent parfois passer par des affichages et traitement particuliers.

Imaginons l'affichage des animations, lors de la construction de l'image représentant les animations sur le terrain. Est-il mieux d'établir un standard d'affichage pour toutes les animations séparant ainsi les vues des animations ou de dire à l'animation dessine toi et lui passer le pointeur vers les outils de dessin? A priori, la première solution est la meilleure mais dans ce cas, cela nous force à avoir un standard de pour dessiner l'objet se qui peut limiter considérablement la diversité et rendre très difficile certains résultats souhaités. Par exemple, certaines animations pourraient être représentées par une image et d'autre peuvent être construites avec des traits ou formes géométriques de base. Pour cette raison nous essayerons de séparer au maximum les modèles des vues dans une certaine mesure de portabilité. Nous devons donc imaginer une architecture nous permettant de migrer sans trop de problème notre application vers un autre système d'interface comme par exemple OpenGL (3D) ou encore Java3D.

Contrôleur principale

Après plusieurs schémas et tests, nous avons décidé de simplifier l'architecture en créant une classe qui s'occupera de gérer toutes les interactions entre les vues et les modèles. La classe qui jouera ce rôle sera la fenêtre principale du jeu. Celle-ci sera donc informée de tous les événements du joueur et des modèles (sous la forme d'écouteurs) et son travail sera d'en informer les autres modèles et toutes les parties de l'interface. Elle sera donc le chef d'orchestre de l'application. Ci-dessous nous vous présentons un schéma simplifié mettant en œuvre les différentes parties citées ci-dessus :

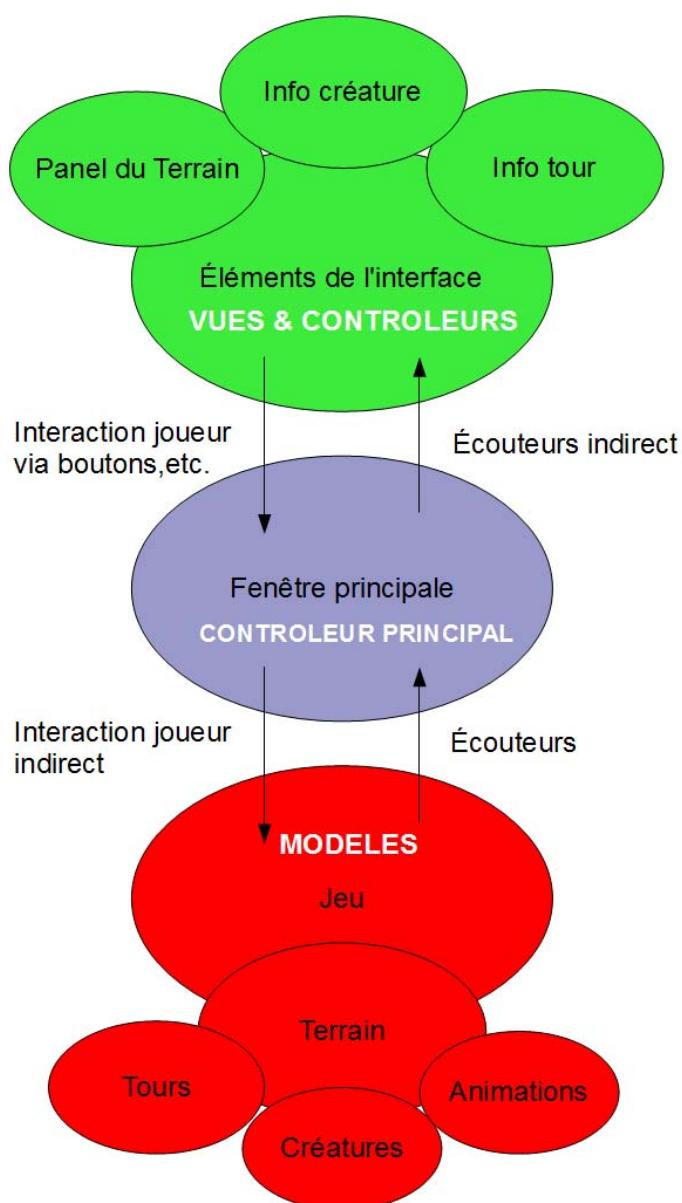


Figure 10 Schéma de l'architecture MVC

Notre organisation

Voici un résumé des différentes classes et leur(s) emplacement(s) dans le pattern :

Vues

- Toutes les fenêtres
- Les panels contenus dans les fenêtres
- Les animations, sons et autres classes modèles (le moins possible)

Modèles

- Les tours
- Les créatures
- Le terrain
- Les données du jeu
- Les animations
- Les sons (lancés par des modèles et pour les vues)

Contrôleurs

- Boutons et objets événementiels contenu dans les fenêtres
- Fenêtre principale de diffusion (explication à la page précédente)

Après une implémentation concrète, nous trouvons cette architecture fonctionnelle mais nous pensons néanmoins qu'elle pourrait encore être améliorée en rendant la fenêtre de gestion des événements (fenêtre principale) indépendante de toute interface. Il s'agirait de créer une classe à part qui jouerait ce rôle de gestionnaire de communications modèles / interfaces.

3.1.8 Gestion de la concurrence

Ici nous allons très brièvement discuter d'un point relatif à la gestion de threads qui nous a posé quelques soucis, la gestion cohérente de la concurrence.

En effet, nous avons eu des soucis d'interactions entre les threads, notamment lors de la lecture et de l'écriture sur le maillage. La cause était que lorsque l'utilisateur dans le thread principal éditait le maillage à l'aide des fonctions prévues à cet effet, il y avait une probabilité élevée qu'un thread gérant une créature veuille accéder également au maillage au même moment, d'où des exceptions liées à la concurrence levée par la machine virtuelle Java.

Nous avons donc résolu le souci par un mécanisme de gestion de la concurrence de base en Java à l'aide du mot clef *synchronized*, prenant également un paramètre relatif à l'objet auquel on veut accéder de manière exclusive. Ce mot clef permet de définir un *mutex* sur une structure ou une méthode qui n'autorise qu'un seul thread à la fois à accéder à la ressource.

Nos soucis étant liés à un accès concurrent sans gestion de priorité et à des soucis de temps réel, nous n'avons pas jugé utile de pousser plus loin la gestion de la concurrence dans nos classes. Nous n'avons pas utilité de concepts de sections critiques par exemple.

Cependant, nos classes respectent les concepts de suretés (rien qui ne devrait arriver ne se passe, plus d'erreurs liées à la concurrence) et surtout de vivacité. La gestion par *mutex* basiques interdit un éventuelle *deadlock* ("interbloquage"), fatal au bon déroulement du programme.

La gestion propre et efficace des problèmes liés à la concurrence faisant l'objet d'un cours à part entière, nous ne nous sommes pas attardés plus qu'absolument nécessaire sur le sujet. Cependant il nous a été important de mentionner dans le rapport les soucis que nous avons eu ainsi que la manière dont nous avons soit résolu soit contourné le problème.

4 Réalisation

4.1 Dossier de réalisation

4.1.1 Résultat de l'interface graphique

Dans ce chapitre, nous vous présentons les résultats de l'interface graphique sous la forme de copies d'écran avec nos commentaires.

Menu principal

Voici la première fenêtre qui apparaît lors du lancement du jeu. Celle-ci vous permet de sélectionner un terrain de jeu pour votre partie. En cliquant sur le terrain, votre partie de jeu débutera. Vous pouvez également accéder aux meilleurs scores des différents terrains depuis le menu (Fichier -> scores). Pour finir vous pouvez aussi ouvrir la fenêtre « à propos » avec le menu sous l'item « Aide ».



