

Le Mans Université
Licence Informatique *2ème année*
Rapport de projet
Necrew Arena
<https://github.com/mariusvitta/tacticsarena>

Vivien-Junior Obanda, Dylan Renaudin, Alexandre Danjou et Marius Vitta

23 avril 2019

Table des matières

1	Introduction	3
2	Organisation du travail	3
3	Conception	4
3.1	Analyse	4
3.2	Cahier des charges	5
4	Développement	5
4.1	Jeu version terminal	6
4.2	Intelligence artificielle	8
4.3	Réseau	8
4.4	Interface graphique	9
4.5	Méthodes et outils de développement	12
5	Résultats	13
5.1	Version terminal	13
5.2	Intelligence Artificielle	13
5.3	Réseau	14
5.4	Interface graphique	14
6	Conclusion	15

1 Introduction

Dans le cadre de la *Conduite de projet*, un module du second semestre de deuxième année de licence informatique, il nous a été soumis un travail de développement et de programmation en équipe favorisant l'application de nos connaissances ainsi que l'apprentissage de nouvelles notions. L'objectif fixé était de développer *Tactics Arena*, un jeu de stratégie sur plateau au dessus duquel deux équipes opposées combattent au tour par tour. Nous avons donc formé un groupe de quatre étudiants composé de Vivien-Junior Obanda, Dylan Renaudin, Alexandre Danjou et Marius Vitta. Par ailleurs, nous avons souhaité revisiter certaines règles classiques de ce jeu dans l'optique d'y apporter une contribution personnelle. Ainsi, notre jeu, nommé *Necrew Arena*, repose majoritairement sur les mêmes principes soient un plateau de jeu, deux équipes, des personnages, des sorts, un combat au tour par tour ... La différence majeure réside dans le nombre de personnages et de sorts que possèdent chacun des joueurs. Par opposition aux règles du jeu traditionnel, dans lequel chaque équipe est composée de plusieurs personnages dotés d'un seul sort chacun, *Necrew Arena* oppose deux équipes formées de deux personnages au maximum. En revanche, ces derniers ont droit à quatre sorts bien distincts répartis selon la classe du personnage choisi. Dans la suite de notre rapport, nous expliquerons de manière détaillée les étapes nécessaires à la création de notre jeu, avant d'auto-évaluer notre travail et de présenter des points d'amélioration possibles à l'avenir.

2 Organisation du travail

Notre priorité, lors de la conception de notre jeu, était de réussir à faire une version fonctionnelle sous console. Pour ce faire, Dylan Renaudin s'est occupé de la création des différentes structures de données, de la gestion des déplacements et des sorts de saut, de soin, de transformations et de défense. L'initialisation de la partie et la gestion des tours de jeu ont été pris en charge par Marius Vitta. Vivien-Junior Obanda s'est employé à programmer les différents affichages, la fonction de mise à jour et les sorts d'attaque en diagonale, en ligne et de gros impact. Tous les autres sorts ont été implémentés par Alexandre Danjou.

Une fois cette version terminée, nous nous sommes chacun chargés de parties différentes. Vivien-Junior s'est occupé de l'interface graphique, Alexandre du réseau, Marius de l'Intelligence Artificielle et Dylan à la modification du code afin de le rendre plus générique. Pour ce qui sont des outils communs, nous nous sommes servis du *Git* et d'*Atom*. Ces outils nous ont permis de mettre en commun nos branches du code développées séparément. Concernant la communication, nous avons utilisé *Discord* et *Whatsapp*.

3 Conception

Les étapes de conception ont été primordiales dans l'avancement de notre projet, de l'analyse à la définition du cahier de charges.

3.1 Analyse

Comme dit précédemment, nous avons décidé de remodeler les règles *Tactic Arena* de manière à rajouter des éléments plus personnels.

Tout d'abord, nous nous sommes renseignés sur les règles d'un jeu de base. Ce dernier consiste à opposer deux équipes de personnages ayant des rôles bien définis sur une carte de type échiquier dont la taille plus ou moins grande. Des soigneurs sont chargés de rendre des points de vie aux personnages. Par ailleurs, des attaquants s'occupent de tuer les personnages de l'équipe adverse. Chaque personnage, à chaque tour et une seule fois, peut se déplacer ou attaquer. L'utilisateur choisit l'ordre dans lequel il souhaite jouer ses personnages. Notre jeu reprend le même schéma, à savoir un affrontement joueur contre joueur, chacun possédant plusieurs personnages. Le but reste le même, en l'occurrence, tuer tous les personnages de l'équipe adverse tout en conservant les siens en vie. Cependant, l'ordre des tours a été changé. En effet, dans notre cas les personnages de chaque équipe jouent de manière alternée dans un ordre défini dès le début de la partie. Si l'un des personnages meurt, l'ordre est conservé et le tour de celui-ci est passé. De plus, chaque joueur ne dispose que de deux personnages. À défaut d'une équipe pouvant aller jusqu'à des dizaines de personnages pour certains *Tactic arena* classiques, nous avons pour les deux personnages du joueur :

- un nom de classe correspondant à la classe choisie par le joueur pour son personnage, il existe quatre classes différentes ;
- des points de vie permettant de connaître l'état d'un personnage, chaque classe de personnage a des points de vie spéciaux pour sa classe ;
- des points de mouvement lui permettant de se déplacer à chaque tour, bien que ces points soient fixés à trois par joueur, une des classes peut augmenter ce nombre grâce à un de ses sorts ;
- des points d'action permettant de savoir combien de sorts le personnage peut encore lancer ;
- des coordonnées donnant sa position sur la carte ;
- quatre sorts différents suivant sa classe.

Nous avons donc créé seize sorts en tout. Parmi eux, nous comptons un sort servant au déplacement, deux utiles pour les soins, trois apportant une modification de statistiques aux personnages, et les autres infligeant des dégâts. Pour chaque sort, nous avons un nom de sort, une portée, des dégâts,

un nombre de points d'action nécessaires à leur utilisation ainsi qu'un nombre d'utilisation par tour.

