- Améliorations / Fonctionnalités ajoutées au projet

Nous avons tout d'abord choisi de reprendre toute l'ébauche de solution du jeu qui nous a été fourni afin de partir d'une base entièrement fonctionnelle et qui répondait à nos besoins. Nous avons donc pris le même jeu réalisé en TP par nos soins. Cependant, cette base étant réalisée sur un unique fichier .cpp, nous avons dû structurer les différentes fonctions essentielles au bon fonctionnement du jeu. Ainsi, nous avons créé plusieurs fichiers sources (.cpp) associés à un fichier d'en-tête (.h) respectif.

Une fois cette première étape réalisée, nous avons commencé à lister différents ajouts de fonctionnalités et d'améliorations. Avant toute chose, nous nous sommes dit que chaque nouvelle fonctionnalité présente dans le jeu doit pouvoir se positionner de façon aléatoire sur la grille. De plus, chacune de ces fonctionnalités est codée dans un fichier source (et donc un autre fichier d'en-tête) à part afin de structurer au mieux notre jeu.

La première fonctionnalité que nous avons rajoutée est le **trou de ver** (plus simplement, c'est un tunnel où l'on peut voyager d'un point A à un point B et inversement). Nous avons tout d'abord créé une fonction permettant de générer l'emplacement des trous de ver de façon aléatoire à l'aide de la fonction rand() du C++. Dans cette fonction, nous avons dû faire attention au fait que la position des trous de vers générée ne corresponde pas à l'emplacement d'un des joueurs ainsi qu'au fait que les deux trous de ver ne se superposent pas (tout cela avec une boucle while). Puis, nous avons créé une seconde fonction permettant le changement de position du joueur en cas de déplacement sur un trou de vers. Ces trous de vers sont représentés par des ronds verts.

Nous nous sommes ensuite penchés sur l'ajout de **murs** dans le jeu. Un mur est composé de cases sur lesquelles les joueurs ne peuvent pas se déplacer. Ils ont une taille aléatoire définie à chaque partie qui varie de 1 à 7 cases se suivant horizontalement. Pour former un mur, nous avons défini deux fonctions. La première est celle qui permet de définir l'emplacement du début et de la fin dur mur en donnant une distance de cases horizontale aléatoire comprise dans l'intervalle [0 ; 6]. Par la suite, dans la fonction initialisant la matrice du jeu, on a indiqué que les cases comprises entre le début et la fin du mur sont des cases mur représentées par le token mur (« X »). Ces cases seront interprétées par la fonction affichant la matrice de jeu en dessinant des croix rouges sur fond noir. La deuxième fonction est celle qui rend le mur infranchissable. On y indique que si un joueur effectue un déplacement en direction d'une case mur, il fait automatiquement un déplacement dans le sens inverse de son arrivée pour le faire rester sur sa case.
En ce qui concerne la génération aléatoire des murs, nous avons eu pas mal de problèmes de débordements de la fin du mur en dehors de la matrice ce qui rendait le jeu quasiment inutilisable. Nous nous sommes rendus compte que cela était dû à une mauvaise utilisation de la fonction rand() et il a donc fallu rajouter une condition : tant que la position en colonne de la fin du mur est inférieure à celle du début du mur, on régénère la position de la fin du mur.

Nous nous sommes ensuite confrontés à une amélioration du jeu cette fois-ci : le **réglage de superposition du joueur et du trou de ver**. En effet, lorsqu'un joueur passait sur un trou de ver, il disparaissait de la grille car il était au même emplacement que le trou de ver d'arrivée. Nous avons donc réglé ce problème dans la fonction d'initialisation de la matrice en mettant une condition qui est que ce si un joueur se trouve au même emplacement qu'un trou de ver alors c'est le joueur qui devra être affiché et non le trou de ver.