Figure 11 Fenêtre du menu principal

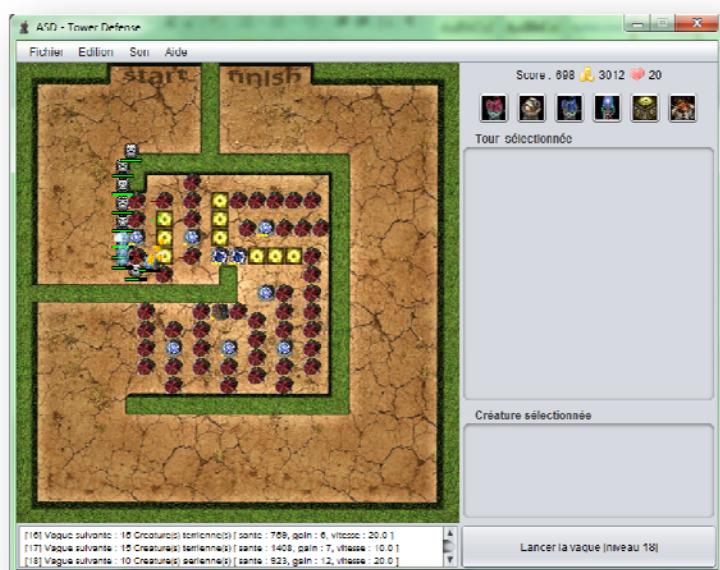


Figure 12 Fenêtre du jeu

Fenêtre du jeu

Voici la fenêtre la plus intéressante du jeu. C'est ici que se déroule le jeu et les interactions du joueur. Toutes les fonctionnalités ont été décrites dans le chapitre dédié à la conception de la fenêtre du jeu. Nous avons tout de même ajouté certaines fonctionnalités annexes disponibles depuis le menu comme la gestion du son, l'affichage du maillage et l'affichage de toutes les portées des tours.

Fenêtre du jeu avec maillage

Vous pouvez découvrir ici, le terrain de jeu avec l'affichage des éléments invisibles (menu Edition). Les éléments invisibles sont : le maillage, les chemins des créatures, les murs et les zones de départ et d'arrivé.

Le trait bleu de cette image montre le chemin qu'effectueront les créatures. Ce chemin utilise les arcs du maillage (en blanc).

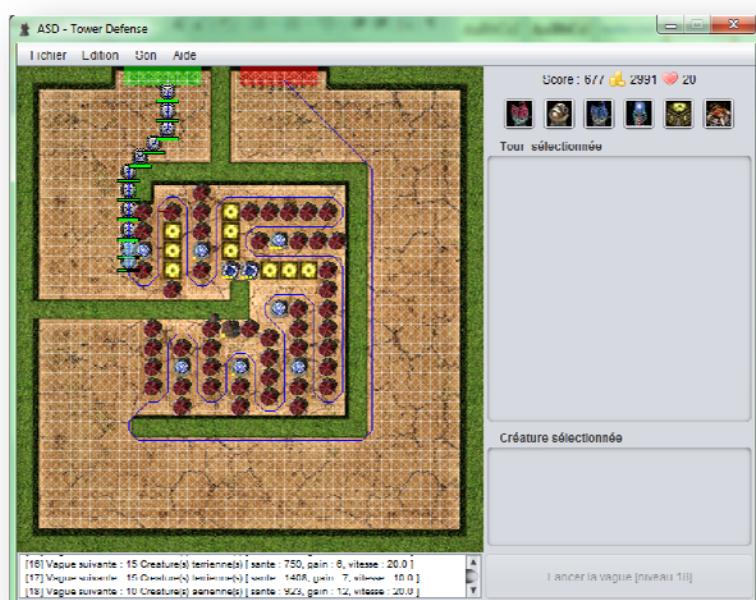


Figure 13 Fenêtre du jeu avec éléments invisibles



Figure 14 Fenêtre d'ajout de score

Fenêtre de sauvegarde du score

Une fois la partie terminée, le jeu vous propose de sauver votre score. Voici la fenêtre où vous pouvez entrer votre nom de joueur.

Fenêtre des meilleurs scores

Une fois votre score sauver vous accéder directement à la fenêtre des 10 meilleurs du terrain sur lequel la partie s'est déroulée.

Vous pouvez également accéder à cette fenêtre depuis le menu principale de l'application (Fichier -> scores)

Les 10 Meilleurs scores [Terrain : WaterWorld]			
DArk	854	12.01.10 00:35	
Jojo le plus fort	645	14.01.10 00:41	

Figure 15 Fenêtre des meilleurs scores

4.1.2 Résultat de l'implémentation du maillage

Ici nous allons discuter de l'implémentation de la partie algorithmique dure du projet, à savoir le graphe dynamique. Nous avons modélisé ce graphe sous la forme d'une classe Maillage qui contient en interne un graphe implémenté par la librairie externe JGraph (voir section « Outils Logiciels »).

L'implémentation du TDA maillage est définie par le cahier des charges suivant (description des méthodes) :

1. *Constructeurs* : La liste des différents constructeurs permettant de créer le maillage en fonction de différents paramètres. L'unité utilisée par default est le pixel, depuis la zone graphique selon les standards. Une exception est levée si les paramètres ne sont pas probables.
 - a. *+Maillage (int,int,int,int,int)* : ce constructeur public prend en paramètres dans l'ordre :
 - i. La largeur totale en pixel du maillage.
 - ii. La hauteur totale en pixel du maillage.
 - iii. Le côté du nœud en pixel. Un nœud est toujours carré.
 - iv. La position de départ du maillage dans la zone graphique en x.
 - v. La position du départ du maillage dans la zone graphique en y.
 - b. *+Maillage (int,int,int)* : mêmes paramètres que le constructeur précédent, avec la position du maillage par default en (0 ;0) .
2. *Méthodes d'éditions* : ces méthodes permettent d'éditer la structure du maillage de manière dynamique après l'avoir créé. Si le rectangle définissant les zones à éditer n'est pas dans la zone, une exception est levée. Méthodes « thread-safe ».
 - a. *+void activerZone(Rectangle)* : Permet d'activer une zone dans le maillage, c'est-à-dire de définir les nœuds contenus dans le rectangle passé en paramètre comme actifs et de les relier aux nœuds actifs adjacents. Le rectangle a une dimension en pixels relatifs à la zone de jeu.
 - b. *+void desactiverZone(Rectangle)* : Permet de désactiver une zone dans le maillage, c'est-à-dire de marquer comme accessibles les nœuds contenus dans le rectangle passé en paramètre et de supprimer les arcs dans le graphe relatifs à ce nœud. Le rectangle a une dimension en pixels relatifs à la zone de jeu.
3. *Méthodes de calcul* : Permet d'obtenir des informations sur le maillage, notamment le chemin le plus court d'un point à un autre. Méthode « thread-safe ».
 - a. *+ArrayList<Point> plusCourtChemin(int,int,int,int)* : C'est la méthode principale du maillage. Elle permet à partir des coordonnées des points de départ et d'arrivée de calculer selon le maillage le chemin le plus court, qu'elle retournera sous forme d'un ArrayList de points représentant les différentes étapes du chemin. Dans l'ordre des paramètres :
 - i. La coordonnée x du point de départ
 - ii. La coordonnée y du point de départ
 - iii. La coordonnée x du point d'arrivée
 - iv. La coordonnée y du point d'arrivée

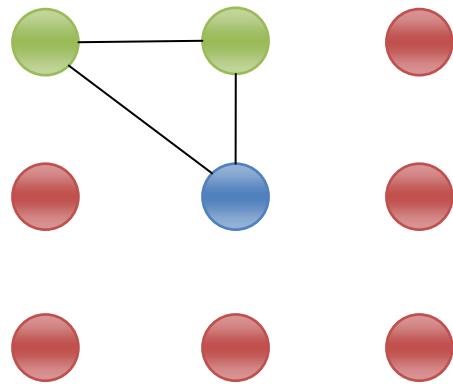
4. *Accesseurs* : Cet ensemble des méthodes permet de consulter l'état du maillage à n'importe quel moment, par exemple de récupérer la liste des noeuds internes au graphe. Nous ne détaillerons pas ici l'ensemble des méthodes, pour les détails voir la JavaDoc.

La représentation interne de la structure se fait selon le principe suivant. L'ensemble des noeuds est stocké dans un tableau, puis passé au graphe pour qu'ils soient organisés logiquement et qu'on puisse leur ajouter des arcs. Comme il s'agit de références passées, on peut accéder de manière équivalente aux noeuds par le graphe (que les noeuds actifs) et par le tableau interne du maillage. Cela permet d'ajouter et de supprimer dynamiquement des noeuds dans le graphe sans à avoir à les régénérer.

Pour la gestion des arcs, c'est-à-dire l'ajout lors de l'activation dynamique d'une zone ou à la création du maillage, on utilise l'algorithme suivant :

```
Pour un noeud cible :  
Ajout du noeud dans le graphe.  
Marquer le noeud comme actif.  
for tout ses voisins do  
    if le voisin est actif do  
        Ajout dans le graphe d'un arc cible <-> voisin.  
        Assignation du poids de l'arc, c'est-à-dire la distance en  
        pixel qui sépare les deux points.  
    End if  
End for
```

Il est essentiel de retenir que le graphe n'est donc pas orienté. La figure ci-contre illustre l'algorithme, le nœud bleu est le nœud cible, il est relié aux nœuds actifs et lui-même marqué comme actif. Cet algorithme permet d'utiliser la même méthode lors de l'initialisation des nœuds que lors de la réactivation d'une zone après avoir été désactivée.