Afin de pouvoir récupérer, modifier, et utiliser toutes ces valeurs, nous utilisons des structures de données pour les personnages ainsi que les sorts. La carte de jeu est représentée par une matrice de caractères. Celle-ci affiche différents caractères en fonctions des éléments qui s'y trouvent. Si c'est un personnage, elle présente un chiffre, si c'est un obstacle fixe bloquant le déplacement ou certains sorts elle affiche un « o » et si c'est une case libre, un « . ». Nous avons aussi observé qu'il nous faudrait différentes fonctions, notamment des fonctions d'affichage de la carte, de gestion de déplacement des personnages, de vérification de l'état d'un ou plusieurs personnages (morts ou en vie) à un tour donné, de gestion de tour de jeu, de mise à jour de la carte, de fin de partie, de création et de suppression des personnages et des sorts, d'initialisation ainsi que des fonctions pour chacun des sorts.

3.2 Cahier des charges

Lors du démarrage du jeu, les joueurs choisissent leurs classes parmi toutes celles disponibles. Après cela chacun à leurs tours placent leurs personnages en fonction des choix proposés sur le plan de jeu. À noter que les choix de positionnement sont limités à 3 cases et sont générés aléatoirement à chaque partie. Une fois les personnages placés, une dizaine d'obstacles apparaissent aléatoirement sur la carte de jeu. Puis vient la boucle principale de tour de jeu dont la condition de fin de partie est la mort de tous les joueurs d'une équipe. Le personnage a le choix d'effectuer une action ou de passer son tour sans subir aucune perte. À la fin de chaque tour de jeu les coordonnées de tous les personnages ainsi que leurs caractéristiques (points de vie, noms ...) sont affichées afin d'avoir une idée sur l'évolution de la partie. À la mort d'un personnage, celui-ci est supprimé de la carte et ses points de vie sont mis à -1. Ainsi le jeu se poursuit sans prendre en compte le tour des personnages morts.

À la fin de la partie le numéro de l'équipe qui a perdu est annoncé puis la suppression des classes, des sorts et des personnages est effectuée.

4 Développement

Le développement représente l'étape majeure de notre projet. Différentes versions de jeu ont été établies afin que celui-ci possèdent de nombreuses fonctionnalités. Il existe donc la version terminal sur laquelle reposent toutes les autres versions, le jeu en réseau, une intelligence artificielle et le jeu en interface graphique.

4.1 Jeu version terminal

Avant de pouvoir créer un jeu en réseau ou en SDL, nous devions commencer par une version du jeu sur le terminal. Cette version est la première que nous avons faite. Nous sommes partis de ce que l'on avait énoncé dans le cahier des charges et l'analyse afin de nous diriger dans notre codage.

Nous avons débuté par la génération de la carte. Il s'est agi de créer une matrice de caractères de taille de onze lignes et onze colonnes (11x11) car nous n'avons que peu de personnages à afficher. Nous avons créé les structures de personnages contenant des sorts ainsi que les structures d'équipes. La structure équipe contient deux personnages, et la structure personnage contient quatre sorts comme présenté ci-dessous :

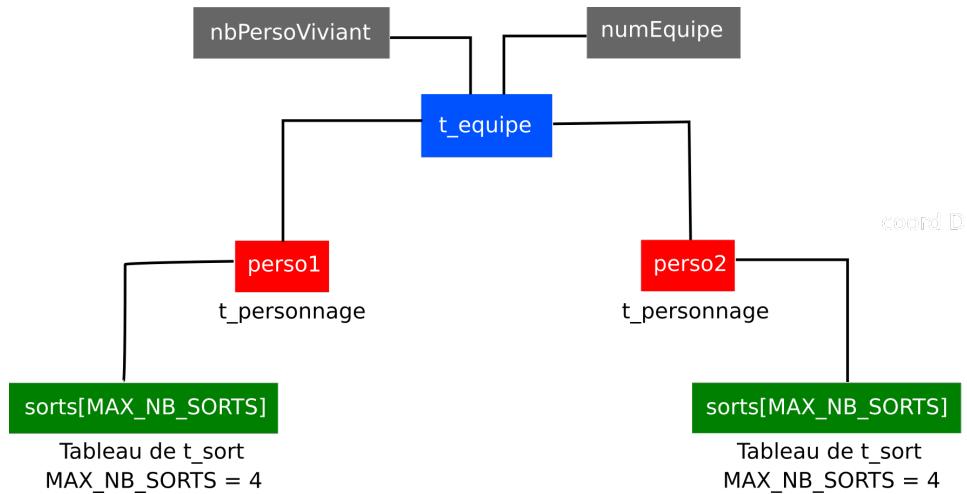


FIGURE 1 – Schéma de composition d'équipe

Ces structures contiennent tous les paramètres dont ces éléments de jeu ont besoin. Une fonction a été pensée afin de créer tous les sorts existants. Ainsi, nous récupérons les valeurs dans un fichier d'extension .txt modifiable à tout moment afin d'avoir plus de sorts à l'avenir voire si le besoin de modifier certaines valeurs dans un but d'équilibrage se présente dans l'optique d'un meilleur déroulement du jeu. Nous avons utilisé la même méthode de création pour les différentes classes. La structure de sort comprend également un pointeur sur fonction capable de lancer les sorts de la même façon pour chaque personnage sans avoir à créer de boucle ou une autre procédure quelqu'elle soit. En outre, nous avons créé toutes les fonctions de sorts dont les personnages ont besoin, il en existe seize. Il est à noter que chacune de ces fonctions a exactement le même prototype que toutes les autres afin de pouvoir utiliser le même pointeur sur fonction au lancement du sort. Tous

ces paramètres sont énumérés ci-dessous :

- une matrice de caractères, donc la carte de jeu ;
- une structure personnage, qui représente le personnage qui lance le sort ;
- deux structures équipes afin de pouvoir interagir avec les autres personnages ;
- un numéro de personnage dans l'équipe ;
- un numéro d'équipe ;
- les dégâts du sort ;
- la portée du sort.

