Déplacement

Dans ce jeu afin de pouvoir déplacer un personnage plusieurs variables sont requises :

1. La connaissance du terrain, la disposition des obstacles.
2. Le nombre de points de préplacement autorisé par personnage.

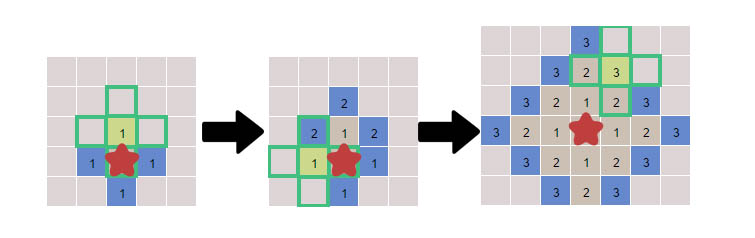
En utilisant les fonctions développées par Modira de gestion de personnage, ces variables sont très accessibles. Une version simple du déplacement serait de demander à l’utilisateur de ne faire qu’un mouvement à la fois en demandant les coordonnées pas à pas, tout en vérifiant si la coordonnée rentrée est juste (obstacle, personnage déjà présent sur la case, coordonnées hors du tableau de jeux). Cette solution est intéressante car rapide à coder mais a un gros désavantage : elle ne pourra jamais être adaptée à d’autre situation. Le but était donc de créer un algorithme de « pathfinding » capable, lui, d’être bien décomposé pour pouvoir réutiliser des fonctions qui peuvent être utiles à la suite du développement.

La fonction de déplacement se décompose donc en trois parties :

1. Une fonction utilisateur qui ne fait pas de calcul si ce n’est s’assurer que les coordonnées rentrées sont correctes.
2. Une fonction qui crée un tableau de distance entre la coordonnée choisie, et toutes les cases accessibles.
3. Une fonction qui depuis le tableau des distances est capable de créer un chemin.

La fonction utilisateur, en plus de vérifier les coordonnées, doit aussi s’assurer que la distance du chemin n’est pas plus grande que la distance maximum de déplacement du personnage actuel.

La fonction qui crée le tableau des distances commence d’un point de départ définie : elle y place 0(l’étoile) et ajoute 1 à chaque cellule adjacente non diagonal (cellules aux bordures vertes), puis elle change de point de départ quand elle ne trouve plus de cellules adjacentes à placer (cellule jaune) afin de placer les prochaines valeurs, tout cela est réaliser jusqu’à ce que la matrice soit remplie ou qu’elle ne trouve plus de point de départ. Lors de la rencontre d’un obstacle la valeur de la distance n’est pas affecter à la cellule. Les points de départ ne peuvent être des obstacles, dans ce cas le chenin ne sera pas direct.



http://www.redblobgames.com/pathfinding/tower-defense/

La troisième fonction crée un chemin en partant de la fin (coordonnées remplie par l’utilisateur) et empile une pile de coordonnées jusqu’à l’arrivée en choisissant le chemin le plus proche grâce au tableau des distances. Cette pile peut être ensuite dépilée pour faire déplacer le personnage, avec une animation, d’un point A à un point B.

Attaque

Avant de pouvoir attaquer il a fallu créer les différentes attaques des personnages (fonctionnalité que nous avons jugée intéressante). Pour cela il a fallu rajouter dans la définition d’un personnage un nouvel élément : un pointeur. Ce pointeur pointe vers le début de la pile des attaques d’un personnage, de ce fait il est très simple d’ajouter une attaque à un personnage, l’utilisation des piles permet une grande flexibilité. Les différentes caractéristiques d’une attaque sont :

1. Son Nom *(obligatoire)*
2. La distance maximum d’attaque. *(obligatoire)*
3. La distance minimum d’attaque. *(obligatoire)*
4. Son cout en points d’action. *(obligatoire)*)
5. La zone ou sont appliqués les dommages. *(obligatoire)*
6. Si l’attaque n’est utilisable seulement en face du personnage qui la lance. *(obligatoire)*
7. Des effets qui altèrent :
   1. Les points de vie maximum d’une cible.
   2. Les points de vie d’une cible.
   3. Les points d’action d’une cible.
   4. Les points de déplacement d’une cible.
   5. Les coordonnées d’une cible.