Donnons également un petit point sur les unités. En effet, c'est un détail qui nous a posé soucis. Nous avons distingué deux types d'unités :

- Le pixel : C'est l'unité avec laquelle est créé, édité et lu le maillage. Les classes qui implémentent le maillage travaillent dans une zone de dessin en pixel, toutes les opérations prennent et retournent cette unité. Le maillage a conscience de sa dimension en pixel, de la largeur du nœud du pixel ainsi que du décalage par rapport à l'origine, toujours en pixel.
- La coordonnée nodale : C'est l'abstraction utilisée à l'intérieur de la classe Maillage pour représenter un nœud dans le tableau de nœud. Par définition une coordonnée nodale égale une largeur de nœud en pixel, à quelques subtilités près.

L'essentiel de la difficulté de travail avec des deux types d'unités (uniquement en interne), est qu'il a fallu non seulement définir une relation pixel->nodale, mais également la relation inverse. Nous avons utilisé l'algorithme suivant pour la conversion :

```
Pixel -> nodale :
for chaque coordonnée x et y en pixel do
    Calcul de la coordonnée modulo taille du nœud.
    Soustraction de cette valeur à la coordonnée.
    Division de la valeur obtenue par la largeur du nœud.
end for
Retour de x et y en nodal.
```

```
Nodale -> pixel :
for chaque coordonnée x et y en nodale do
    Multiplication de la coordonnée nodale par la largeur du nœud en
    pixel.
end for
Retour de x et y en pixel.
```

En effet, la coordonnée nodale converti en pixel représente le centre du nœud, tandis qu'une coordonnée en pixel est converti vers la coordonnée nodale la plus proche.

Résumons donc les propriétés du maillage :

- Graphe non orienté (pour permettre une navigabilité dans les deux sens).
- Graphe pondéré (pour permettre de représenter la distance entre les points pour l'algorithme de Dijkstra).
- Toujours un chemin entre le point de départ et d'arrivée.
- Edition dynamique du maillage par activation ou désactivation d'une zone.

- Lecture sur le maillage par une liste des arcs actifs.
- Lecture sur le maillage par une liste de points représentant le chemin le plus court entre un point A et un point B donné.
- Utilisation de méthodes de conversions pour les différentes unités.

Afin de faire fonctionner la classe Maillage, il y a quelques classes annexes (pour chaque classe voir la JavaDoc pour des informations plus détaillées) :

- **Arc** : Il s'agit d'un arc dans le maillage. Il étend la classe DefaultWeightedEdge (de JGraph) pour pouvoir être intégré dans le maillage et pour pouvoir jouer avec le poids de l'arc. Il contient également une référence vers le nœud de départ et le nœud d'arrivée. Une méthode permet également de le convertir en Line2D pour un affichage plus propre dans la zone graphique.
- **GenerateurDArcs** : Requis dans la structure du JGraph, permet de générer les arcs. Utilisé lors de la création du graphe dans la classe Maillage. Il en a été fait une classe externe pour une éventuelle réutilisation ultérieure.
- **Nœud** : Représente un nœud dans le graphe. Il étend la classe Point pour le stockage des coordonnées x et y en pixel et pour un affichage plus facile. Il contient des attributs comme son état (actif ou inactif), sa largeur en pixel, ainsi que quelques méthodes statiques de conversion de coordonnée et de calcul du centre du nœud en fonction d'une coordonnée x et y passée en paramètre.
- **PathNotFoundException** : Exception levée par le maillage si aucun chemin n'est trouvé. Classe externe pour permettre aux classes utilisant la classe maillage de traiter correctement cette exception.

4.2 Description des tests effectués

4.2.1 Génération du maillage (graphe associé) avec *JGraphT*

Pour ce test, nous avons mesuré le temps de génération du graphe associé à un terrain (une « map ») du jeu. Nous avons décidé d'effectuer ce test avec un graphe doté d'un nœud tous les 2 pixels, ainsi que 8 arcs par nœuds, répartis uniformément. Les nœuds sont tous reliés les un aux autres.

Voici un schéma du graphe généré pour un terrain de 4x4 pixels :

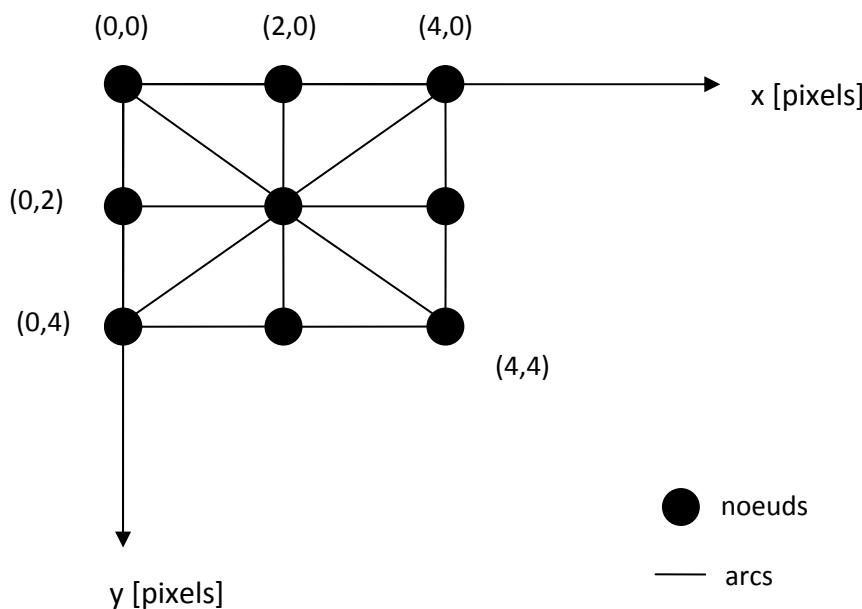


Figure 4.2.1 : graphe généré avec un nœud tous les 2 pixels.

Le résultat, après avoir pris les mesures correspondantes, figurent sous deux formes. Dans la première, on mesure le temps de génération du maillage en fonction d'une taille de terrain en pixels (par exemple 800x600). Dans la seconde, on mesure l'espace mémoire nécessaire en fonction toujours de la taille du terrain en pixels.

Voici les résultats obtenus, sous forme de graphiques :

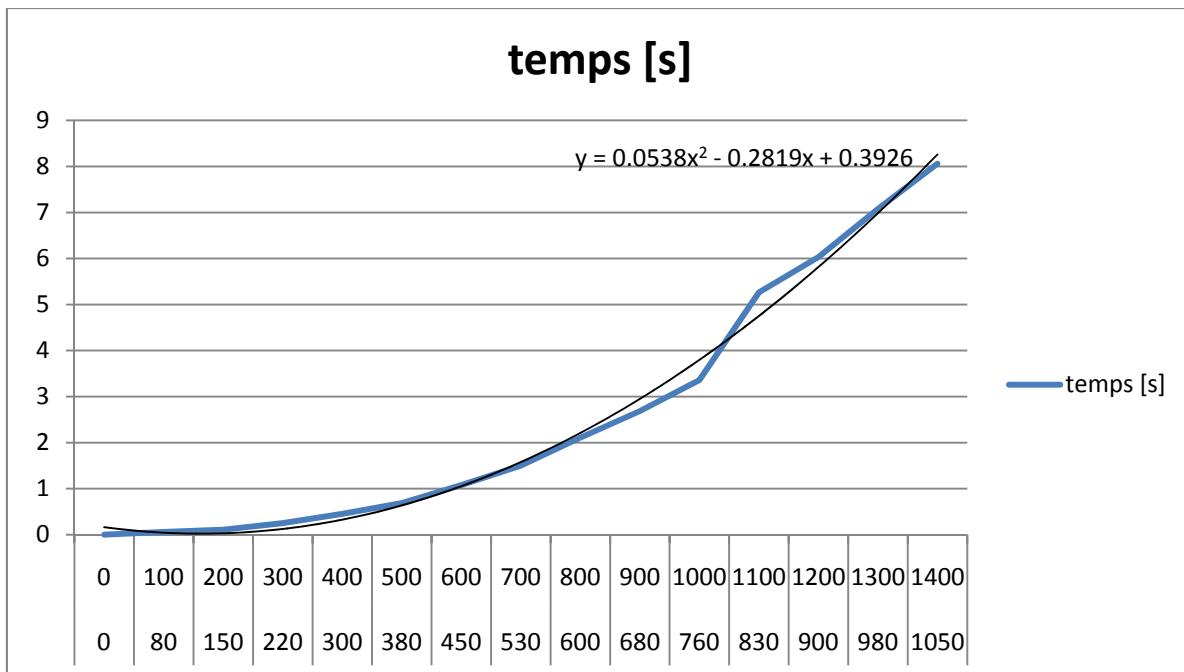


Figure 4.2.1.1 : temps de génération d'un maillage (graphe).

