Université du Maine - Le Mans
Conduite de projet
Licence Informatique - Semestre 4
Année 2019-2020

TACTICS ARENA

BUTEL Robin
CHAUVIN Lucien
DOUCET Thibault
LAFAY Louis

GitHub du projet : https ://github.com/Projet-S4-Info/tactics-arena.git

14-05-2020

Table des matières

1. Introduction

Le sujet de ce projet est la création d'un jeu de stratégie multijoueur en ligne, dans la lignée de « Tactics Arena Online ». Le principe du jeu est un duel tour par tour où chaque joueur contrôle des "pions" ; avec pour objectif d'être le premier à éliminer tous les pions de son adversaire. Le combat se déroule sur une grille de 30 par 30 (rappelant un échiquier de petite taille). Chaque joueur est capable d'obtenir les informations des personnages de son adversaire et donc d'agir en conséquence. Les joueurs intéragissent entre eux en bougeant leurs personnages sur le terrain de jeu ainsi qu'en infligeant des dégâts aux ennemis et en appliquant des malus ou des bonus aux autres personnages. Chaque personnage a à sa disposition un nombre d'actions limité par tour. Les joueurs sont libres de pouvoir utiliser autant de points d'action que nécessaire (dans la limite de ceux qui leur sont accordés) avant de pouvoir passer la main au joueur adverse. Ceci est répété jusqu'à ce qu'un gagnant soit décidé, c'est à dire à la mort de l'ensemble des personnages de l'adversaire, ou en cas d'abandon de celui-ci.

Ce document utilise des hyperliens, n'hésitez pas à cliquer.

2. Organisation

[*Le projet est divisé en plusieurs parties et ce, dans un souci d'organisation. En effet, chaque contributeur a un rôle qui lui est propre et pour lequel il est plus efficace. Un contributeur a donc pour objectif de gérer l'aspect réseau du projet, un autre le moteur d'affichage, un autre le fonctionnement du jeu et pour finir l'aspect design de l'ensemble. Il est donc possible d'établir un cahier des charges par catégorie, permettant de définir les grandes lignes à suivre, mais également les limites.*]

2.1 Répartition des tâches

Il est important de répartir les tâches entre chaque contributeur afin de permettre un travail plus efficace dans le temps, mais également spécialiser chacun dans son propre domaine.

Le projet est découpé en quatre grandes parties : la conception Robin BUTEL, le moteur de jeu et l'interface Thibault DOUCET, la partie réseau Lucien CHAUVIN et le style graphique Louis LAFAY.

Robin BUTEL se charge de "penser le jeu". Il met en place les structures utilisées au cours du jeu, conçoit aussi les personnages, réfléchit au fonctionnement du jeu et de son déroulement. Lucien CHAUVIN s'occupe de la partie communication et réseau ainsi que de l'affichage des menus multi-joueurs et la prise en charge du multithreading. Thibault DOUCET met en place l'affichage du jeu, des menus ainsi que la création de cartes. Il est à l'origine du moteur de jeu et s'occupe également de la gestion des sons joués lors de l'affichage des animations. Louis LAFAY dessine les personnages, modelise les differentes sprites et animations. Il s'occupe aussi de la recherche de chemin pour le mouvement des personnages.

2.2 Gestion des délais

La gestion des délais est souvent la bête noire dans la gestion de projets informatiques. En effet, il est très souvent compliqué de prévoir à l'avance le temps que chacun doit ou peut mettre dans une fonctionnalité ou un debug. Un diagramme de Gantt permettant de visualiser les différentes étapes du développement est néanmoins disponible sur le Git du projet.

2.3 Bugtracking

La définition d'une variable globale ***verbose*** a été rapidement implémentée pour éviter un excès de messages dans la console. La variable globale prend pour valeur un entier correspondant à plusieurs paliers de *debug*. Plus cette valeur est grande, plus il y aura de messages précisant l'activité courante du programme. Deux systèmes de tests ont été également mis en place, le premier étant un moyen de tester le jeu en mode hors ligne, en ne jouant qu'une équipe, l'autre étant alors passive. Ceci a été possible en utilisant un booléen global ***is_online*** qui nous permet d'adapter le code en fonction de l'environement. Le deuxième système de test est un fichier appelé *test.c* contenant une seule fonction *void test()*. Cette fonction a été configurée pour s'activer à chaque fois qu'une zone définie du menu principal reçoit un clic. Un moyen simple d'afficher l'exécution d'un sous-système dans la console pour vérifier qu'il fonctionne correctement.

3. Conception

[*Description des différentes contraintes imaginées avant le développement du projet, présentation du cahier des charges.*]

3.1 Général

L'objectif général de ce projet est le développement d'un jeu dans la lignée de « Tactics Arena Online », jouable à deux joueurs en réseau. Il est donc question de la conception d'un jeu en tour par tour où deux adversaires se combattent armés d'une multitude de "pions" différents (qu'on appellera personnages), avec pour objectif d'être le premier à éliminer tous les personnages de l'adversaire. Un terrain de jeu est mis en place sur lequel les joueurs peuvent bouger leurs personnages, tout en voyant, de manière précise, l'action appliquée par l'adversaire. Des personnages et un système de jeu correspondant doivent alors être conçus.

3.2 Système de jeu (partie 1)

Qu'est-ce qui est indispensable pour un jeu dans la lignée de « Tactics Arena Online » ?. Tout d'abord, les personnages doivent pouvoir se déplacer sur un terrain de jeu, ainsi qu'intéragir avec leurs ennemis et leurs alliés ; soit en leur infligeant des dégâts, soit en appliquant des bonus ou des malus. Pour cela, chaque personnage aura accès à une capacité de mouvement ainsi que quatre autres sorts (compétences).

Le premier de ces quatre sorts (appelé universellement skill 0) est la façon la plus basique, que possède un personnage pour attaquer ses adversaires ; ses dégâts peuvent être de deux types différents : magiques, ou physiques (cela vaut pour tout autre sort). Les deux prochains sorts (universellement appelés skill 1 et skill 2 respectivement) sont des capacités plus complexes, qui sont plus coûteuses (deux points d'action à la place d'un seul (*voir figure 3.2*)) et qui ont des délais de récupération après chaque utilisation. Le dernier sort (appelé skill 3 ou compétence ultime (ult)) est un sort encore plus coûteux (trois points d'action (*voir figure 3.2*)) mais qui, à lui tout seul, peut avoir énormément d'impact sur une partie. Du fait de cet impact, une capacité ultime doit avoir un délai de récupération considérablement plus long que toute autre capacité.

En plus de cela, chaque personnage possède une compétence passive, qui sert à approfondir leur identité ; ainsi qu'à offrir des indices au joueur pour une utilisation optimale de son personnage. Ces compétences passives se déclenchent automatiquement quand certaines conditions sont respectées. Tous les personnages sont dotés de plusieurs caractéristiques (qu'on appelle ***Stats***)(*voir figure 3.2*) qui influent sur les attributs de leurs compétences (portée ou dégâts). Ces caractéristiques ont une valeur minimum de zéro et une valeur maximum de vingt.

3.3 Personnages

Les personnages sont l'élément phare du jeu et leur conception oriente dans de nombreuses directions les décisions prises lors de l'élaboration du projet. C'est pourquoi ils ont été imaginés dès les débuts de la conception. La composition de l'équipe d'un joueur est identique à celle de son adversaire, et est composée d'un exemplaire de chaque personnage disponible. *Plus d'informations sur les personnages sont disponibles en annexe* ***Personnages.pdf****.*

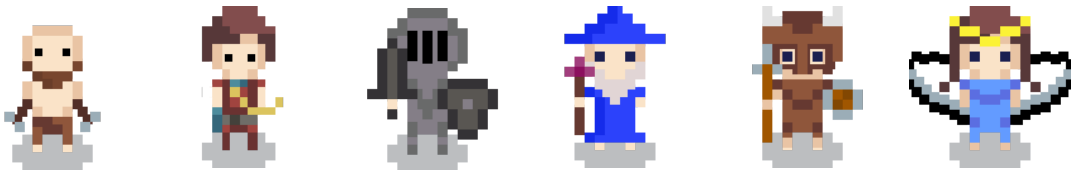


FIGURE 3.1 – Sprites des personnages

3.4 Système de jeu (partie 2)

L'ensemble du gameplay s'articule autour des aptitudes attribuées aux personnages et non l'inverse. C'est pour cela que le système de jeu est conçu pour accommoder le plus possible, de manière générale, les compétences des personnages.