Il existe 4 catégories de sorts bien distincts :

- Les sorts modifiant la position des personnages : *Saut, Attire*.
- Les sorts infligeant des dégâts aux personnages adverses : *Ptitcoup, GrosCoup, Diago, Ligne, Doubletape, Coupzone, ChenChen, Big-Shaq, Fuego*.
- Les sorts modifiant les statistiques du personnage, tant bien sur les points d'action que sur les points de mouvement : *Armure, Transformation_mino, Transformation_felin*.
- Ceux permettant de soigner : *Soin, Revitalisation*.

Une fonction d'initialisation a été implémentée afin que les joueurs choissent où ils souhaitent placer leurs personnages. Dans cette fonction, nous créons également des obstacles nécessaires pour bloquer certains sorts ainsi que les déplacements des personnages. Nous affichons ensuite la carte à l'aide d'une fonction affichage qui présente simplement tous les caractères de la matrice grâce à deux boucles *for* imbriquées (*cf. figure 4*).

La fonction déplacement permet à l'utilisateur de taper la direction dans laquelle il souhaite aller, h pour le haut, b pour le bas, g pour la gauche et pour la droite. Le personnage s'y déplace case par case et a droit à trois déplacements. Quant à la fonction de mise à jour de la carte, elle récupère la position de tous les personnages ainsi que leur état (mort ou vivant) et place les caractères représentant les personnages à leurs coordonnées respectives dans la map pour l'affichage. De plus, nous avons implémenté une fonction qui gère le tour de jeu. Elle nous fait choisir les différentes actions, évoquées en amont, que l'on veut effectuer comme se déplacer ou lancer un sort (*cf. figure 5*). Elle gère les points d'action et de déplacement des personnages en les décrémentant quand il faut mais peut également les incrémenter pour les transformations par exemple.

Pour finir, afin de réduire la consommation d'espace du programme, une fonction de suppression a été créée. Elle libère toutes les allocations mémoire faites notamment pour les personnages et les sorts avant de remettre

les pointeurs à NULL.

4.2 Intelligence artificielle

En ce qui concerne l'intelligence artificielle, nous avons quelque peu manqué de temps dans le développement. Il est à noter que nous ne savions pas très clairement quel était le type d'algorithme le plus approprié à notre jeu, étant donné que les parties s'effectuent au tour par tour avec des personnages dotés de plusieurs sorts. Il nous a alors été proposé de développer un algorithme Min-Max.

En effet, l'algorithme Min-Max repose sur le choix du meilleur coup possible. Pour ce faire, il a recours à une fonction d'évaluation pour le joueur et une autre pour l'adversaire. Ces deux fonctions mutuellement récursives sont utilisées de la manière suivante : s'il existe un coup gagnant, il est retourné pour être joué par la suite, sinon tous les coups possibles sont évalués et seul le meilleur est renvoyé.

L'une des fonctions, joueur ou adversaire, tente d'obtenir le maximum des coups possibles, inversement, l'autre fonction est focalisée sur leur minimum. Nous avons donc adapté cet algorithme à notre jeu en y apportant quelques modifications afin de le simplifier. Nous avons exclu l'étape d'évaluation du meilleur coup de l'adversaire dans l'obtention du potentiel coup gagnant du joueur. Ainsi dans la fonction d'évaluation, nous utilisons principalement la portée, elle nous permet de vérifier qu'un sort est utilisable ou non. Si le sort peut être lancé par l'IA, nous recherchons l'adversaire le plus proche avant de calculer la distance qui les sépare. Enfin cette distance est comparée à la portée de notre sort. Si la distance entre les deux personnages est inférieure à notre portée, nous utilisons le sort. Toutefois, le choix de ce dernier n'est pas anodin car l'algorithme lance le sort qui inflige le plus de dégât parmi tous ceux qui ont la même portée. En revanche, si la distance est supérieure à cette portée, nous vérifions si un déplacement est préférable.

Nous avons décidé de dissocier l'évaluation du meilleur sort possible de celle du déplacement afin de décomplexifier notre algorithme. Le personnage se déplace également vers l'adversaire le plus proche. S'il en existe deux à distances égales, il se rapproche de l'ennemi ayant le moins de points de vie. Ce déplacement est basé sur la fonction *déplacement* du programme principal. Il est effectué case par case afin de trouver la direction vers le point d'arrivée. Nous avons créé un type énuméré comprenant les quatre points cardinaux. Par ce biais, nous facilitons la compréhension du code et, dans la même optique, nous l'optimisons.

4.3 Réseau

Pour ce qui est du réseau, nous avons choisi un modèle client-serveur. Le client n'est donc pas également un hôte. Ce choix nous a permis, par la

suite, de gérer les mises à jour du plan de jeu uniquement du côté du serveur, ainsi nous excluons la possibilité que des joueurs mal intentionnés ne trichent en envoyant un plan de jeu erroné au serveur. Concernant les fonctions, nous avons dû en adapter plusieurs déjà existantes, il s'agit notamment des sorts. Nous avons rajouté des paramètres et créé des variables globales. Toutefois, certaines fonctions telle que celles de gestion des déplacements ou de tour de jeu ont été entièrement réimplémentées afin d'y rajouter des vérifications utiles à la communication réseau. Nous avons également créé des fonctions serveur et client faisant toutes deux appel à leur tour à d'autres fonctions nouvellement implémentées, afin de choisir l'objet que l'on souhaite envoyer ainsi que le destinataire de cet envoi. C'est le cas de la fonction *send_all* qui transmet des informations à tous les joueurs à l'exception d'un client. Elle est d'une grande aide lorsque l'on souhaite transmettre des informations sur le joueur dont c'est le tour. Ainsi il ne sera pas envoyé à ce dernier une instruction qu'il sait déjà.

La fonction serveur a donc pour but de mettre à jour le plan de jeu, d'exécuter la partie, d'envoyer les informations aux clients et de vérifier les réponses reçues par ceux-ci. Par ailleurs, la fonction client sert à recevoir les informations du serveur et de mettre le client en attente. Inversement, elle permet également au client d'envoyer, au serveur, les actions qu'il souhaite effectuer.