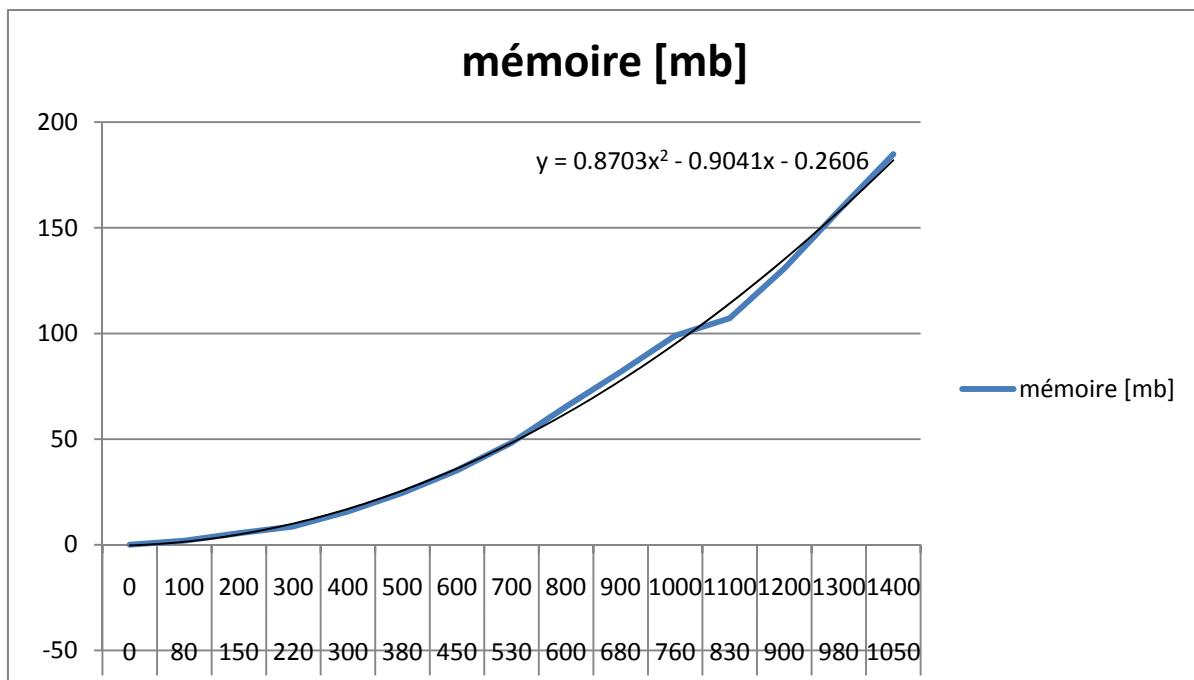


Figure 4.2.1.2 : mémoire utilisée lors de la génération d'un maillage (graphe).

En conclusion de ce premier test, on peut voir plusieurs choses intéressantes. Tout d'abord, rappelons que nous utilisons pour générer les maillages (graphes) une librairie externe, d'où l'utilité de ces tests, car nous ne sommes que des utilisateurs de cette librairie.

Premièrement, on voit sur le premier graphique que le temps de génération d'un maillage est en $O(n^2)$, grâce à la fonction y d'approximation. Il en va de même pour le second graphique, cette fois pour la mémoire. Cette première conclusion n'est pas surprenante, puisque pour créer le maillage, il faut créer n nœuds puis ensuite pour chaque nœud, il faut créer m arcs.

Deuxièmement, on voit qu'au niveau du temps de génération ainsi que de la mémoire, il va falloir choisir une valeur « correcte », c'est-à-dire une valeur optimale de la taille en pixels du terrain pour optimiser le temps et la mémoire lors de la génération du maillage, qui se fait au lancement de notre application. Nous avons donc choisi d'utiliser un terrain de jeu composé d'au maximum 500x500 pixels. En effet, d'après les graphiques, cela signifie que le temps de génération est d'environ 1 seconde et la mémoire utilisée d'environ 30 mégaoctets, ce qui est relativement raisonnable.

Cependant, pour baisser encore ces valeurs, nous avons décidé de créer finalement un nœud tous les 10 pixels, et non plus un nœud tous les 2 pixels. De ce fait, nous aurons un graphe qui aura 5 fois moins de nœuds et d'arcs que le graphe que nous générerons dans ce test (car pour une zone de 10x10 pixels, on avait avant 5 nœuds, alors que maintenant on en a plus qu'un). On peut alors s'attendre à un temps de génération d'environ 0.2 seconde ainsi qu'une mémoire utilisée d'environ 6 mégaoctets, ce qui est cette fois parfaitement raisonnable.

4.2.2 Recherche du chemin le plus court

Dans ce test, nous nous bornerons à faire des captures d'écran de l'application et à démontrer le bon fonctionnement de l'algorithme de recherche de chemin le plus court (*Dijkstra*).

Voici une série de trois captures d'écran illustrant les trois situations les plus intéressantes, suivies d'explications :



Figure 4.2.2.1 : terrain de jeu avec le chemin le plus court en bleu, situation 1.

Dans cette première situation, aucun obstacle n'est placé. Le chemin optimal est celui tracé en bleu. On voit nettement que ce chemin est le plus court. Cette situation est la plus triviale et ne nécessite pas de remarques particulières, si ce n'est que l'algorithme de recherche du chemin optimal fonctionne parfaitement.

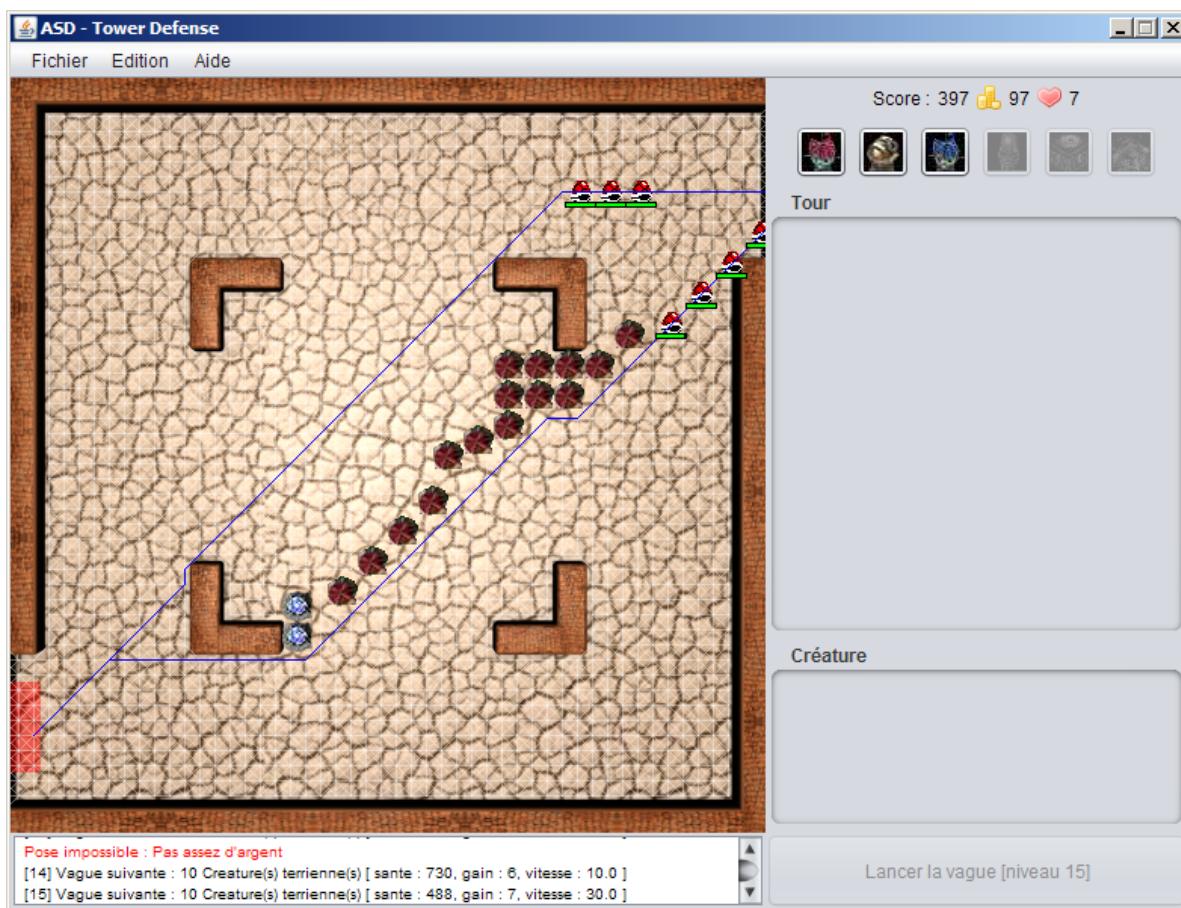


Figure 4.2.2.2 : terrain de jeu avec le chemin le plus court en bleu, situation 2.