Tout d'abord, pour accommoder les utilités des compétences autres que celles infligeant des dégâts, il nous faut un système de bonus et de malus. Ces bonus et malus, appellés ***Modifiers*** (ou *mods*) peuvent se résumer en deux différentes catégories autres que seulement "bonus" ou "malus". Pour commencer, il y a les *mods* qui augmentent et/ou diminuent les *Stats* des personnages (voir figure 3.2). Ils sont communément appelés ***Stat Change***. L'autre catégorie est plus complexe, puisqu'elle concerne des utilisations plus poussées qu'un simple ***Stat Change***. Elle concerne des *Modifiers* appelés ***Status Effect***.

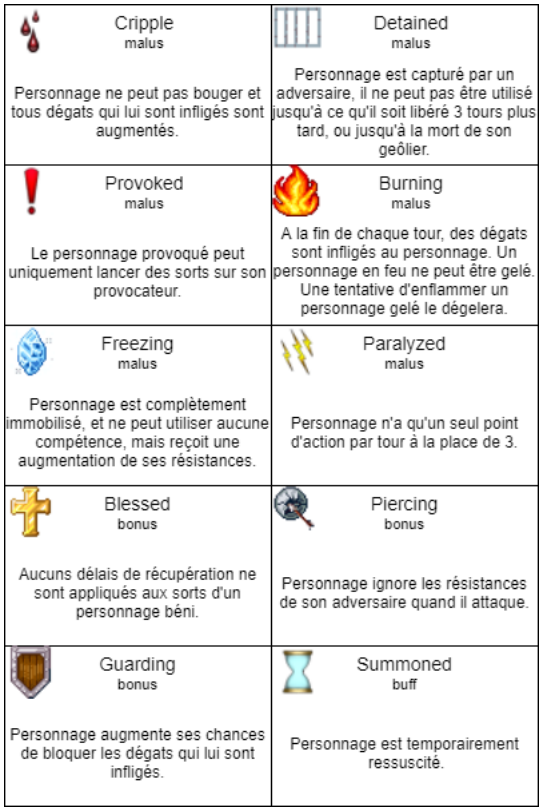
3.5 Interface utilisateur et affichage

Les consignes du projet n'obligent pas le développement d'une interface graphique, en revanche, il est préférable d'en créer une ; non seulement pour une question de facilité d'utilisation, mais également de design et de gameplay. C'est pourquoi l'affichage est considéré comme un aspect essentiel du projet.

3.5.1 Interface utilisateur

L'interface utilisateur (UI) fait référence à l'ensemble des éléments visibles à disposition de l'utilisateur permettant à une application de solliciter des interactions et d'y répondre [1]. Il est donc essentiel d'imaginer une interface claire et fonctionnelle, permettant à l'utilisateur de pouvoir interagir et récolter des informations de manière simple (*voir figure 3.3*). L'interface imaginée utilise les bords de l'écran au maximum afin de ne pas obstruer la visibilité du plateau de jeu central, et certains éléments ne sont affichés que s'il est important de les afficher. Celui-ci doit adopter un style graphique uniforme, utilisant des icônes simples et facilement identifiables permettant alors à l'utilisateur d'obtenir les informations voulues au moment voulu. Pour finir, l'utilisateur a la possibilité d'obtenir une description de l'élément qu'il cherche à consulter au simple passage de sa souris.

Description brève des différentes caractéristiques des personnages



Description brève des différents effets de statut

FIGURE 3.2 – Stats et Status effects

Concernant son contenu, celui-ci doit permettre d'afficher les caractéristiques d'un personnage si sélectionné (voir figure 3.2). Ces informations permettent aux joueurs de consulter l'état d'un personnage ainsi que les possibles bonus et malus qui l'affectent.

Egalement, celui-ci doit afficher l'ensemble des personnages de son équipe ainsi que les points d'action restants de chacun, pouvant alors être un outil pour l'utilisateur permettant de déterminer s'il est possible pour lui d'effectuer davantage d'actions durant son tour de jeu.

L'interface doit naturellement afficher également les possibilités d'actions disponibles pour chaque personnage qu'il contrôle, qu'il s'agisse de le bouger, utiliser une de ses compétences ou encore changer sa direction. Un autre élément important de l'interface est le tableau des "logs" (la liste des derniers évènements qui ont eu lieu) lui permettant de connaître d'éventuels évènements qu'il aurait pu omettre, et celui-ci ne doit pas être surchargé.

Pour finir, l'interface doit comporter un bouton lui permettant de mettre fin à son tour, mais également de consulter s'il a la possibilité de jouer ou non.

3.5.2 Plateau de jeu

Le plateau est l'élément principal du jeu. En effet, c'est sur celui-ci que les personnages seront affichés, et c'est également l'élément qui sera à l'origine de la plupart des interactions des joueurs. Celui-ci est composé de blocs (ou *Tiles*) dont l'ensemble forme le plateau. Celui-ci a été imaginé d'une taille de 30 blocs par 30 blocs,

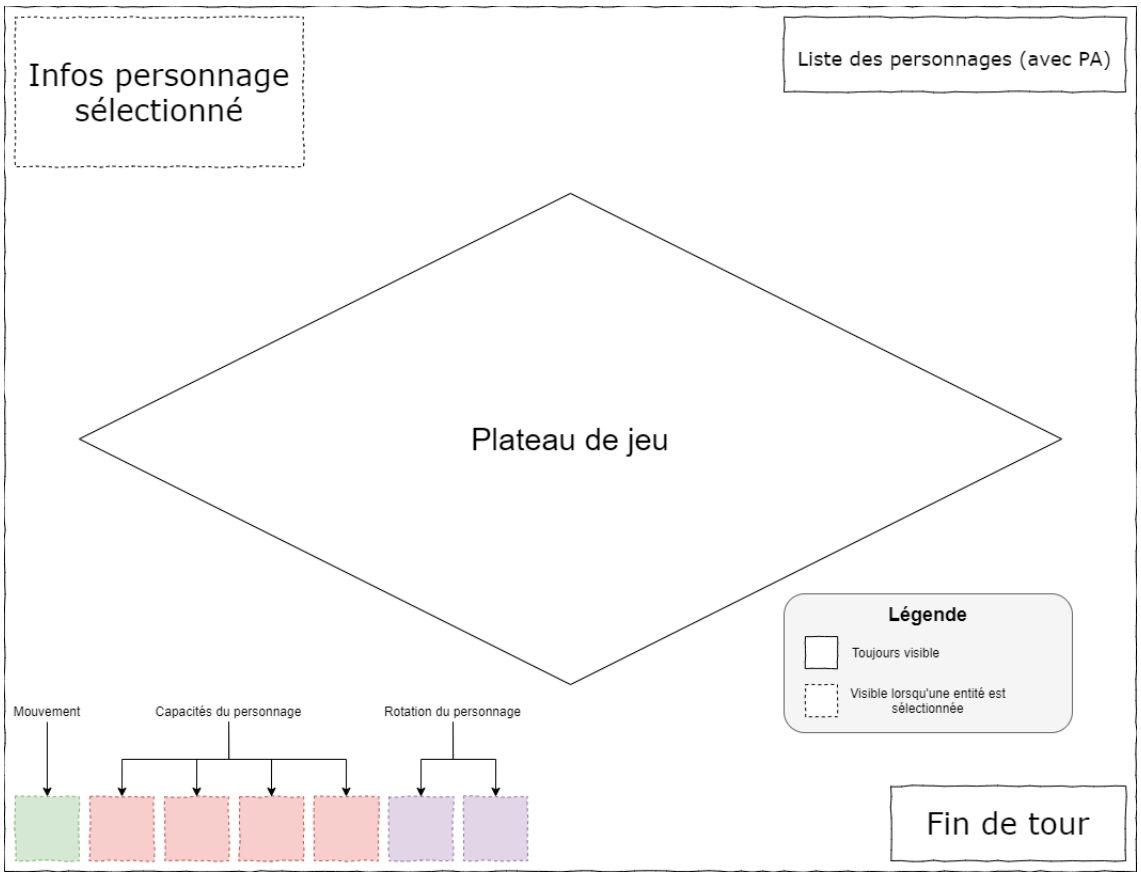


FIGURE 3.3 – Schéma de conception de l'interface utilisateur imaginée pour Tactics Arena