4.4 Interface graphique

Necrew Arena se voulant plus ouvert et accessible à tout utilisateur, nous lui avons attribué une interface graphique en supplément de la version console. Pour ce faire, nous nous sommes appuyés sur les bibliothèques SDL plus particulièrement sur les bibliothèques *SDL_images* et *SDL_ttf*. Afin de préparer au mieux le codage de toute fonction à venir, nous avons implémenté des fonctions nous permettant de créer et d'initialiser la fenêtre, les polices ainsi que toutes les textures. Notre code a évolué, continuellement au fil des besoins qui se sont présentés lors de l'implémentation et a découlé sur la programmation des divers affichages et sur l'adaptation des fonctions existantes afin qu'elles soient utilisables depuis notre interface graphique. De ce fait, des procédures et des fonctions, autres celles concernant les affichages et les fonctions de jeu, ont été mises en oeuvre et font office d'outils.

Affichages : De nombreux affichages sont effectués pour parvenir au rendu final du jeu vu par l'utilisation. À L'ouverture du jeu, nous observons une image d'accueil avec le logo de *Necrew Arena* au centre de la fenêtre. Deux images ont été choisies et sont affichées aléatoirement une par une à un instant précis (*cf. figure 8*). Par la suite il est proposé aux joueurs de choisir les personnages qui composeront leurs équipes puis de choisir l'emplacement de ces derniers. L'utilisateur est guidé par des instructions apparaissant en bas

de la fenêtre lors du choix des personnages et en haut à droite de celle-ci lors du choix des emplacements. Les propositions de case conduisant à ces choix d'emplacements reposent sur l'affichage de la matrice de jeu. Contrairement à la version console, l'affichage du plan de jeu se fait ligne par ligne en ordre décroissant sur l'interface graphique, en d'autre termes, elle s'effectue de la dernière à la première colonne de la première ligne et pareillement pour les lignes suivantes. La procédure des deux boucles *for* imbriquées utilisées pour l'affichage sur le terminal est la même, à la différence que la seconde boucle est inversée. Cet algorithme d'affichage favorise le respect de la superposition des images. De plus, nous avons un plan de jeu isométrique construit à l'aide de cases en formes de losanges. Afin de préserver les coordonnées de ces cases, une structure *t_case* a été mise en place. Cette structure récupère le type de la case. Ce type peut correspondre à une case de portée (case bleue), une case libre (case grise), un obstacle ou encore une case occupée par un personnage. La structure contient également la position de la case (sa ligne et de sa colonne) dans la matrice de jeu ainsi que les coordonnées, sur la fenêtre de l'interface, des quatre points délimitant cette case.

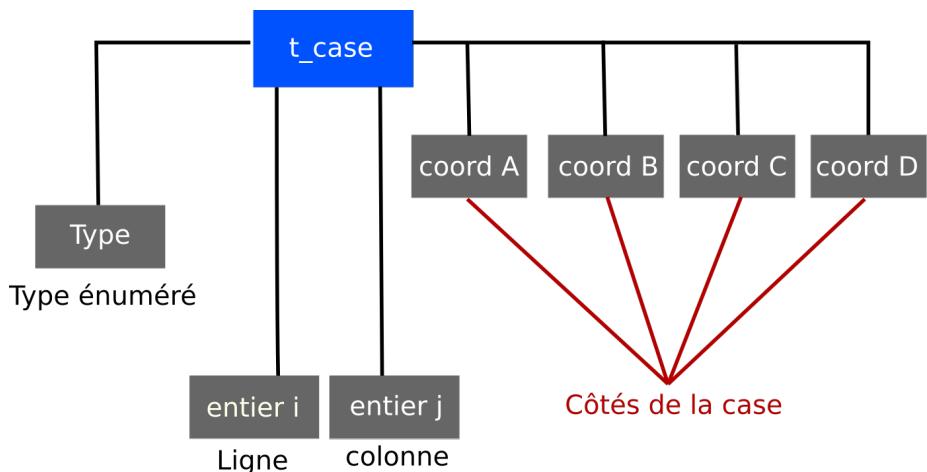


FIGURE 2 – Schéma de la structure *t_case*

Des fonctions d'affichages de menu ont également été implémentées. Il s'agit principalement du menu proposant à l'utilisateur d'effectuer des actions (se déplacer, utiliser un sort ou simplement passer son tour), du menu présentant les différents sorts en fonction du personnage et le menu de fin de partie avec les options *rejouer* et *quitter le jeu*. Le point commun entre ces menus est qu'ils reposent tous sur des boutons de couleur grise ou bleue selon l'état de la sélection. Si une option est choisie le bouton s'affiche en bleu sinon en gris. Afin de préserver les informations de ces options, nous

nous aidons d'une structure *t_choix* qui comprend le numéro de celui-ci, ainsi que les coordonnées de trois points suffisants pour délimiter son bouton d'activation. Ainsi lors d'un clic la vérification est faite de façon plus simple et optimale.

Adaptation des fonctions en variante SDL : Outre les fonctions d'affichages, il a été nécessaire d'adapter voire de réimplémenter certaines fonctions déjà existantes en version console. D'abord les fonctions d'initialisation et de déplacement. Elles ont été implémentées de telle sorte que les données du clic opéré par l'utilisateur soient traitées. Des cases bleues, représentant des cases accessibles sont placées sur le plan de jeu en attendant de recevoir un clic. Une fois ce dernier effectué, nous vérifions qu'il est bien compris dans le périmètre de la case accessible. C'est ainsi que sont fait les choix des cases d'emplacement ou de déplacement. L'utilisation des sorts reposent sur le même principe. En effet, pour certains sorts notamment les sorts d'attaques, des cases bleues de portée de frappe sont affichées et le choix revient à l'utilisateur. Un effet de clignotement a été rajouté sur le personnage subissant le sort pour plus de dynamisme et les dégâts affichés au dessus de sa tête. Par ailleurs, concernant les sorts de transformation les images représentant les personnages sont changées en fonction de leur nouveau statut : par exemple, pour la transformation Minotaure du Druide, l'image du druide est remplacée par celle du minotaure. Par conséquent le jeu paraît plus réaliste. À noter que la fonction principale *main* a été complémentairement réimplémentée et nous avons rajouté une option *rejouer* en fin de partie.