Dans cette deuxième situation, il est intéressant de constater qu'il peut y avoir plusieurs chemins. En fait, c'est parce qu'on peut bouger dynamiquement les tours faisant office d'obstacle pendant le jeu. Ici par exemple, les trois premières créatures en haut avaient à la base un autre chemin optimal, en fonction des tours qui étaient placées à ce moment-là. Puis, après avoir supprimé/ajouté des tours, les quatre créatures suivantes se sont attribuées un nouveau chemin. Il est donc intéressant de voir que la recherche du chemin optimal est bel et bien dynamique : le chemin peut varier en tout temps. De plus, on constate que chaque créature possède son propre chemin optimal, toutes les créatures n'étant pas au même endroit (c'est-à-dire au même nœud en terme de graphe) à un temps donné t .

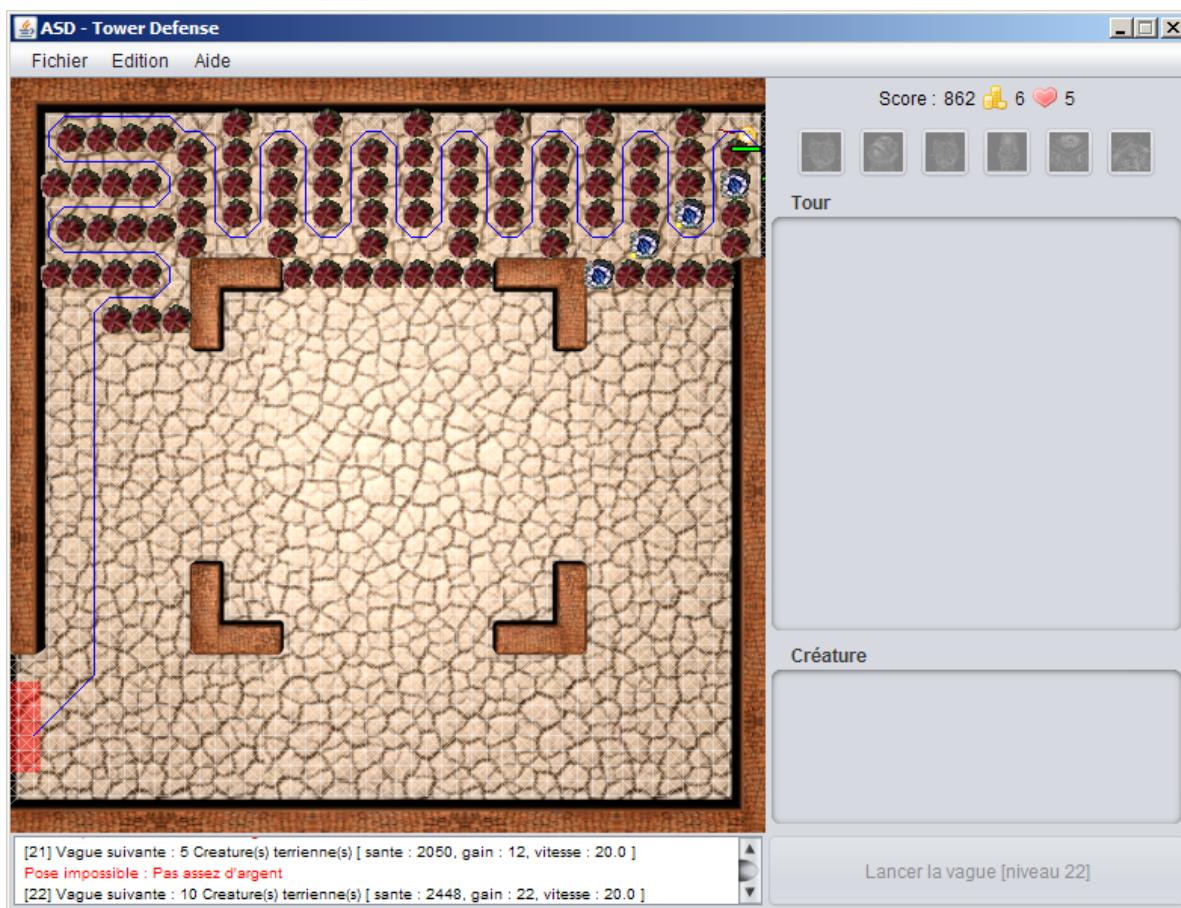


Figure 4.2.2.3 : terrain de jeu avec le chemin le plus court en bleu, situation 3.

Dans cette troisième et dernière situation, on voit que le joueur a pris le soin de créer une espèce de labyrinthe. Comme les créatures doivent se rendre du point de départ (en haut à gauche) jusqu'à l'arrivée (en bas à droite) coûte que coûte, elles prendront le meilleur chemin existant. En effet, ici elles seront obligées de parcourir tout le labyrinthe, car le chemin résultant, malgré qu'il soit long, devient un chemin optimal. De plus, on voit nettement qu'à la sortie du labyrinthe créé par le joueur, le chemin est à nouveau « livré à lui-même » car il n'y a plus aucun obstacle (d'où la ligne droite allant vers l'arrivée qu'on voit après la sortie du labyrinthe).

4.2.3 Enregistrement des scores et leur tri

Lorsqu'un joueur termine une partie, son score est sauvegardé dans un fichier « *sérialisé* » sur le disque dur dans le dossier */donnees*. Nous allons ici simplement tester cette fonctionnalité et fournir des captures d'écran démontrant son bon fonctionnement.

Voici un schéma de jeu classique :

1. Le joueur lance la partie et joue jusqu'à ce qu'il perde.
2. Le joueur a perdu ; il est invité à inscrire son nom.
3. Le joueur est ensuite invité à consulter la liste des dix derniers meilleurs scores classés par ordre décroissant (le meilleur score en premier), le score qu'il vient d'obtenir étant lui aussi inclus.

4. Sans le voir directement, les scores sont automatiquement sauvegardés dans un fichier sur le disque dur pour que le joueur puisse à tout moment revoir ses anciens scores.

Voici à présent trois captures d'écran destinées à démontrer les étapes 2, 3, et 4 décrites ci-dessus :

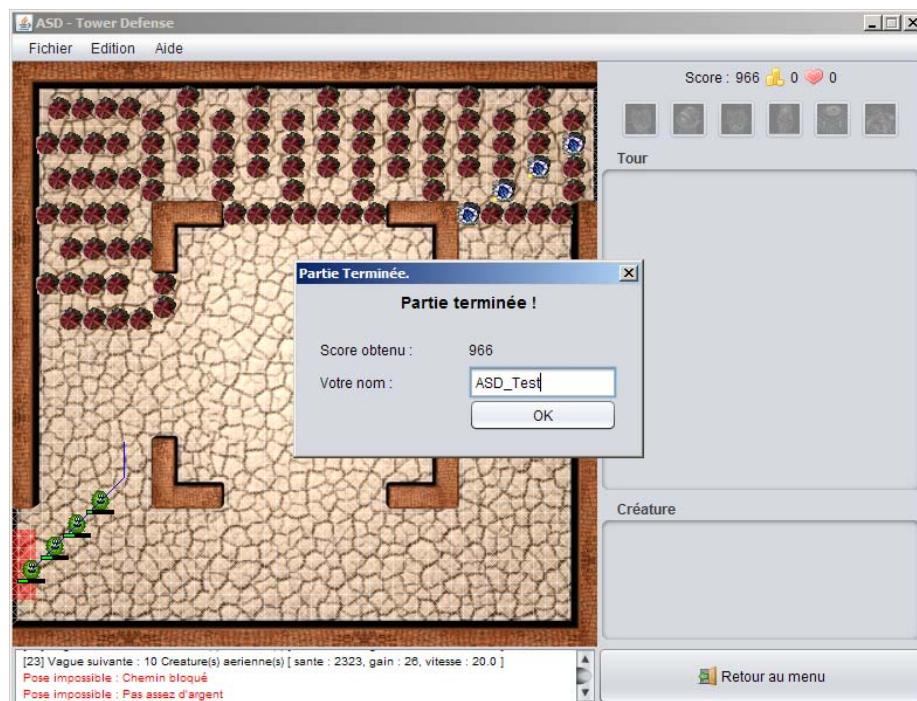


Figure 4.2.3.1 : le joueur a perdu ; il est invité à inscrire son nom. "