permettant au gameplay de ne pas être trop rapide et archaïque (dans le cas d'un plateau trop petit) mais également de ne pas être trop long et ennuyant (dans le cas d'un plateau trop grand).

3.5.3 Style graphique

Le style graphique joue une part très importante dans un jeu. C'est ce qui traduit visuellement les programmes à travers des personnages, attaques, environnements, déplacements. Pour un jeu tel que Tactics Arena, il est préférable d'avoir des graphismes en pixel art (cela veut dire que les dessins sont pixélisés). Cela donne au jeu un coté simple, abordable et "old school". De plus, la taille des sprites (dessins) est considérablement réduite comparée à des graphismes un peu plus récents et permet donc une meilleure optimisation. Il est aussi grandement plus simple de dessiner en pixel art, ce qui a été un choix décisif.

3.6 Connectivité réseau

Le réseau est une partie essentielle dans un jeu multijoueur. En effet pour pouvoir jouer à plusieurs sur une partie il existe deux possibilités :

- La première est de jouer sur la même machine à tour de rôle. Cette première option a l'avantage d'être simple à mettre en place, en revanche il faut que les deux joueurs soient ensemble physiquement pour

pouvoir jouer au jeu

- La deuxième option est de mettre en place un système de connexion entre minimum deux joueurs afin de transmettre les actions d'un joueur à un autre

L'avantage de cette dernière est que les joueurs n'ont plus l'obligation d'être au même endroit pour jouer. Les joueurs peuvent se connecter entre eux quelque soit l'endroit tant qu'ils ont accès à internet. Pour la réalisation de Tactics Arena le choix s'est porté sur la deuxième option. En effet le jeu nécessite une connexion distante entre deux joueurs. *(cf : voir 4.2.1 plus de détails dans l'annexe réseau shéma 3).*

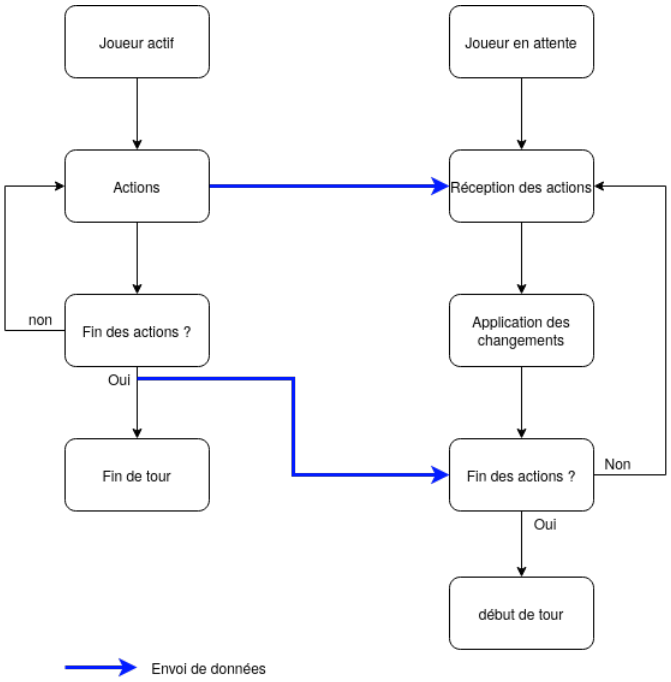


FIGURE 3.4 – Schéma simplifié de la connectivité imaginée pour Tactics Arena

4. Développement

[*Y sont décrits ici les algorithmes et les outils, utilisés au sein du projet ainsi que les optimisations effectuées permettant un fonctionnement plus efficace de l'ensemble.*]

4.1 Structures de données

Les structures de données peuvent être séparées en trois catégories majeures ; les structures pour les données reliées au système de jeu, les structures reliées à l'affichage, et les structures reliées à la communication réseau.

4.1.1 Structures du système de jeu

Veuillez vous référencer à l'annexe "Structure de Donnees.pdf" pour toute structure mentionnée dans cette section. Pour plus de détails encore veuillez consulter la documentation doxygen.

Pour garder le code le plus intuitif possible, tous les *Stats*, les *Status*, les *personnages* (plus techniquement appelés classes) ainsi que leurs compétences sont énumérés. On réfère désormais à ces énumérations comme leurs identifiants (ou ids) respectifs. Toutes ces énumérations et leurs valeurs peuvent être trouvées sur la documentation doxygen.

4.1.1.1 Entity

Au début d'une partie, chaque personnage est défini dans une structure nommée ***Entity***. Il leur est attribué un id, qui vaut l'id de leur classe + 1 et est négatif si c'est un adversaire (par exemple, le Berserker, qui est la première valeur (valeur 0) dans l'énumération ***classId***, aura comme id 1 ou -1 si c'est un adversaire).

La structure *Entity* contient un pointeur sur la ***Class*** du personnage, ainsi qu'un attribut de type ***Coord*** qui indique les coordonnées actives du personnage. Elle comprend aussi un attribut de type énuméré ***lifeId*** qui indique si un personnage est mort ou vivant, ainsi qu'un autre attribut qui agit en tant que compteur de points d'action (*voir figure 3.2*).

La structure *Entity* possède aussi trois tableaux, le premier indique les *Stats* du personnage, on peut parcourir ce tableau en utilisant ***statId*** (l'énumération des *Stats*). Le deuxième tableau indique si le personnage est affecté par des *Status effects*, ce tableau est parcouru en utilisant ***statusId*** (l'énumération des *Status effects*), un zéro indique que le personnage n'est pas affecté par l'effet en question, un 1 indique l'inverse, le seul cas spécial étant pour la provocation, le nombre n'étant pas égal à 1 mais plutôt à l'id du provocateur. Le dernier tableau indique les délais de récupération pour chaque sort, si le délai est à 0, le sort peut être utilisé. Cette structure est parcourue en utilisant le modulo calculé de l'***abilityId*** (l'énumération de toutes les compétences) par le nombre de compétences par personnage.

4.1.1.2 Class

Chaque type de personnage est stocké dans une structure appelée ***Class***. Cette structure contient les *Stats* de base d'un personnage, ainsi que certaines informations comme son nom, l'information écrite de son passif, et son id. Cette structure contient un tableau d'***Ability***, qui possède toutes les informations sur ses compétences.

4.1.1.3 Ability

Toutes les informations concernant les compétences des personnages sont stockées dans une structure appelée ***Ability***. Chaque compétence a un id énuméré par l'enum ***abilityId*** (l'énumération de toutes les compétences) et qui est stocké dans le premier attribut de cette structure. Cette structure contient aussi plusieurs informations diverses comme la portée du sort, son coût en points d'action, son délai de récupération, ainsi que sa description.