Outils favorisant la mise en place de l'interface : D'autres éléments ont également servi au développement de l'interface graphique : il s'agit notamment de fonctions. Nous avons développé une fonction permettant d'écrire du texte à un emplacement précis de la fenêtre selon une police, une taille et un style donnés. Une autre fonction nous a permis de faire clignoter un joueur. Cet effet nous sert lorsque nous infligeons des dégâts suite à l'utilisation d'un sort. Des fonctions de recherches ont également été implémentées. Les outils majeurs dans le codage de la SDL sont les fonctions de vérification de clic. Étant donné que le plan est isométrique et les cases de jeu des losanges, il a été impératif de vérifier que les clics sont bien dans le périmètre souhaité. Pour ce faire, l'on calcule les équations des quatre droites de la case puis on vérifie que l'ordonnée du clic est bien comprise dans celle-ci en remplaçant l'abscisse du même clic dans les équations trouvées à l'aide du calcul. Par conséquent, le clic doit être compris entre les droites (AB), (BC), (CD) et (AD) représentées ci-dessous :

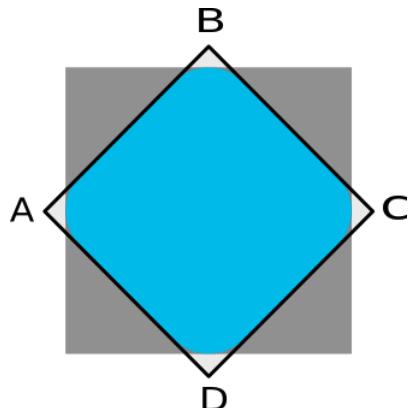


FIGURE 3 – Délimitation d'une la case de jeu

4.5 Méthodes et outils de développement

Compilation séparée : Étant en groupe, nous avons divisé le programme du jeu en différents fichiers pour simplifier notre développement. Cela implique l'utilisation d'outils tels que le makefile favorisant une compilation séparée. Cette méthode est efficace, elle a grandement facilité notre travail et nous a fait gagner du temps. L'on est donc parvenu à avoir plusieurs exécutables. Ces exécutables nous permettent de tester nos différents morceaux de code écrits au cours du projet. Pour ce faire, dans le makefile, nous créons tous les objets de fichiers source « .c » en incluant le fichier d'en-tête principal pour chacun. Lors de l'édition des liens, nous créons des variables contenant les fichiers objets nécessaires à tous les exécutables que l'on souhaite tester. Nous insérons également des macros afin rendre le makefile plus lisible et compréhensible pour tout le groupe de projet.

Outils de débogage : Lors de l'implémentation de la SDL, nous avons rencontré des dysfonctionnements liés à une erreur de segmentation. Nous avons alors fait recours à l'outil de débogage gdb. Nous sommes ainsi parvenus à la détecter. Cette erreur de segmentation était dûe à de nombreuses ouvertures de polices avec la commandes *TTF_OpenFont* dans la fonction *void SDL_ecrire_texte*. Ces polices n'étant jamais refermées mais surtout étant ouvertes des infinités de fois dans la boucle principale, nous assistions à un plantage du programme. À l'aide du gdb, le problème a été réglé en utilisant la fonction *TTF_CloseFont* juste après l'ouverture d'une police (*cf. figure 6*).

Documentation : Dans l'optique de soumettre un projet plus compréhensible, nous avons tenu à générer des pages *doxygen* le décrivant. Sont

regroupés dans celles-ci, tous les prototypes des fonctions, les structures utilisées ainsi que leurs descriptions.

5 Résultats

Toutes les charges du jeu, établies lors de la conception, remplissent, plus ou moins, respectivement les objectifs prévus à cet effet. Ainsi, différents résultats ont été obtenus.

5.1 Version terminal

La version terminal du jeu est fonctionnelle. Nous pouvons effectuer une partie complète avec tous les sorts qui se lancent correctement sans bugs ou erreurs. À la fin de la partie, les personnages, les équipes ainsi que les sorts sont bien tous supprimés (*cf. figure 7*).

Nous pouvons encore améliorer notre jeu en procédant à quelques ajouts et modifications.

- Créer des tableaux de personnages afin d'obtenir un code plus générique et optimal.
- Rajouter une option pour rejouer.
- Retirer des paramètres inutiles des fonctions de sorts notamment. Ceux-ci sont la portée et les dégâts, passés en paramètre, alors que récupérables dans les structures de sorts.
- Implémenter d'autres modes de jeu : deux contre deux, quatre équipes ...
- Rajouter une sauvegarde de la partie. Ainsi nous pourrons charger cette dernière, si nous souhaitons la poursuivre par la suite.

5.2 Intelligence Artificielle

L'IA fonctionne également mais de manière assez particulière. En effet, elle fonctionne pour tous les sorts présentant des portées et des dégâts positifs, entre autres les sorts d'attaque. Cependant, les sorts de portée négative ou nulle et ceux modifiant les coordonnées des personnages du joueur ou de l'équipe adverse ne sont pas utilisés par l'algorithme. Ce dernier a été conçu de telle sorte que tous les sorts effectuent des dégâts sur les adversaires. De surcroît, certaines améliorations peuvent être apportées. L'on pourrait, par exemple, rajouter une évaluation de l'état du personnage afin de savoir si l'on active le sort de soin ou non. Par le même biais, l'on pourrait améliorer l'utilisation des sorts opérés par notre intelligence artificielle. La fonction d'évaluation du meilleur sort a été pensée de façon à ce que les sorts n'aient aucune contrainte d'utilisation : en ligne seulement ou en diagonale.

Il s'agit donc d'intégrer le facteur de portée en plus des moyens d'évaluation du meilleur sort déjà implémentés.

5.3 Réseau

Pour ce qui est du réseau, il existe encore quelques bugs que l'on pourrait corriger notamment côté client. Ces bugs résident au niveau des vérifications, des saisies et des boucles d'attentes demandant un nombre différent de réceptions et d'envois suivant les actions. Toutefois, le réseau fonctionne en version terminal du jeu tel qu'il a été pensé (*cf. figure 9*). La partie se déroule correctement, cependant plusieurs améliorations sont possibles comme la gestion de plus de deux joueurs. Cela permettrait de partager une partie avec plusieurs participants dans une même équipe, mais également d'avoir un nombre différent de personnages par équipe de deux. De même il serait intéressant d'implémenter une version SDL en réseau, ou encore de gérer la portabilité du jeu sous d'autres systèmes d'exploitation. De plus, certaines procédures du code peuvent encore être optimisées.