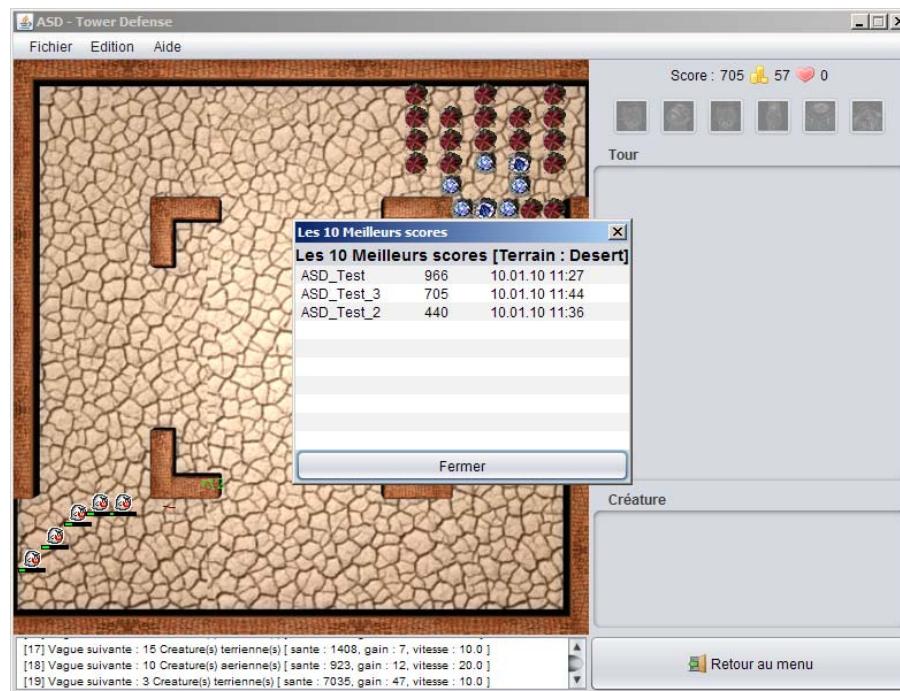


Figure 4.2.3.2 : le joueur voit la liste des 10 derniers meilleurs scores triés.

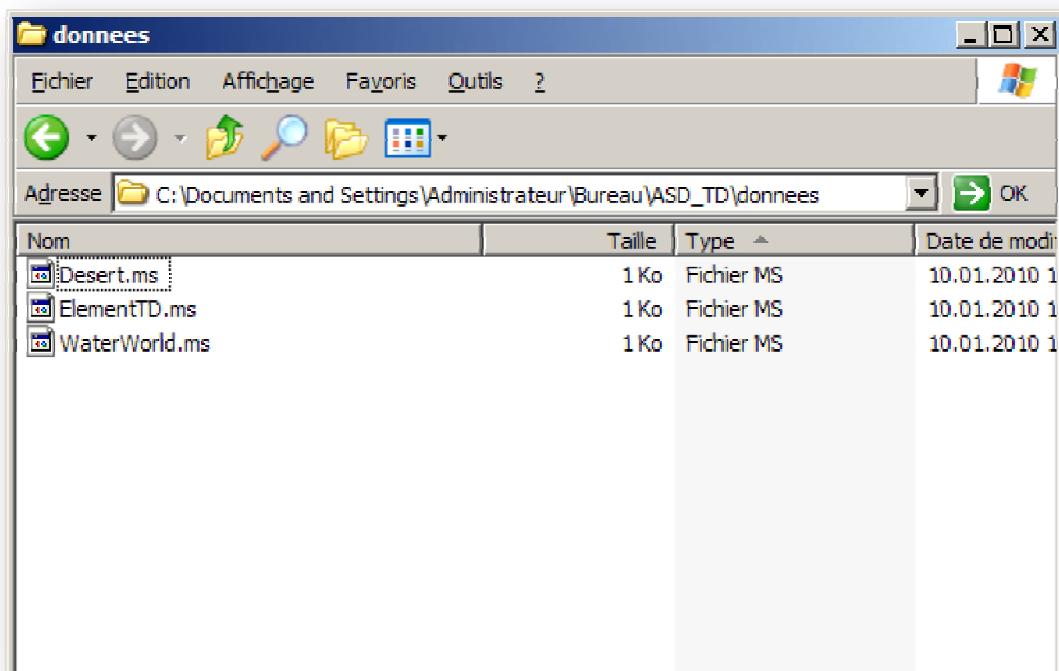
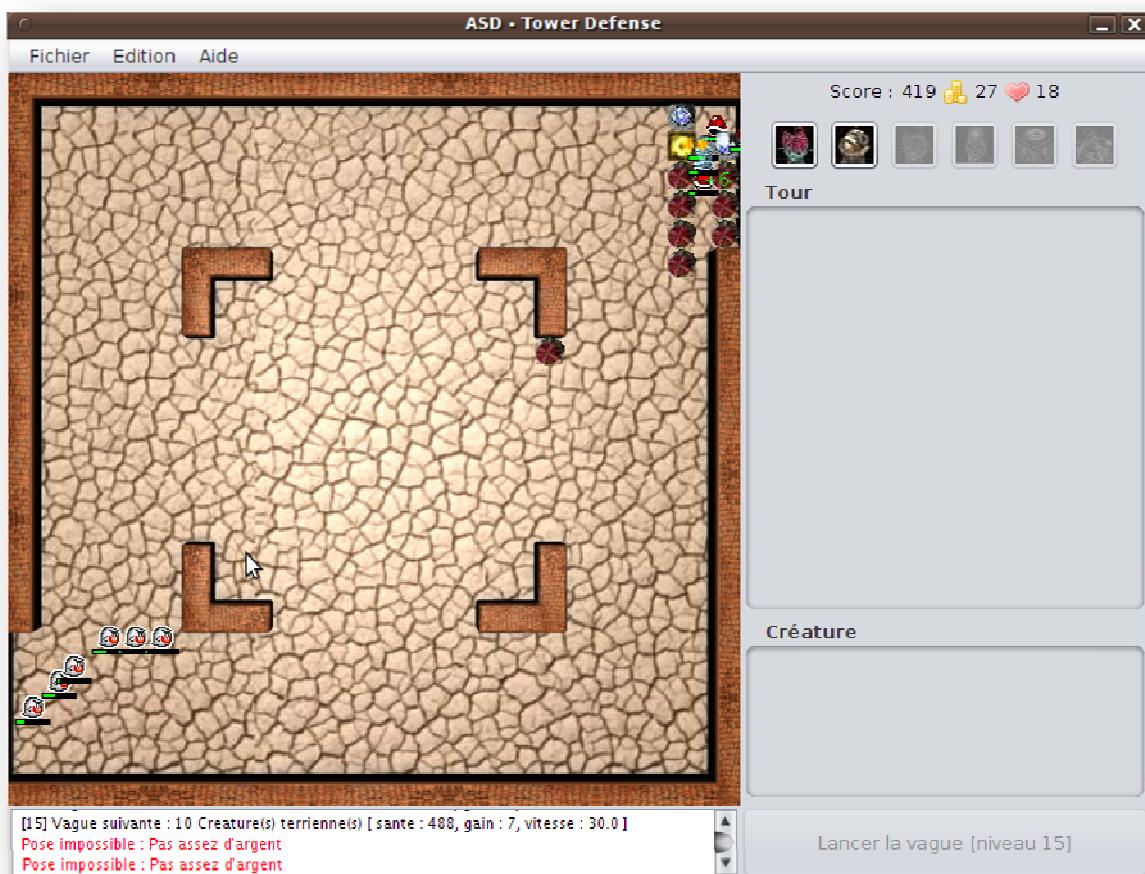


Figure 4.2.3.3 : Les scores sont sauvegardés dans le dossier /donnees.