Ability comprend aussi un attribut de type énuméré ***targetType*** qui indique sur quelles cases cette compétence peut être utilisée (par exemple uniquement sur un ennemi).

Ability possède également des doubles pointeurs vers des structures qui indiquent les dégâts qu'inflige cette compétence, une liste de coordonnées qui indique la zone d'application de cette compétence, ainsi qu'une liste de *Modifiers* que peut appliquer cette compétence. Le choix du double pointeur a été fait pour éviter les informations redondantes, pour que plusieurs compétences puissent pointer vers les mêmes sous-structures. Utiliser des doubles pointeurs permet d'éviter des problèmes de libération dynamique de mémoire. La liste des coordonnées et la liste des *Modifiers* sont accompagnées par un nombre d'éléments pour pouvoir les parcourir.

Pour finir, un des attributs de cette structure est une méthode. Cette méthode est utilisée pour appliquer des facettes de la compétence que l'algorithme général ne gère pas. Cette méthode vient aussi avec un attribut du type énuméré ***fnid*** qui indique à quel moment dans l'algorithme général la méthode doit être appliquée, ou si, tout simplement, il n'y a pas de méthode.

4.1.1.4 Modifier

La structure ***Modifier*** contient l'information sur l'application d'un *Status Effect* ou d'un *Stat Change*. Cette structure contient un float qui indique la probabilité que la modification soit appliquée ainsi qu'un attribut de type ***targetType*** qui indique sur quel(s) personnage(s) cette modification peut être appliquée. Pour finir, la structure ***Modifier*** contient un attribut de type ***Status*** qui précise de quel *Modifier* il s'agit.

4.1.1.5 Status

La structure ***Status*** stocke l'information d'un *Status Effect* ou d'un *Stat Change*. Son premier attribut indique la valeur du changement de *Stat*. Si cette valeur est égale à zéro, alors la modification est un *Status Effect* et non un *Stat Change*. Le prochain attribut indique quel effet appliquer ou quel *Stat* changer. Enfin, le dernier attribut indique le nombre de tours durant lesquels la modification sera appliquée.

4.1.1.6 Action

La structure ***action*** est la façon dont les actions faites par les joueurs sont stockées et communiquées. Cette structure contient trois attributs : l'id du personnage qui effectue l'action, les coordonnées où l'action est effectuée, ainsi que l'id de la compétence (*abilityId*) effectuée.

4.1.2 Structures pour l'affichage

Veuillez vous référencer à l'annexe "Gestion des textures.pdf" pour toute structure mentionnée dans cette section. Pour plus de détails encore veuillez consulter la documentation doxygen.

L'utilisation de textures préchargées contraint à organiser celles-ci de manière à ce que chacune puisse être retrouvée et affichée simplement. C'est pourquoi elles sont organisées en dictionnaires : chaque texture ou ensemble de texture est chargé avec un identifiant qui lui est associé. En revanche, la structure de chaque dictionnaire est différente selon l'utilisation de chaque texture (textures générales, icônes, textures d'animations et textures de personnages). Ce système aboutit aux trois structures suivantes :

- ***TabTexture***, pour les textures générales
- ***CharTexture***, pour les textures des personnages
- ***AnimTexture***, pour les textures des animations

4.1.2.1 TabTexture

La structure ***TabTexture*** est une texture généraliste utilisée pour le chargement et l'affichage des textures uniques (comme des icônes ou des éléments d'interface basiques). Celles-ci ont un identifiant unique, ainsi que les textures disponibles en deux tailles (si nécessaire) : 64x64 pixels et 128x128 pixels. Cette dernière résolution est utilisée en cas de zoom de l'utilisateur.

4.1.2.2 CharTexture

La structure ***CharTexture*** quant à elle partage beaucoup de similarités avec ***TabTexture***. Conçue pour contenir toutes les textures nécessaires à l'affichage des personnages et de leurs mouvements, elle possède toujours un identifiant ainsi que les deux résolutions de texture. En revanche, celle-ci contient l'ensemble des textures de l'animation du mouvement du personnage, ainsi que les 4 directions (N,E,S,W) dans lesquelles le personnage peut s'orienter. On aboutit alors à une matrice de 8 lignes et 6 colonnes : 4 directions possibles en 2 résolutions différentes, et pour chacune d'entre elles, 6 textures correspondant à celles utilisées lors du mouvement du personnage. En bonus, elle contient une texture du personnage de face, utilisée notamment pour l'affichage de la liste des personnages ou encore de sa "carte d'identité".

4.1.2.3 AnimTexture

AnimTexture est la structure utilisée pour sauvegarder les textures utilisées lors de l'affichage des animations et nécessite davantage d'attributs que celles citées précédemment. En plus de contenir les textures des différentes animations dans les deux résolutions disponibles (sous forme de tableaux de taille maximale constante), celle-ci possède un identifiant directement en lien avec la compétence à laquelle elle correspond. Cet identifiant est du type ***abilityId*** (voir section 4.1.1.3), et permet de retrouver l'animation associée à une compétence simplement. Pour compléter le tout, cette structure possède tous les attributs nécessaires à la manière dont l'animation de la compétence doit être affichée :

- ***nb_steps*** qui correspond au nombre de textures de l'animation
- ***aoe*** qui définit si l'animation doit être affichée sur l'ensemble des cases affectées par une compétence de zone (VRAI si c'est le cas)
- ***on_ground*** qui précise si l'animation doit être jouée au niveau de la tête du personnage ou à ses pieds (VRAI si elle doit être affichée au niveau de ses pieds)
- ***speed*** qui définit le temps (en millisecondes) entre chaque *sprite* de l'animation
- ***sound_effect*** qui permet de définir un effet sonore à l'animation de la compétence (cet attribut utilise une structure ***Mix_Chunk ** ** générée par la bibliothèque **SDL_mixer**, voir partie 4.3.1 pour plus de précisions).

4.1.3 Structures de réseau

Les structures de réseau sont des structures utilisées par les fonctions d'envoi et de réception. Il y en a exactement trois : *ServerStatus_t, Multitile* et *t_user*. Ces structures comportent des informations relatives aux connexions par exemple dans *Multitile*, il y a un tableau d'entiers représentant les identifiants des blocs présents dans une carte sélectionée.

4.2 Réseau

4.2.1 Connectivité

Pour communiquer des informations, le choix se fait entre TCP *(Transmission Controle Protocol)* et UDP *(User Datagramm Protocol)*. Le protocole TCP fournit une transmission fiable avec correction d'erreurs. L'autre protocole, UDP, offre quand à lui une émission de données sans contrôle et sans connexion.

La communication entre les deux joueurs est basée sur le protocole TCP. Il est mis en place à l'aide de sockets. Ce protocole s'est montré pratique par rapport à l'utilisation et l'échange de données souhaités, en effet le protocole TCP permet une correction d'erreurs. De plus, il doit y avoir obligatoirement une connexion active entre les deux joueurs ce qui est favorable dans un jeu où les données sont émises en temps réel.

4.2.2 Serveur/Client

La création du serveur et du client commence par l'initialisation de la structure *SOCKADDR_IN* prédéfinie dans le header de socket *(socket.h)* qui sert de sauvegarde des informations de la connexion. Il faut ensuite indiquer le choix du protocole entre TCP et UDP. A l'aide des fonctions définies dans ce même header, les fonctions permettent d'obtenir un entier qui sert de descripteur de fichier qu'on peut utiliser pour envoyer et recevoir des données. Pour le serveur, il faut ensuite lier le descripteur de fichier avec les informations sauvegardées dans la structure *SOCKADDR_IN*, puis passer en état d'écoute de ce descripteur de fichier afin de savoir s'il y a une tentative de connexion ou non. Si tentative il y a, alors le serveur accepte et établit la connexion entre le serveur et le client. Cette connexion reste active tant qu'elle n'est pas fermée. *(Voir Annexe réseau schéma 1)*.