Nous avons ajouté la fonctionnalité de **boost**. Soit n, un entier naturel généré aléatoirement et compris dans l'intervalle [1 ; 3]. Une case boost, lorsqu'un joueur se déplace sur celle-ci, le propulse jusqu'à la (n)ième case dans la direction de son arrivée. Pour créer une case boost, nous utilisons deux fonctions. La première va générer son emplacement aléatoirement ainsi que son degré d'intensité n. Un joueur se propulsant sur un mur ne passera jamais à travers ni le surmontera, il rebondira dessus et atterrira sur la dernière case libre avant le mur. Si un joueur se propulse dans l'entrée du trou de ver, il passera dedans et ainsi se retrouvera de l'autre côté.

Nous avons également implémenté une fonctionnalité du même style que la précédente qui est le **boost orienté**. Pour cela, tout comme pour le boost réalisé précédemment, nous avons créé une fonction de génération du boost orienté pour ce qui concerne son emplacement et son nombre de cases à sauter mais aussi pour la direction imposée par le boost. La fonction génère donc un nombre compris entre 4 et 1 (haut, bas, droite, gauche). Nous avons ensuite, créé une fonction permettant l'exécution de ce boost en fonction de la direction et de l'indice générés en amont. Tout comme pour le boost, nous avons mis les fonctions de mur et de trou de ver afin que le joueur ne puisse se déplacer sans éviter ces obstacles comme c'était le cas au début. Cette case est représentée par une flèche indiquant sa direction (^, v, <, >) ainsi que le nombre de cases qu'elle va sauter.

Nous nous sommes ensuite penchés sur une amélioration à faire : le **réglage de superposition du joueur 2 sur le joueur 1** quand ce dernier gagne la partie. Pour ce faire, nous avons mis en place un booléen dans le programme principal (« Notreppal() ») qui passe à vrai uniquement si le joueur 1 gagne. Puis, dans l'initialisation de la matrice, nous avons mis une condition indiquant que le joueur 1 devient prioritaire sur le joueur 2 si leurs positions sont les mêmes et si le booléen est à vrai. Le problème ne s'est pas posé en cas de victoire du joueur 2 car ce dernier est affiché après le joueur 1 dans le code.

Nous avons ajouté un **menu permettant le choix d'un nombre de tours prédéfini** par nous-même. Nous avons choisi de proposer 3 nombres : 10, 20 et 30 tours. Nous avons également apporté une quatrième option nommée « Surprise » qui génère aléatoirement soit 10, soit 20 ou soit 30 tours.

Nous avons ensuite codé une fonctionnalité nommée **case double tour**. Si un joueur passe sur cette dernière alors il a la possibilité de rejouer immédiatement et ainsi avoir un tour d'avance

sur son adversaire. Tout comme pour les fonctionnalités précédentes, nous avons fait en sorte que lors de la génération de la position de la case, celle-ci ne se superpose pas avec les autres éléments de la grille. Ainsi si un joueur passe sur cette case, le booléen indiquant si c'est au joueur 1 ou 2 de jouer ne change pas afin de permettre un second tour au joueur.

Nous nous sommes occupés ensuite d'une amélioration concernant uniquement le code et non pas l'aspect visuel et interactif. Nous avons rendu nos différentes fonctions, créées précédemment, **applicables pour n'importe quel joueur**. En effet, auparavant, nos fonctions étaient divisées en deux parties : l'une pour le joueur 1 et l'autre pour le joueur 2. Cependant, nous nous sommes rendus compte que cela multipliait par deux la taille des fonctions et donc le code n'était pas optimisé. Nous avons ainsi réglé cela en rendant les fonctions utilisables pour tous les joueurs.

Nous avons également éliminé un **problème de débordement des joueurs de la grille** provoquant l'arrêt du jeu. Ainsi, nous avons mis des conditions dans le code de déplacement des joueurs qui, en fonction de leur position sur la grille, doivent rejouer si jamais ils vont dans une direction impossible (contre une paroi).