4.2.4 Test sur un autre environnement

Nous avons testé notre application sur un environnement Linux. Comme elle est codée en Java, elle est de toute façon portable, car cette tâche revient aux concepteurs du langage. Malgré cela, nous avons jugé bon de le tester.

Voici une capture d'écran de l'application tournant sur une machine Linux Ubuntu 9.10 :



4.2.5 Test global de l'application

Nous avons décidé, peu de temps avant le rendu de ce laboratoire, de diffuser, en « *beta-test* », une version beta de notre jeu sur Internet, à l'adresse <http://code.google.com/p/asd-tower-defense/>. Ainsi, après avoir envoyé quelques e-mails, plusieurs personnes ont joué à notre jeu et nous ont fait des petits feedbacks.

Voici par exemple la capture d'écran qu'un contact nous a envoyé, pendant qu'il testait notre application :

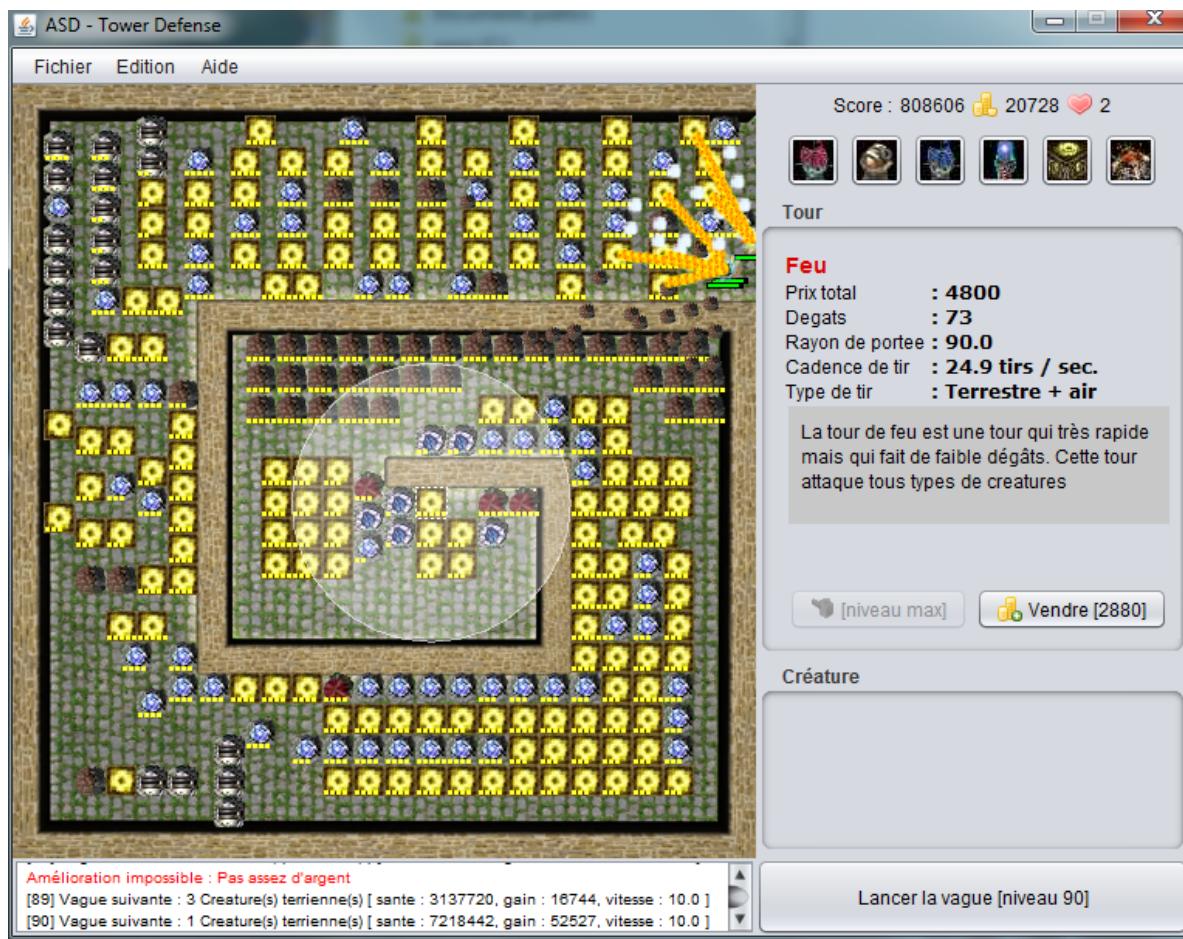


Figure 4.2.4.1 : Capture d'écran fournie par un de nos « beta-testeurs ».

Ce test comporte beaucoup d'informations intéressantes.

Tout d'abord, nous pouvons voir que le joueur est arrivé au niveau 90 sans bug ni crash de l'application, ce qui représente plus d'une heure de jeu. L'application est donc robuste à ce niveau-là.

Deuxièmement, on voit que le joueur a pleinement utilisé les fonctionnalités du jeu : il utilise tous les types de tours et d'améliorations possibles, ce qui est très positif pour nous car cela signifie qu'aucune des fonctionnalités n'est inutile aux yeux du joueur.

Troisièmement, nous pouvons voir qu'au niveau des chiffres, c'est-à-dire le score, les prix de vente des tours, les prix d'achat des tours, l'argent économisé ainsi que le nombre de vies, ils évoluent parfaitement bien avec l'avancement du jeu. Il n'y a effectivement pas de valeur aberrante ou totalement inadaptée.

Quatrièmement, en ce qui concerne la fenêtre de log, en bas, nous pouvons constater qu'elle est parfaitement à jour.

Enfin, ce joueur ainsi que plusieurs autres nous ont rapporté qu'à partir du niveau 100, certains bugs surviennent. Ceci est principalement dû à la gestion des threads qu'il faut encore améliorer, mais qui n'entre pas dans les objectifs principaux de ce projet. Pour plus d'informations, consulter les sections 5.3 – 5.4.

4.3 Erreurs restantes

Notre application ne comporte pas d'erreurs « graves », si on s'en tient au cahier des charges. Tout fonctionne parfaitement et nous en sommes très contents. Cependant, beaucoup d'améliorations sont à considérer (qui dépassent évidemment le cadre de ce projet) que nous détaillons dans la section 5.3 - 5.4. Il s'agit principalement du caractère *Temps réel* de l'application ainsi que de la gestion des threads. Mais ceci dépasse le cadre de ce projet, c'est pourquoi nous ne le considérons pas comme étant une erreur.

5 Conclusions

Mener à bien un projet n'est pas une chose aisée, et nous sommes très satisfaits de pouvoir conclure de manière très positive ce travail. En effet, si on s'en tient au cahier des charges initial, les objectifs sont largement atteints. En ce qui concerne l'organisation et la répartition des tâches ainsi que la collaboration des membres, une bonne entente a régné durant tout le projet. Nous sommes donc satisfaits et le résultat est au rendez-vous. Dans les sous-sections suivantes, nous développons en détails quelques conclusions importantes.

5.1 Objectifs atteints / non-atteints

Voici un récapitulatif des objectifs et de leur état, repris du cahier des charges initial.

Illustrer le concept de graphe de manière ludique et interactive.

L'objectif a été atteint avec succès.

Notre application permet de jouer tout en observant, si on le souhaite, le concept de graphe permettant aux créatures de se déplacer selon un chemin précis.

Acquérir de l'expérience dans la planification et l'accomplissement d'un projet conséquent.

L'objectif a été atteint avec succès.

Nous avons mené à bien cette tâche grâce notamment à l'utilisation du logiciel **svn** permettant de faire des versions à tout moment du travail en cours ainsi que d'organiser au mieux l'évolution du projet.

Utiliser et découvrir des librairies existantes implémentant le concept de graphe

L'objectif a été atteint avec succès.

En effet, nous avons pour ce projet utilisé deux librairies externes codées en Java. Elles nous ont été très utiles et nous avons appris à manier les librairies externes.

Apprendre à mettre en œuvre une interface graphique en Java.

L'objectif a été atteint avec succès.