4.2.3 Transmission de données

Afin de transmettre des données, la création de fonctions d'envoi et de réception est nécessaire. Quand l'un des joueurs veut envoyer des données, après une action par exemple, l'autre joueur est en attente de réception de données.

L'envoi et la réception des données sont faites dans des fonctions génériques. Pour l'émission, la fonction *sendStruct* prend en paramètres un void *, la socket de la connexion ainsi que la taille des données transmises.

Au niveau de la réception, la fonction *recep* prend en paramètres un void * (un pointeur sur la structure désirée), la taille de ce void *, la socket et aussi un pointeur sur une fonction d'affichage en cas de besoin. Dès lors que le joueur réceptionne des données, il envoie au joueur actif une confirmation de réception, puis applique les changements à faire en fonction des données reçues. La fonction générique de réception remplace automatiquement la structure donnée en paramètre par la structure reçue via le réseau. L'application des changements se fait alors après l'appel de cette fonction générique. *(Voir Annexe réseau schéma 2)*.

4.2.4 Cartes multijoueurs

Du fait que le joueur puisse créer des cartes et les nommer comme il le souhaite, il faut éviter les confusions entre le fait de ne pas posséder la carte séléctionnée par l'hébergeur de la partie et le fait de posséder une carte ayant le même nom mais différente dans son contenu. Pour ce faire, il faut envoyer une carte à travers la connexion. Il faut donc avoir une structure nommée ici *Multitile* possédant un tableau d'entiers contenant les identifiants des blocs présents sur la carte. Par la suite il faut envoyer ce tableau via les fonctions génériques vues précédement. A la réception, le client génère, à partir de ce tableau et grâce à la fonction *build_map*, une carte temporaire qui est une copie de la carte sélectionnée par l'hébergeur.

4.3 Outils Graphiques et Sonores

4.3.1 La bibliothèque SDL2

SDL (Simple DirectMedia Layer) est une librairie cross-plateformes conçue pour fournir un accès bas niveau à l'audio, au clavier, à la souris, aux joysticks et au matériel d'affichage via OpenGL et Direct3D. SDL est officiellement compatible avec Windows, Mac OS X, Linux, iOS, et Android [2]. Cette bibliothèque est au coeur de l'affichage de Tactics Arena, et est complétée à l'aide de plusieurs extensions (**SDL_ttf** pour les textes et polices, **SDL_image** pour la gestion des textures depuis une source au format PNG et **SDL_mixer** pour la gestion de l'audio).

4.3.2 Les sprites

Le style de Tactics Arena étant basé sur du pixel art, les personnages et les animations doivent rester dans le thème. Pour créer ces sprites, il faut prendre en compte plusieurs éléments. Premièrement, le plateau de jeu est en 3D isométrique, c'est-à-dire que si les sprites ne sont pas adaptés, les déplacements ne sont pas naturels. Pour ce faire, il faut donc dessiner chaque sprite de personnage pour qu'ils aient uniquement des déplacements en diagonale. Deuxièmement, il faut que les sprites aient une taille adéquate au terrain de jeu. Les normes du 64 pixels par 64 pour le jeu normal, et celle du 128 pixels par 128 pour le jeu agrandi ont été établies dès le début du projet. Troisièmement il faut un outil pour pouvoir créer tous ces sprites, après de nombreuses recherches un logiciel correspond parfaitement à nos critèresa été trouvé ; il s'agit de « Asperite ». C'est un logiciel spécialisé pour le pixel art et permet aisément de gérer les tailles ainsi que tous les détails importants comme le format ou bien les animations. Le format universel pour tous les sprites est le « PNG », pour des soucis de compatibilité avec SDL2. En ce qui concerne les animations, elles sont créées à partir de sprites qui s'enchaînent rapidement (dizaine de sprites). Pour finir, les *status* sont les seules sprites qui échappent à la règle du 64 et 128 pixels. En effet, ils sont utilisés différemment et requièrent une taille légèrement plus petite, du 32 pixels par 32.

4.3.3 Le moteur de jeu

Le moteur de jeu utilisé ici est un moteur très basique utilisant une géométrie en 2D isométrique, c'est-à-dire qu'il simule de la 3D à partir de textures uniquement en 2D et en simulant une impression de profondeur (impression donnée par la forme des textures). Cette technologie est ici préférée à la 3D car celle-ci est plus simple à afficher, mais également plus adaptée pour ce type de jeu. Il n'est d'ailleurs pas totalement correct de parler ici de "moteur de jeu", puisque celui-ci reste un ensemble de fonctions utilisant les bibliothèques SDL2 (qui ont pour but de simplifier l'usage des fonctionnements de base de SDL2).

4.3.3.1 Affichage du plateau de jeu

L'affichage du plateau de jeu consiste en la superposition de textures 2D dans un ordre particulier permettant la simulation de perspective (voir figure 4.1). Cette superposition aboutit en un plateau plat (tous les blocs sont à la même altitude) d'une taille de 30 blocs par 30. Il existe plusieurs types de blocs pouvant être affichés : des blocs d'herbe, d'eau, de glace ou encore de neige. Chaque bloc est représenté par une structure ***Tile*** qui possède tous les attributs nécessaires permettant de le décrire :

- ***tile_id*** qui correspond à l'ID du bloc. Cet identifiant détermine quelle texture adopte le bloc en question
- ***selected*** et ***hover*** qui sont utilisés pour la sélection ou le survol d'un des blocs (voir partie 4.3.3.2 pour plus d'informations)
- ***entity*** qui est un pointeur vers une entité si celle-ci est présente sur le bloc en question (voir partie 4.1.1.1 pour plus d'informations)

- ***trap*** qui est un pointeur sur une structure ***Trap*** qui définit si un piège est présent sur le bloc ou non
- ***walkable*** qui définit s'il est possible pour une entité de marcher sur le bloc en question

Lors de l'affichage de chaque bloc, la présence d'une entité à sa surface est vérifiée. Si c'est le cas, la texture correspondant à l'entité est affichée à sa surface, ce qui permet au moteur d'afficher les personnages ainsi que les blocs dans leur intégralité en un seul tour de boucle.

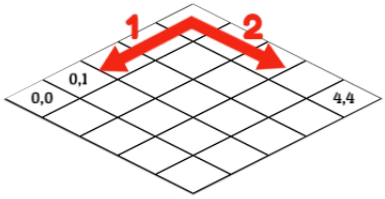


FIGURE 4.1 – Schéma d'affichage du plateau de jeu

4.3.3.2 Sélection et survol d'un bloc

La sélection d'un bloc est un aspect essentiel aux interactions des joueurs avec le plateau de jeu. En effet, c'est cet outil qui permet l'essentiel des interactions des joueurs. Cette sélection est possible en traduisant les coordonnées (en 2D) de la souris dans la fenêtre en coordonnées 2D isométrique. Il est possible d'imaginer ces coordonnées grâce à une grille axonométrique. Pour finir, on génère des vecteurs entre l'origine du bloc et la position de la souris, puis on compare ceux-ci avec les vecteurs qui sont propres à chaque bloc (sur les quatre côtés de la surface du bloc). Le résultat permet de déterminer quel bloc est sélectionné (ou survolé) par la souris (voir figure 4.2). Une fois ces positions calculées, il est alors possible de modifier l'attribut du bloc sélectionné : ***selected*** s'il s'agit d'une sélection de bloc, ***hovered*** s'il s'agit d'un survol.