Nous avons ajouté un **mode de jeu en 3 manches**. Pour pouvoir ajouter une gestion des modes (choix entre mode classique soit une seule partie ou un mode battle en 3 manches), on a dû revoir la structure de la fonction principale. La boucle qui tournait tant qu'il n'y avait pas de victoire a été inscrite dans une boucle qui tourne pour le nombre de fois qu'on lui a indiqué. Ainsi, le choix du mode classique la fait tourner pour 1 tour ; le choix du mode battle, pour 3 tours. Après le choix du mode, l'utilisateur est redirigé vers le menu du choix du nombre de tours qui sera appliqué à chaque manche. Le mode battle nous a donc forcé à revoir la manière dont était caractérisée une victoire. Ainsi, un tableau de score est affiché pour chaque manche. Lorsqu'un joueur gagne une manche, son score est incrémenté de 1, dans le cas où nous faisons face à une égalité, personne ne prend de points mais le nombre de manche est incrémenté de 1 passant de 3 à 4 manches et ainsi de suite, ce qui nous assure d'avoir un gagnant si la situation se débloque. Un joueur gagne si son score est supérieur à celui de son adversaire et s'il est aussi supérieur ou égal à 2. Ainsi une battle prévue pour 3 manches peut se terminer en deux manches avec des scores de 2-0 ou 0-2.

Nous avons mis en place un **menu principal** permettant de choisir le mode auquel nous souhaitons accéder. Par le simple usage de numéros (1, 2 ou 3), nous pouvons choisir parmi nos trois modes respectifs :
1 → Mode classique
2 → Mode 3 manches
3 → Didacticiel
Cela a été réalisé à l'aide d'un switch case. Par ailleurs, lorsque nous choisissons un mode, le nom du terminal s'adapte en fonction de ce dernier. Bien entendu, si nous tapons un autre numéro que 1, 2 ou 3 ou même un autre caractère aucun mode ne va démarrer et le menu va simplement se réafficher.

Nous avons ajouté une amélioration dans le jeu : un **fichier de configuration**. Ce dernier permet de personnaliser divers paramètres du jeu tels que les touches de déplacement ou encore l'apparence des joueurs. Nous avons tout d'abord créé une fonction vérifiant que le fichier est bien présent et accessible renvoyant un message d'erreur le cas échéant. Ensuite, à l'aide d'une map, nous avons stocké les différentes valeurs des paramètres à la clé correspondante tout en prenant soin d'ignorer les lignes de commentaires (commençant par un « # »). Puis, nous avons remplacé les variables mises en dur dans le code par un appel de la fonction renvoyant la map ainsi que la clé du paramètre correspondant. Nous avons créé deux types de map : une <string, char> et une <string, unsigned>. Cette dernière est utilisée pour le paramétrage du nombre de manches et que des nombres de tours prédéfinis. La première map, quant à elle, permet de gérer le paramétrage des touches de déplacement et de l'apparence des joueurs. Pour être sûr que les map contenaient bien les informations voulues et bien associées entre elles, nous avons créé une fonction ShowMap() (code présent en annexe) générique qui permet d'afficher l'intégralité de la map et ainsi d'en vérifier son contenu.

Enfin, nous avons rajouté un nouveau mode étant : le **didacticiel**. Ce dernier se présentant sous un document écrit visant à former à l'utilisation du jeu. Par l'explication détaillée de commandes ainsi que de l'environnement, nous souhaitons rendre le jeu plus compréhensible. Cela permet alors à une personne novice, ne connaissant pas le jeu de pouvoir s'immerger rapidement dans ce monde virtuel sans être déboussolé et dépassé, le tout en détaillant les fonctionnalités du jeu.

- Fonctions de test

Fonction permettant d'afficher le contenue d'une map et ainsi en vérifier l'exactitude des informations

```
template <class U, class T>
void ShowMap (const map <U, T> & MyMap)
{
	for (const pair <U, T> & Elem : MyMap)
	{
		cout << "Clé : " << Elem.first << endl
			<< "Valeur : " << Elem.second << endl << endl;
	}
} // ShowMap
```