5.4 Interface graphique

Concernant l'interface graphique, nous nous sommes rapprochés au maximum de l'objectif fixé (*cf. figure 10 et 11*). Nous avons une page d'initialisation sur laquelle se repose le choix de composition d'équipe de l'utilisateur. Celle-ci affiche le pannel des personnages du jeu. Des messages dirigeant l'utilisateur ainsi des identificateurs des personnages déjà choisis sont présentés sur la fenêtre (*cf. figure 12*). Au cours du jeu, un menu proposant des actions est affiché en haut à gauche de la fenêtre, si l'utilisateur clique sur *se déplacer* des cases en bleu correspondant à la portée de déplacement sont placées sur le plan de jeu. Si celui-ci choisit l'option *Utiliser un sort*, le menu de sorts du personnage dont c'est le tour, s'affiche au centre de la fenêtre. Ce menu nous permet d'utiliser le sort que nous souhaitons (*cf. figure 13*). Lors de l'utilisation d'un sort, une portée bleue est affichée, c'est le cas par exemple du sort Ligne du personnage Archer (*cf. figure 14*). Si l'on clique sur la portée et que celle-ci est occupée par un personnage adverse, ce dernier clignote et les dégâts subis sont affichés au dessus de sa tête. La barre de vie est également réduite. En effet, la couleur de celle-ci varie. Ainsi, si les points de vie du personnage sont supérieurs à la moitié des points de vie maximum, ils s'affichent en vert, s'ils sont supérieurs au quart de ces derniers ils deviennent oranges, sinon c'est l'état d'alerte, ils sont présentés en rouge. Une fois la partie terminée, le menu de fin de jeu est affiché avec les deux personnages gagnants (*cf. figure 15*). Des points de l'interface graphique peuvent être améliorés, notamment l'intégration d'animations et de bruitages, la gestion de l'élargissement de la fenêtre et une fermeture de fenêtre impliquant également la fin immédiate du programme. L'on pourrait également rajou-

ter une option *info* dans laquelle seront stipulées les informations du jeu (les développeurs, les règles ...).

6 Conclusion

En conclusion, les fonctionnalités principales que nous nous sommes fixés comme contraintes ont été respectées. Le jeu fonctionne en réseau et sur le terminal. De plus, il est doté d'une interface graphique. Notre objectif de départ était d'intégrer toutes les options proposées par le sujet. Malheureusement par manque de temps le point de l'intelligence artificielle n'a pas pu être inclus dans les différentes versions, réseau et graphique. Le programme a été réalisé dans les temps bien que nous ayons effectué plusieurs changements de la structure du code au cours du développement. De nombreuses améliorations sont opérables. Nous pouvons rajouter différents choix de mode de jeu au programme : personnage contre personnage ou encore quatre joueurs ayant chacun un personnage. Nous pouvons également adapter ces différents mode de jeu avec l'intelligence artificielle. Cette dernière peut également être rajoutée au programme principal et à l'interface graphique. Ce module nous a appris à travailler en groupe sur un projet où l'aspect temporel du développement est différé suivant l'avancée de chacun des membres de l'équipe. Chaque partie du code est réutilisée par tous. Nous avons également appris à utiliser différents outils tels que le git pour gérer les modifications de code même depuis notre domicile ou encore le gdb nécessaire au débogage des branches de notre programme. Nous avons pu approfondir nos connaissances sur la compilation séparée notamment la création de makefile avec des macros. Par ailleurs, nous avons découvert de nouvelles bibliothèques en l'occurrence la librairie SDL et toutes ses composantes, des bibliothèques non abordées dans les modules du cursus, cela nous permet de voir d'un aspect différent la programmation en langage C. Nous avons également développé des capacités de recherches et d'auto-apprentissage.

Annexe

```
L'équipe 1 choisit ses classes :  
[1] : Guerrier  
[2] : Archer  
[3] : Tank  
[4] : Druide  
Choix personnage 1: 1  
Choix personnage 2: 2  
L'équipe 2 choisit ses classes :  
[1] : Guerrier  
[2] : Archer  
[3] : Tank  
[4] : Druide  
Choix personnage 1:  
3  
Choix personnage 2: 4  
---- Choix des cases possibles pour l'équipe 1 ----  
[1]{x=4 y=8}  
[2]{x=2 y=7}  
[3]{x=6 y=7}  
---- [Plan de jeu] ----  


|   |   |   |   |   |   |   |   |   |   |   |   |   |   |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| . | . | . | . | . | . | . | . | . | . | . | . | . | . |
| . | . | . | . | . | . | . | . | . | . | . | . | . | . |
| . | . | . | . | . | . | . | . | . | . | . | . | . | . |
| . | . | . | . | . | . | . | . | . | . | . | . | . | . |
| . | . | . | . | . | . | . | . | . | . | . | . | . | . |
| . | . | . | . | . | . | . | . | . | . | . | . | . | . |
| . | . | . | . | . | . | . | . | . | . | . | . | . | . |
| . | . | . | . | . | . | . | . | . | . | . | . | . | . |
| . | . | . | 2 | . | . | . | . | 3 | . | . | . | . | . |
| . | . | . | . | . | 1 | . | . | . | . | . | . | . | . |
| . | . | . | . | . | . | . | . | . | . | . | . | . | . |
| . | . | . | . | . | . | . | . | . | . | . | . | . | . |

  
Choix pour le perso 1 de l'équipe 1: 1  
Choix pour le perso 2 de l'équipe 1: 3
```

FIGURE 4 – Phase d'initialisation du jeu

```
[Tour numéro:1][Tour du équipe 1][personnage :1]{Caractère : 1}

----- Quelle action souhaitez vous effectuer ? -----
[1]:Se déplacer ?[nombre de déplacement:3]
[2]:Utiliser un sort ? [nombre de points d'actions:2]
[3]:Passer son tour
choix:□
```