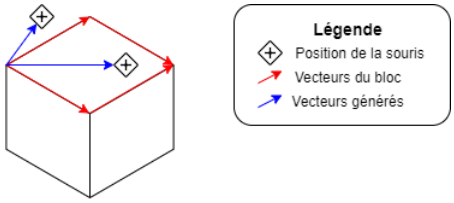


FIGURE 4.2 – Schéma des vecteurs utilisés lors de la sélection d'un bloc

4.3.3.3 Optimisations

Par souci d'efficacité, le moteur présente des optimisations permettant de diminuer le nombre de textures générées et affichées à chaque rafraichissement de l'écran :

- L'ensemble des textures utilisées par le moteur est préchargé au lancement du jeu. Celles-ci sont enregistrées dans des dictionnaires de textures (voir partie 4.1.2 pour plus d'informations, un schéma détaillé du préchargement des textures est disponible en annexe **Geestion des textures.pdf**)
- Les blocs n'apparaissant pas dans la fenêtre de l'écran ne sont pas traités, permettant de réduire autant que possible le nombre de blocs à traiter et à afficher simultanément.

- Un système de cache pour les textes à afficher permet d'enregistrer les textures des textes les plus récurrents, permettant de ne pas avoir à générer de nouvelles textures pour un texte déjà existant, limitant ainsi le nombre d'accès aux fichiers de police.

4.4 Algorithmes et systèmes

4.4.1 Déplacements et Pathfinding

Le but du pathfinding (qui veut littéralement dire : "Trouver chemin") est, pour une application informatique, de déterminer l'itinéraire le plus court entre deux points. C'est une variante plus pratique sur la résolution des labyrinthes par exemple. Dans Tactics Arena, ce processus est utilisé pour quasiment toutes les fonctions de déplacement.

Pour Tactics Arena, il fallait un algorithme de pathfinding à la fois simple, rapide et efficace. Après quelques recherches, un certain algorithme est sorti du lot : Flood fill algorithm. Afin d'expliquer le processus simplement, il faut distinguer deux étapes. La première étape consiste à préparer la recherche de chemin et la deuxième à le chercher. Ces étapes sont découpées en fonctions, ***fill tiles*** et ***pathfinding***. Le processus de ***fill tiles*** est simple, le but est de remplir une matrice d'informations que la fonction pathfinding pourra traiter. Dans cette matrice il y aura d'abord le point de départ (les coordonées données en paramètre) et le point d'arrivée, les autres cases ne seront pas encore initialisées.

Il est maintenant temps d'initialiser les autres cases. La fonction va commencer au point de départ et traiter les quatre cases qui l'entourent (celle du haut, celle du bas, celle de gauche et enfin celle de droite) grâce à une file pour être sûr que tous les éléments sont traités correctement (voir step 1 de Annexe_Pathfinding). Si la case est "*walkable*" ou si rien n'empêche quelqu'un d'aller sur cette case alors on incrémente de 1 la valeur de la case. Ainsi on continue jusqu'au point d'arrivée (voir step 2 de Annexe_Pathfinding).

Une fois que toutes les cases qui séparent le point de départ et le point d'arrivée ont été traitées, la fonction ***fill tiles*** passe le relais à la fonction ***pathfinding*** (voir step 3 de Annexe_Pathfinding).

À ce stade, les trois quarts du processus sont déja accomplis. La fonction ***pathfinding*** part du point d'arrivée et décrémente chaque case jusqu'à trouver le point de départ en stockant dans un tableau donné en paramètre toutes les étapes nécessaires pour y arriver. Nous avons donc en résultat, stockées dans un tableau, toutes les coordonnées qu'il faut parcourir en partant du point de départ jusqu'au point d'arrivée tout en étant le plus rapide possible.

Les fonctions de déplacement n'auront plus qu'à exploiter le tableau reçu de la fonction ***pathfinding*** pour déplacer correctement les entités de coordonnées en coordonnées d'un point de départ au point d'arrivée. Pour ce faire, il y a deux fonctions, une première ***simple_move*** qui se charge seulement de bouger le personnage de coordonnées en coordonnées à partir du tableau et une deuxième,***total_move***, qui est utilisée dans le jeu actuellement. Cette fonction consiste à faire la même chose que ***simple_move*** mais au lieu de seulement bouger le personnage dans sa position initiale, la fonction prend en compte la direction dans laquelle il se déplace ainsi que l'indice de l'animation (sa posture) et adapte sa texture en fonction de celui-ci. A chaque étape du mouvement, une vérification permet de déclencher un éventuel piège qui serait placé sur la trajectoire du personnage, ou d'activer *Sentinel*(voir le passif du Ranger, annexe "*Personnages.pdf*").

4.4.2 Statelist

La ***Statelist*** est le système conçu pour le suivi des *Modifiers* appliqués. Pour le décrire simplement, ***Statelist*** est un système de list à base de pointeurs. Les informations stockées sont la structure ***Status*** de la modification appliquée, ainsi que le pointeur sur l'***Entity*** concernée. Deux listes sont utilisées, une pour les *Modifiers* appliqués pendant le tour local(*stSent*), une pour ceux appliqués pendant le tour adverse(*stReceived*). À la fin de chaque tour, la liste opposée est parcourue et la durée des modifications est désincrémentée.

Plusieurs primitives on été écrites en plus des primitives des listes. La première est la primitive *list_ change* qui diminue ou augmente la durée d'une modification en fonction d'un de ses paramètres. Si la durée passe en dessous de un, alors l'élément est renvoyé pour pouvoir retirer la modification du personnage concerné.

La deuxième est *list_ search* qui prend en paramètre un pointeur sur Entity et/ou un Status effect. Un seul ou les deux paramètres peuvent être rentrés (le pointeur peut être NULL, ou le Status = -1). Cette fonction renvoie le premier élément trouvé qui respecte les paramètres rentrés. Ceci est utile quand un personnage est éliminé et il faut purifier la liste des entrées le concernant, ou alors quand un personnage gelé est dégelé en avance par du feu.

La troisième primitive unique est *list_ check* qui renvoie VRAI si l'élément courrant est un bonus, FAUX si c'est un malus. Ceci est utile pour les compétences *Fury* du Berserker et *Focus* du Ranger (voir l'annexe "*Personnages.pdf*").

4.4.3 Bordures et Zones

Il y a deux types d'actions qui nécessitent des bordures et des zones pour leur portée dans Tactics Arena, les mouvements et les compétences. Etant donné la différence en nature de fonctionnement des deux mécaniques (points de mouvements vs vision/portée), leurs bordures et leurs zones sont calculées de manière différente.

Tout d'abord, les bordures pour les compétences sont calculées en utilisant une formule mathématique qui prend en compte la vision et l'attribut range d'une compétence (voir figure 4.3). Ceci donne la distance

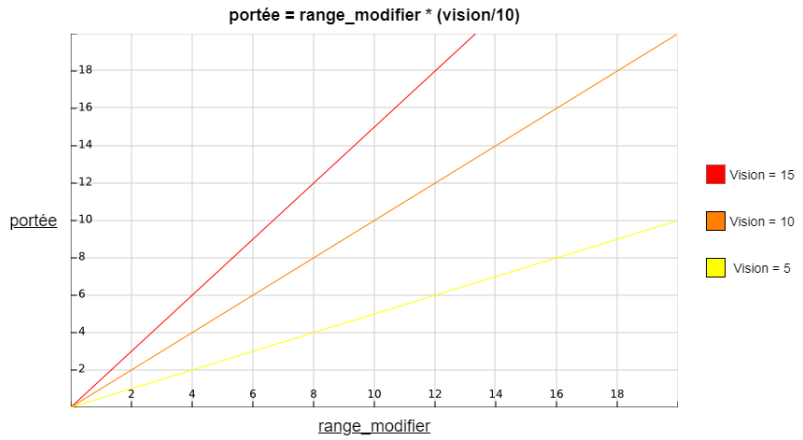
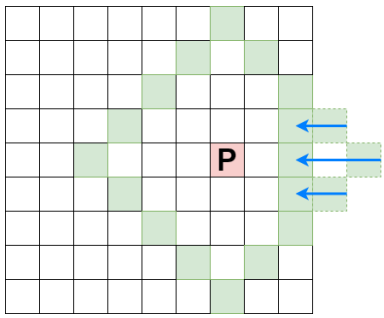