Pour rendre la vie plus facile à la création d’une attaque, la fonction prend en paramètre les caractéristiques obligatoires, et un nombre de paramètres indéfini d’effets. L’avantage est un code plus propre ou l’on voie apparaitre seulement les effets qui nous intéressent. Exemple pour l’archer : *"HP", -1,"coord\_r", -2,"coord\_c", -2* indique que l’attaque fait 1 de dommage, et fait reculer le personnage soit de 2 sur l’axe des abscisses soit de 2 sur l’axe des ordonnées.

Du fait du stockage des attaques dans un pointeur qui pointe vers le début d’une pile, il a fallu écrire des primitives qui nous étaient essentielles. Une fonction d’affichage de la pile des attaques d’un personnage, une fonction qui crée le contenant de la pile elle-même appelée seulement lors de la création d’une attaque, une autre fonction nécessaire était la récupération d’une attaque en fonction de son emplacement dans la pile, afin que l’utilisateur choisisse une attaque en fonction de son affichage. Quand le programme est quitté ces piles sont correctement nettoyées.

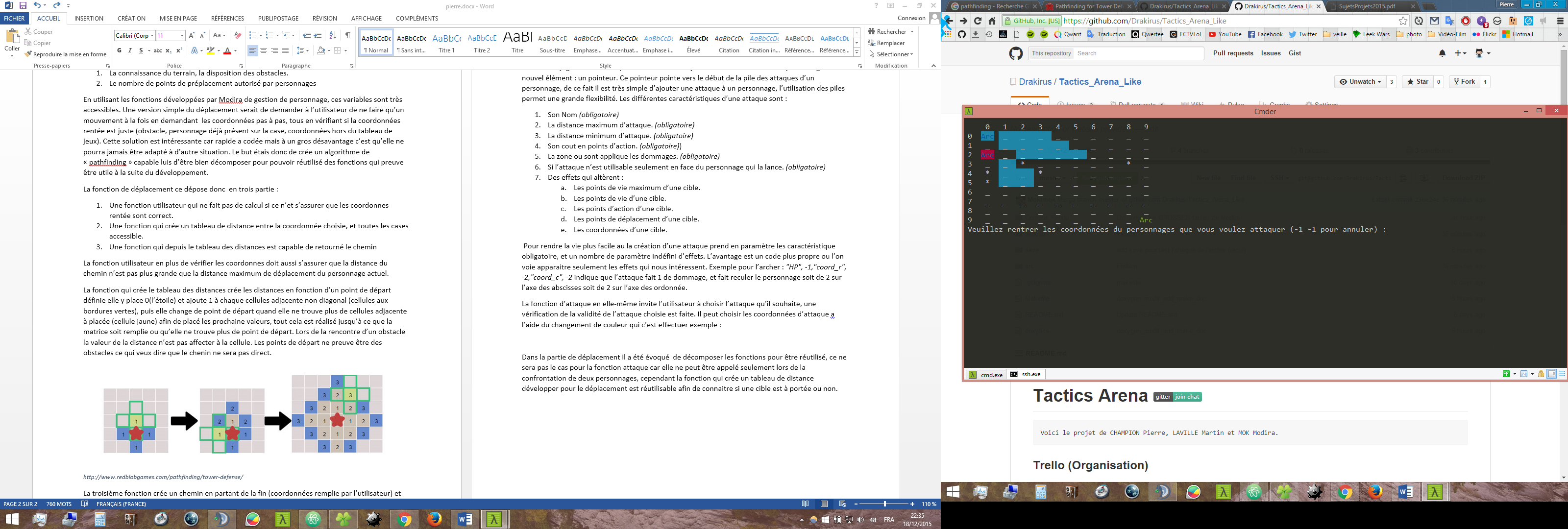
La fonction d’attaque en elle-même invite l’utilisateur à choisir l’attaque qu’il souhaite dans la liste, une vérification de la validité de l’attaque choisie est faite. Il peut ensuite choisir les coordonnées du personnage à attaquer à l’aide du changement de couleur qui s’est effectuer : exemple figure 1 pour une attaque qui a pour distance min 2 et distance maximum 5.