FIGURE 5 – Options du jeu

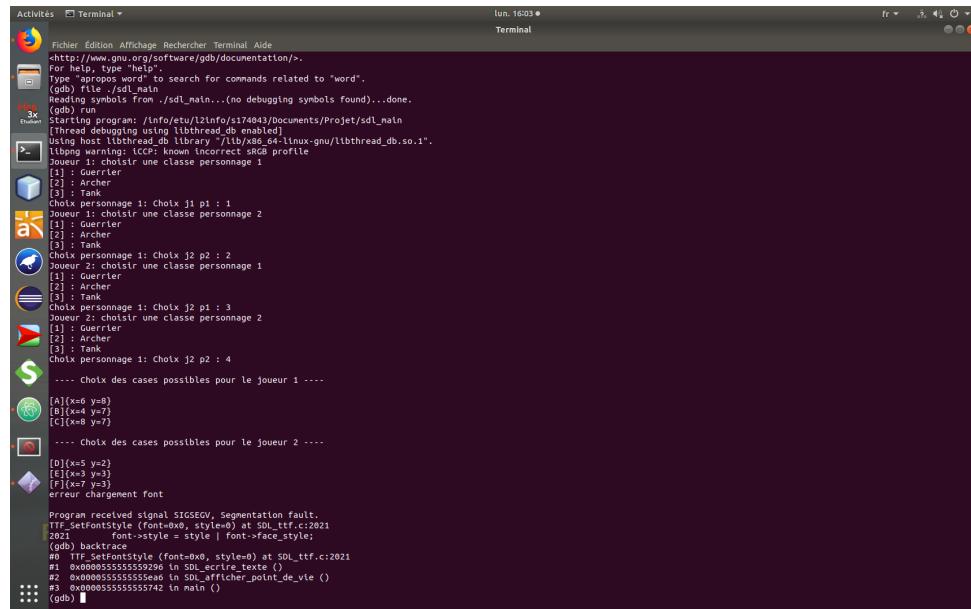


FIGURE 6 – Exemple de débogage

```
Le équipe 1 a perdu
==16059==
==16059== HEAP SUMMARY:
==16059==     in use at exit: 0 bytes in 0 blocks
==16059==   total heap usage: 122 allocs, 122 frees, 16,186 bytes allocated
==16059==
==16059== All heap blocks were freed -- no leaks are possible
==16059==
==16059== For counts of detected and suppressed errors, rerun with: -v
==16059== ERROR SUMMARY: 0 errors from 0 contexts (suppressed: 0 from 0)
```

FIGURE 7 – Test de fuite de mémoire



FIGURE 8 – *Images d'accueil Necrew Arena*

```

Terminal
user@User-System:~/Bureau/Projet_L2/tacticarena$ ./test_reseau
où voulez vous placer votre Tank ?
Choix :
[1] 0 0 0 0 | 0 0 0 0 | 0 0 0 0 | 0 0 0 0 |
[2] 0 0 0 0 | 0 0 0 0 | 0 0 0 0 | 0 0 0 0 |
[3] 0 0 0 0 | 0 0 0 0 | 0 0 0 0 | 0 0 0 0 |
[4] 0 0 0 0 | 0 0 0 0 | 0 0 0 0 | 0 0 0 0 |
[Tour numéro : 1][Tour de l'équipe 1][personnage : Guerrier][Caractère : 1]
..... [Plan de jeu] .....
user@User-System:~/Bureau/Projet_L2/tacticarena$ ./test_reseau
Voulez vous jouer en local[1] ou distant[2] ?
Combien de joueurs 2 ou 4 ?
2
localhost responde to 127.0.0.1IP : 127.0.1.1
En attente de connection
En attente de connection
tous les joueurs sont ready !