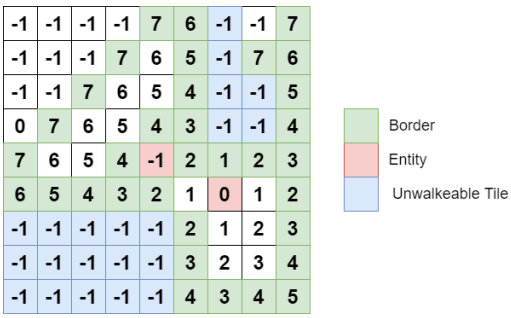
FIGURE 4.3 – La portée d'une compétence en fonction de son attribut range et de la vision du personnage

maximale entre le lanceur du sort et la case ciblée. Une fois les quatre extrémités les plus lointaines de la bordure définies, une boucle est utilisée pour parcourir les cases entre chaque extrémité et les ajoute à la bordure (*voir figure 4.4*). Toute coordonnée qui sort des limites du terrain est normalisée jusqu'à la case du terrain la plus proche. Pour la bordure d'un mouvement, on utilise une matrice remplie via la fonction *fill_ tiles*. Toute case adjacente à une case "unwalkable" est ajoutée à la bordure. (*voir figure 4.4*).

Les zones de compétences (utilisées pour l'affichage) sont calculées en utilisant les bordures générées au-dessus, ainsi que la fonction *isInRange*. Cette fonction vérifie qu'un point donné est à l'intérieur d'une bordure. Pour faire cela, une ligne est tracée d'une extrémité du terrain de jeu jusqu'au point, le nombre de bordures rencontrées sur le passage est compté (*voir figure 4.5)*. Si le point est sur une bordure alors le point est bien dans la bordure, la fonction renvoie VRAI ; si aucune bordure n'a été rencontrée, alors le point ne peut pas être dans la bordure, la fonction renvoie FAUX. Sinon, on continue à tracer la ligne jusqu'à ce qu'on trouve



Calcul de bordure de compétence à 4 de portée



Calcul de bordure de mouvement à 7 de portée

FIGURE 4.4 – Calculs de bordures

une autre bordure. Si on en rencontre une, alors le point est bien à l'interieur de la bordure, VRAI est renvoyé. Si on arrive à la fin du terrain sans en avoir rencontré de nouvelle, le point n'est pas dans la bordure et FAUX est renvoyé.

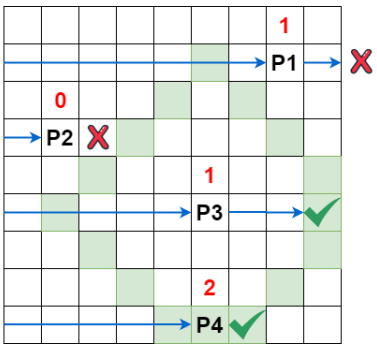


FIGURE 4.5 – Vérification de l'emplacement d'un point par rapport à une bordure

La zone pour le mouvement est calculée beaucoup plus simplement, en utilisant une matrice remplie via la fonction *fill_ tiles* ; on ajoute à la zone toute case dont la valeur est supérieure à zéro.

4.4.4 Vérification des actions

Des vérifications sont faites pour s'assurer que le joueur ne puisse effectuer que des actions correctes.

Tout d'abord, on utilise la fonction *able_ ability* pour verifier si le joueur a assez de points d'actions pour utiliser la compétence, si le délai de récupération de la compétence n'est pas active, ou si le personnage n'est pas empêché par un Status effect comme être gelé par exemple. Si *able_ ability* ne retourne pas une valeur favorable, alors le joueur est empêché de sélectionner la compétence et une icône indiquant la raison est affichée par dessus.

Si la compétence peut être lancée alors une vérification est refaite au moment de la sélection de la cible. La fonction *Cast_ check* est appelée pour faire les dernières vérifications. On vérifie d'abord que si le lanceur est provoqué, son sort cible bien son provocateur.

Ensuite, on utilise la fonction *tile_ type* pour vérifier que la case correspond bien à une case que cette compétence peut cibler (Par exemple, si la compétence choisie ne peut cibler que des ennemis, elle vérifie qu'une ennemi est bien présent sur la case).

Enfin, on vérifie que la case sélectionnée est bien dans la portée du sort. Pour faire cela on utilise deux choses, tout d'abord, la bordure de la portée (ayant été calculée avant afin de l'afficher au joueur), ainsi que la fonction *isInRange* pour vérifier que la cible choisie est bien à l'interieur de la bordure qui délimite la portée du sort.

4.4.5 Application des dégâts

Les dégâts bruts d'une compétence sont calculés en utilisant le *Stat* indiqué dans la structure ***Damage*** d'une compétence (soit Attaque ou Magie). On la multiplie par l'autre attribut de *Damage*, qui sert à varier les dégâts des compétences d'un même personnage. Puis les résistances de la cible sont prises en compte pour calculer les dégâts nets. (*voir figure 4.6*).

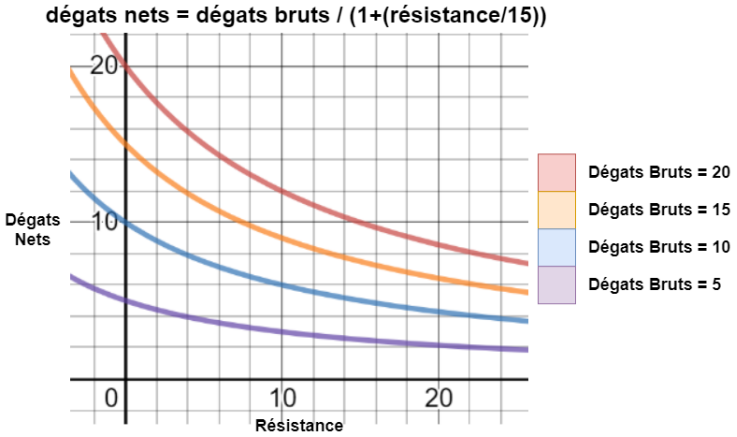


FIGURE 4.6 – Dégâts nets en fonction des dégâts bruts et de la résistance de la cible

4.5 Déroulement d'une partie

Voici le déroulement d'une partie d'un point de vue algorithmique afin de pouvoir mieux expliquer les algorithmes principaux. Une partie est divisée en trois stades majeurs : l'initialisation de la partie, la phase de jeu et la fin de la partie. La phase de jeu est simplifiée en une répétition de tours.

4.5.1 Initialisation de la partie

Une fois la connexion entre les deux joueurs établie et le chargement des textures complété, la partie peut enfin commencer. Tout d'abord, les personnages sont initialisés dans des structures ***Entity*** (*voir section 4.1.1.1*). Puis, l'endroit d'apparition des personnages (appelé spawn point ou tout simplement spawn), est défini. Pour cela, on utilise la fonction *closest_free_tile*, qui nous indique la case libre la plus proche de celle passée en paramètre ; ceci pour éviter aux personnages d'apparaître sur des cases "unwalkable" ; en effet, certaines maps pourraient avoir de l'eau ou du vide dans les coins du terrain de jeu (là où les personnages apparaissent).

Une fois tous les personnages initialisés, l'hôte fait un pile ou face (la fonction *coin_flip*) pour décider quel joueur commence la partie.

4.5.2 Déroulement d'un tour

Chaque tour commence par la fonction *turn_ start* qui initialise certaines variables nécessaires pour le commencement d'un tour. Une vérification se fait pour vérifier que le joueur actif possède bien des personnages avec lesquels il peut jouer, puis la main est passée au joueur.