Figure 1 attaque archer

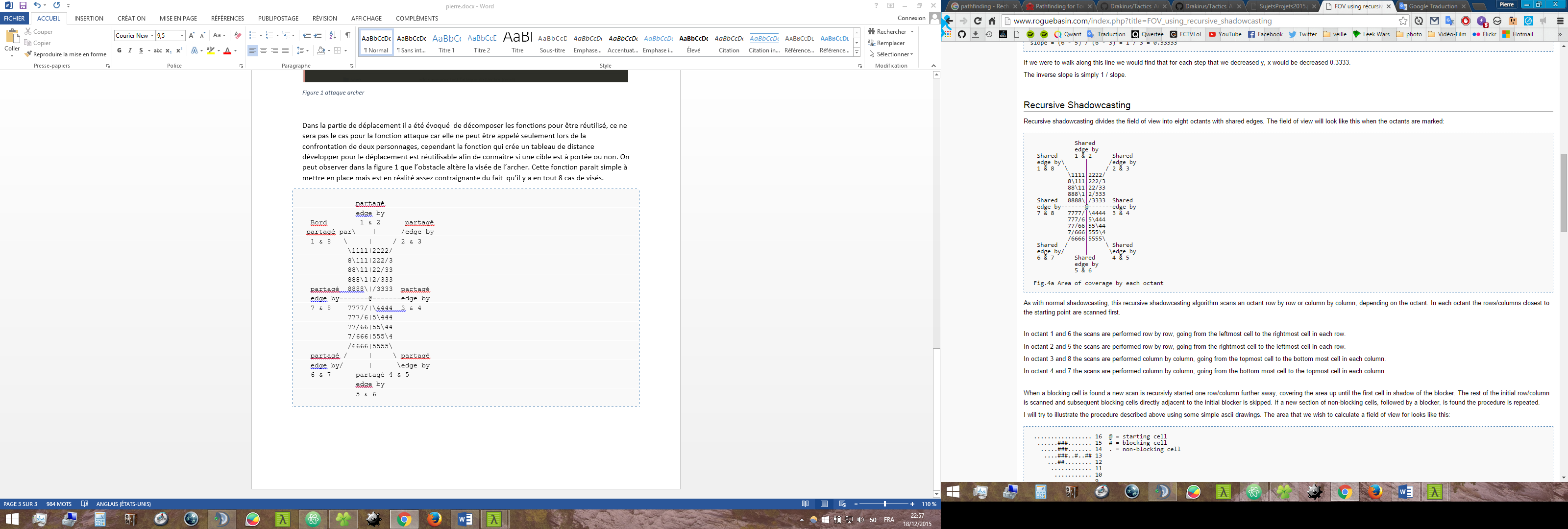
Dans la partie de déplacement il a été évoqué de décomposer les fonctions pour être réutilisé, ce ne sera pas le cas pour la fonction attaque car elle ne peut être appelée seulement lors de la confrontation de deux personnages, cependant la fonction qui crée un tableau de distance développé pour le déplacement est réutilisable afin de connaitre si une cible est à portée ou non. On peut observer dans la figure 1 que l’obstacle altère la visée de l’archer. Cette fonction parait simple à mettre en place mais est en réalité assez contraignante du fait qu’il y a en tout 8 cas de visés. 

Figure 2, champ de vision

Sur la figure 2 (<http://www.roguebasin.com/index.php?title=FOV_using_recursive_shadowcasting>).