```

FIGURE 9 – Jeu en réseau

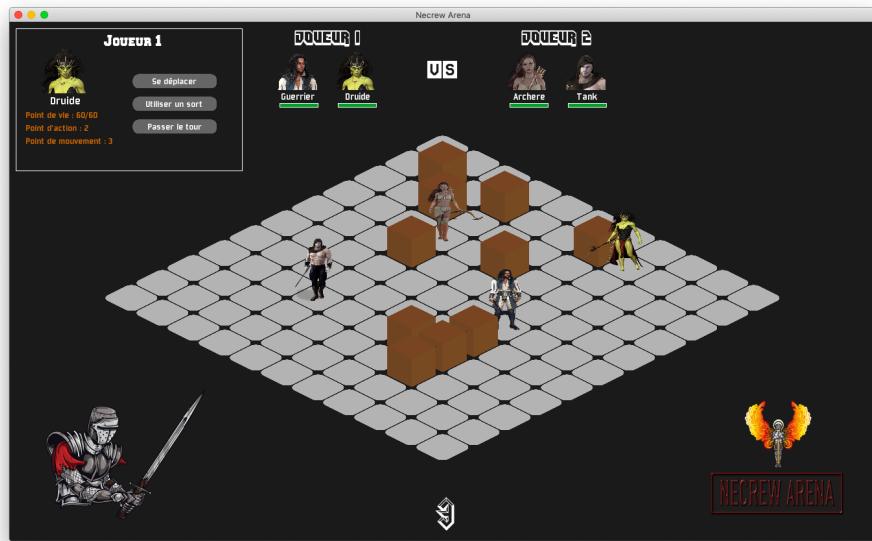


FIGURE 10 – Jeu Necrew Arena



FIGURE 11 – *Jeu Tactics Arena*

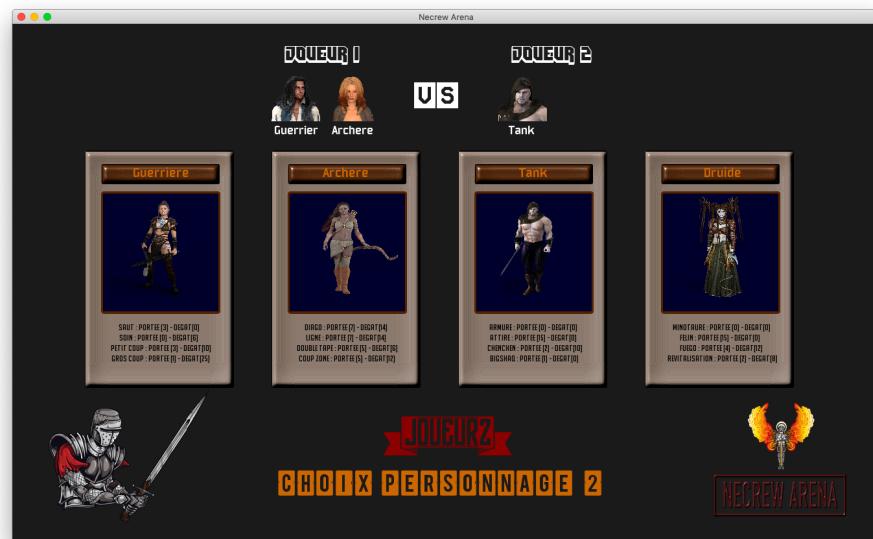


FIGURE 12 – *Choix des personnages*

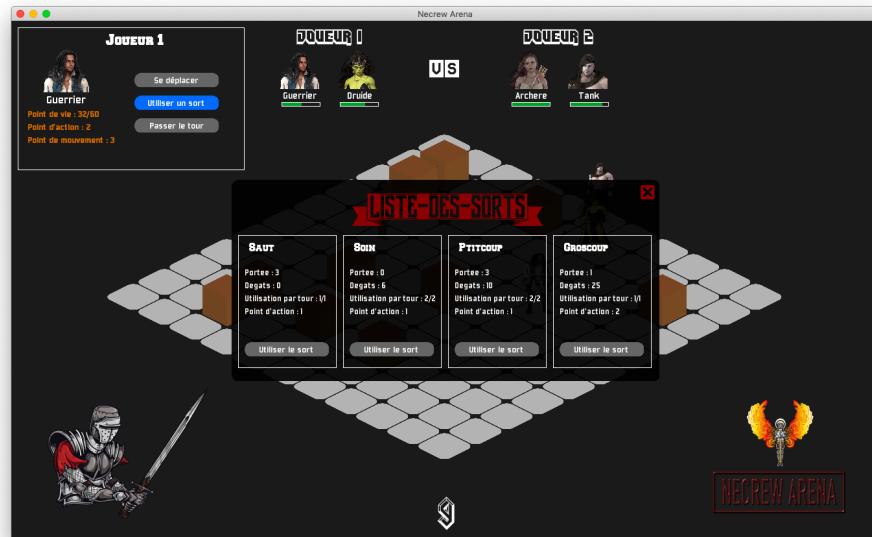


FIGURE 13 – *Menu des sorts*

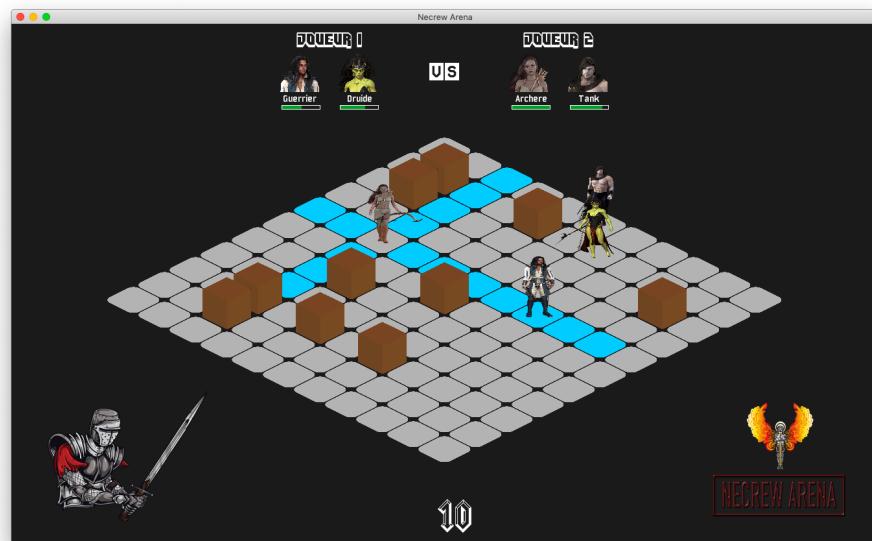


FIGURE 14 – *Sort Ligne de l'Archer*

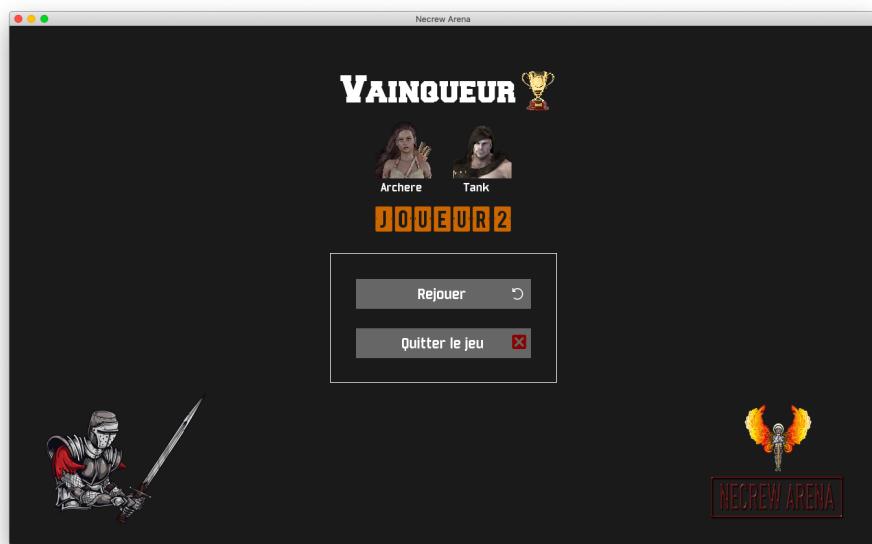


FIGURE 15 – *Menu de fin de jeu*