Une fois l'action choisie par le joueur, et les vérifications faites (*voir section 4.4.4*), les données concernant l'action sont stockées dans une structure de type ***action***. L'action est ensuite envoyée au joueur adverse qui, à la réception, convertit l'id du personnage reçu (le multiplie par -1) pour correspondre à ceux du système local. Dépendamment du fait que l'action soit un mouvement ou une compétence, deux fonctions différentes sont appelées (*voir section 4.4.1 pour les mouvements).*

Une compétence est traitée par la fonction *apply_ action*. Celle-ci appelle la méthode de la compétence ainsi que *apply_ to* (une fonction qui parcourt toute case affectée par une compétence, y repère les personnages, et leur inflige des dégâts ou des *Modifiers*, en fonction du ciblage de la compétence). Ces fonctions sont appelées dans l'ordre dicté par l'attribut de type *fnid* de la compétence. Avant d'appliquer des dégâts ou des *Modifiers* (les deux ont un aspect aléatoire), une communication est faite pour que le joueur en attente ait le même résultat que le joueur actif.

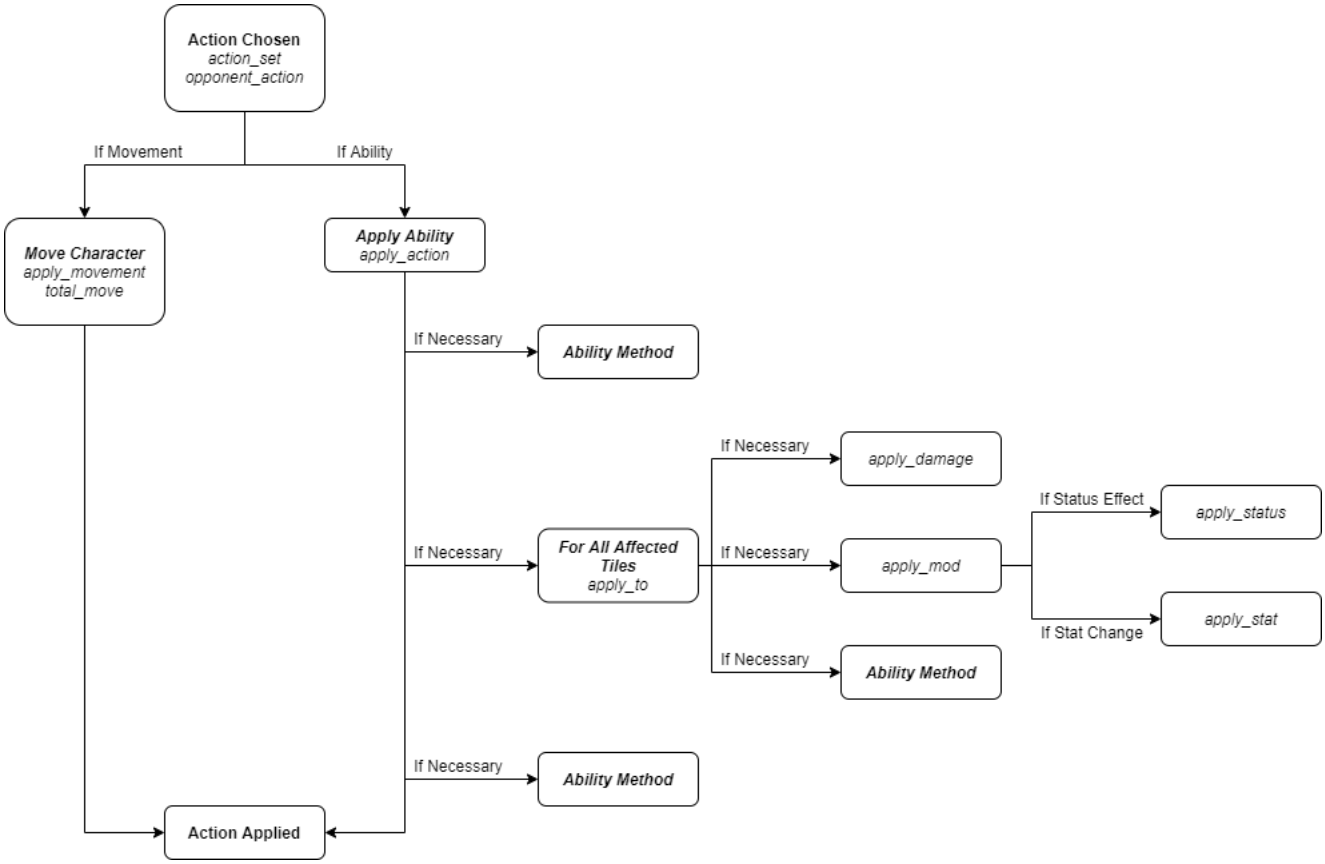


FIGURE 4.7 – Algorithme de gestion d'action

Ceci est répété autant de fois que le joueur a de points d'action sur ses personnages. Entre chaque action faite par le joueur actif, la fonction *play_ check* vérifie qu'il y ait toujours des personnages disponibles. Si non, son tour se finit. À la fin du tour, la fonction *turn_ end* est appelée pour réduire les délais de récupération des

sorts utilisés, réduire la durée des *Modifiers* actifs, et infliger des dégâts aux personnages enflammés.

4.5.3 Fin de la partie

A la fin de chaque tour (ainsi que pendant le tour, pour le joueur actif), une vérification de l'état de la partie est faite via la fonction *game_over*. Celle-ci indique si la partie est toujours en cours, et si non, si le joueur local a perdu ou gagné. Ceci est stocké dans une variable qui sert de condition à la boucle principale du jeu (la fonction *game_loop*). Si la partie est finie, alors la boucle s'arrête, la fenêtre de jeu est fermée, et une fenêtre indiquant les résultats de la partie est affichée.

5. Conclusion

5.1 Fonctionnalités

Tactics Arena correspond dans la majorité à ce que qui était prévu dans la conception. La quasi totalité des fonctionnalités souhaitées est présente et fonctionnelle.

5.2 Améliorations

Afin d'améliorer le jeu, il aurait été utile de concevoir un chat en temps réel ainsi que l'utilisation de plus de méthodes afin de limiter la complexité du programme et le rendre plus modulable. De plus la possibilité de jouer à plus de deux personnes serait une amélioration notable. En effet cela permettrait d'avoir plusieurs modes de jeu tels que du combat en équipe ou bien chacun pour soi. Outre une structure de fichiers plus effective, une meilleure utilisation des branches git-hub aurait été judicieuse afin d'optimiser la gestion des différentes versions de Tactics Arena. De plus, l'affichage des modifications actives sur un personnage de façon constante dans l'interface utilisateur ainsi que l'augmentation du nombre d'effets sonores augmenteraient l'immersion dans le jeu.

5.3 Délais

Concernant le planning prévisionnel, les délais communs n'ont pas été tenus à tous points de vue. En particulier, à la fin du développement, suite à de nombreux problèmes lors de la mise en réseau du jeu. En revanche, les delais prévus relatifs aux aspects individuels du développement des différentes parties telles que le réseau, l'affichage graphique ou bien les abilités des personnages ont été tenus.

5.4 Apports personnels

Le projet a apporté au groupe des méthodes d'optimisation du temps ainsi que de développement d'un projet en groupe ; telles que la documentation du code qui permet une réutilisation simple des fonctions par tous les membres du projet, ou encore le partage des tâches. D'un point de vue organisation, un diagramme de Gantt permet d'avoir une vision globale du temps imparti pour chaque fonctionnalité.

Pour conclure, un projet tel que Tactics Arena profite à tous les membres du groupe de développemnt, tant du point de vue technique que du point de vue organisation. Celui-ci a permis de mettre en évidence l'ensemble des notions étudiées depuis le début de la licence et de faire le lien entre les différents modules ; mettant ainsi en avant l'ensemble des compétences nécessaire à un projet de plus grande taille.

Bibliographie

[1] LeMagIT.fr. Interface Utilisateur (UI), 2016.

[2] LibSDL.org. Simple DirectMedia Layer, 2020.

Table des figures