Notre application n'aurait aucune raison d'être sans interface graphique. C'est le point-clé d'un jeu tel que celui que nous avons créé. Il va sans dire que ce point a été plus qu'approfondi.

Séparer le travail en plusieurs niveaux d'abstraction pour faciliter l'élaboration et l'évolutivité de ce projet.

L'objectif a été atteint avec succès.

En effet, nous avons réparti notre code en fonction de l'architecture bien connue **MVC**. Cette étape n'a pas été facile mais le résultat est très satisfaisant.

Comprendre la nécessité d'utiliser des algorithmes complexes dans les applications informatiques.

L'objectif a été atteint avec succès.

Une célèbre personnalité de l'informatique a dit un jour : « Algorithms + data structures = programs ». Cette citation est toujours d'actualité, simplement car sans algorithme, notre jeu n'aurait jamais vu le jour. Tout repose en effet principalement sur le fameux algorithme de recherche du chemin le plus court, plus connu sous l'appellation « *Algorithme de Dijkstra* ».

Mettre en œuvre un algorithme de recherche de chemin le plus court (ACPC).

L'objectif a été atteint avec succès.

L'algorithme ACPC (*Dijkstra*) décrit à la section 2.1.1.4 est l'algorithme-clé de notre application. Il est utilisé tout au long du jeu et est mis en œuvre avec succès.

5.2 Points positifs / négatifs

5.2.1 Points positifs

Les points positifs que nous tirons de ce projet sont les suivants :

- travail en groupe
- implication des membres
- motivation des membres
- état final du projet
- perspectives d'avenir du projet

5.2.2 Points négatifs

Les points négatifs que nous tirons de ce projet sont les suivants :

- projet très ambitieux au départ, difficile à mettre en œuvre
- manque de temps
- répartition des tâches difficile

5.3 Difficultés particulières

La principale difficulté que nous avons rencontrée dans ce projet réside dans l'aspect « *temps réel* » de l'application. En effet, comme nous n'avons pas encore tous les bagages nécessaires en ce qui concerne la gestion de la concurrence (et notamment les threads en Java), il a été difficile de s'auto-évaluer et d'apprendre à la volée ces notions. Néanmoins, nous n'avons pas hésité à solliciter les professeurs et assistants afin de poser nos questions. Au final, nous avons tout de même réussi à contourner ce problème et nous en sommes satisfaits. Toutefois, il est bien probable qu'à l'avenir cette couche de l'application soit entièrement revue.

5.4 Avenir du projet

Ce projet, ambitieux au départ, n'est pas sur le point d'être définitivement terminé. En effet, toute une série d'améliorations et de fonctionnalités pourraient être ajoutées. De plus, grâce à l'utilisation d'un logiciel de « *versionage* » avec son serveur associé, le projet peut sans problème rester ouvert. Les membres participant au développement peuvent même évoluer et on pourrait voir naître de nouveaux contributeurs. Tout le principe du développement logiciel en groupe « *Open Source* » est ici mis en œuvre grâce au logiciel **svn**. De ce fait, le projet reste complètement ouvert à toute perspective d'avenir.

Voici une liste non exhaustive des idées principales et améliorations qui pourraient survenir, par ordre de priorité décroissante :

- Revoir complètement la partie « *temps réel* » de l’application et supprimer la mauvaise gestion des threads pour n’en faire plus que deux : un pour la mise à jour de l’affichage et un pour le calcul des données.
- Corriger tous les bugs existants et ceux qui peuvent survenir.
- Améliorer l’interface graphique pour être encore plus convivial et simple.
- Faire une barre de progression lors du chargement d’un terrain.
- Pouvoir envoyer chaque nouveau score de chaque joueur sur un serveur afin d’avoir la liste de tous les meilleurs scores existant, et le afficher sur une page Internet.
- Séparer la difficulté du jeu en trois modes : facile, normal et expert.
- Pouvoir enregistrer la partie en cours et la charger à un moment ultérieur.
- Pouvoir mettre le jeu en mode pause.

6 Annexes

6.1 Sources – Bibliographie

- ✓ **Abdelali Guerid, Pierre Breguet, Henri Röthlisberger,**
[Algorithmes et structures de données avec Ada, C++ et Java](#)
- ✓ **JGraphT - a free Java Graph Library,**
<http://jgrapht.sourceforge.net/>
- ✓ **MP3 library for the Java Platform,**
<http://www.javazoom.net/javalayer/javalayer.html>
- ✓ **Unity**
<http://unity3d.com/unity/>
- ✓ **Listes d'un très grand nombre de « Tower Defense »**
<http://www.logiste.be/blog/125-tower-defense-games/>
<http://www.freegamesnews.com/fr/?cat=44>
- ✓ **Wikipedia du célèbre jeu « Warcraft 3 »**
http://fr.wikipedia.org/wiki/Warcraft_III
- ✓ **Wikipedia décrivant le concept des jeux « Tower Defense »**
http://fr.wikipedia.org/wiki/Tower_defense

6.2 Journal de bord de chaque participant

Voir annexe 6.2.

6.3 Manuel d'Utilisation

Voici un bref manuel du jeu afin d'expliquer comment l'utiliser. Attention, pour pouvoir lancer le jeu, il faut installer Java. Pour plus d'informations concernant son installation, aller sur <http://java.com/fr/>.

6.3.1 Lancer le jeu

Pour lancer le jeu, il suffit, à partir du CD, de double-cliquer sur l'icône correspondante au système d'exploitation. Par exemple, pour Windows, il suffit de double cliquer sur le raccourci ASD_TD_Windows se trouvant à la racine du CD.

Si le système d'exploitation est autre que Windows ou Linux, il faut se rendre dans le dossier game_data/ se trouvant à la racine du CD, puis il suffit de taper, dans une console :
`java -jar ASD_TD.jar`

6.3.2 Choisir un terrain

Une fois le jeu lancé, le joueur est invité à choisir un terrain. Il suffit de cliquer sur l'un des quatre boutons représentés par une petite image carrée afin de lancer le terrain correspondant.

6.3.3 Jouer

Une fois le terrain choisi, le jeu se lance et la partie commence. Il suffit alors de placer des tours, puis de lancer les vagues de créatures avec le bouton se trouvant en bas à droite. Différents boutons peuvent s'activer au cours du jeu dans le panneau de droite. Ils permettent de vendre, d'améliorer ou d'acheter des tours.

6.3.4 Raccourcis clavier

Voici la liste des différents raccourcis claviers :

- v : permet de vendre une tour une fois celle-ci sélectionnée
- a : permet d'améliorer une tour une fois celle-ci sélectionnée
- barre espace : lancer la vague suivante

6.3.5 Couper le son

Il suffit de cliquer sur le menu Son -> activer ou Son -> désactiver si on souhaite activer ou désactiver tout entité sonore. Ce menu n'est disponible que dans la fenêtre de jeu, car dans le menu principal, il n'y a de toutes façons pas de son.

6.3.6 Fin du jeu

Le jeu se termine une fois que le joueur n'a plus de vies (le nombre de vie restant s'affiche en haut à droite à côté du petit cœur pendant le jeu). A chaque fois qu'une créature atteint l'arrivée, le joueur perd une vie.

Lorsque le jeu se termine, le joueur est invité à inscrire son score afin que celui-ci puisse être sauvegardé automatiquement. Il ne reste plus qu'à quitter le jeu ou à recommencer une partie.

Attention, si on ouvre le jeu directement depuis le CD, on ne pourra pas enregistrer les scores, car les données ne peuvent pas être sauvegardées sur un CD comme si c'était un disque dur. C'est pourquoi il est conseillé de copier le contenu du CD dans un répertoire local puis de lancer le jeu directement depuis ce dossier.

6.3.7 Consulter les scores

Lors du lancement de l'application, aller dans fichier -> scores puis choisir un terrain. Les scores obtenus avec le terrain sélectionné s'affichent alors, triés par ordre décroissant.

6.3.8 Quitter le jeu

Pour quitter le jeu, il suffit de cliquer sur la petite croix en haut à droite de la fenêtre, ou de sélectionner fichier -> quitter.

6.4 Archives du projet

6.4.1 CD

Le contenu intégral du projet se trouve dans le CD fourni en annexe. Il contient notamment les sources Java, les exécutables, les images, sons et icônes, les raccourcis ainsi que les différents lanceurs (.bat, .sh, etc.).

6.4.2 Internet

Le contenu du projet peut également être retrouvé sur le dépôt svn prévu à cet effet à l'adresse suivante : <http://code.google.com/p/asd-tower-defense/>