CAILLET Claudia FRANÇOIS Marguerite THERMEAU Frank

# - RUZZLE -

# Tables des matières (pages)


**Spécifications externes** **3**
- Fonctionnalités 3
- Choix techniques 3
- Choix graphiques 3
- Conventions de codage 4

**Rôle et fonctionnement des principales entités** **6**
- Boucle de jeu 6
- Gestion de l'affichage 6
- Gestion de la grille 7
- Gestion des mots et des lettres 7
- Gestion du dictionnaire 8
- Gestion du timer 9
- Autres fichiers 9

**Organisation & difficultés rencontrées** **11**
- Organisation 11
- Difficultés rencontrées 12

**Conclusion** **13**


# Tables des matières (liens)


Spécifications externes
- Fonctionnalités
- Choix techniques
- Choix graphiques
- Conventions de codage

Rôle et fonctionnement des principales entités
- Boucle de jeu
- Gestion de l'affichage
- Gestion de la grille
- Gestion des mots et des lettres
- Gestion du dictionnaire
- Gestion du timer
- Autres fichiers

Organisation & difficultés rencontrées
- Organisation
- Difficultés rencontrées

Conclusion

# I. Spécifications externes

## A. Fonctionnalités

Les fonctionnalités que nous avons réalisées sont celles demandées dans les contraintes du projet et présentes dans le jeu original Ruzzle. Nous ne les avons pas toutes implémenté par manque de temps cependant. Voici le déroulement d'une partie dans notre version du jeu :

- Le joueur lance le jeu et clique sur le bouton "Play", ce qui affiche l'écran de jeu et lance le timer.
- Chaque clic gauche sur une lettre, si elle est autorisée (c'est-à-dire si elle est adjacente à la dernière lettre sélectionnée et qu'elle n'est pas déjà sélectionnée), l'ajoute au mot courant.
- Chaque clic droit termine le mot, ce qui l'ajoute au score s'il était valide (c'est-à-dire compris dans le dictionnaire).
- Le joueur peut quitter la partie à tout moment.
- Le joueur peut terminer la partie à tout moment ; sinon elle se terminera automatiquement à la fin du timer.
- A la fin de la partie, le joueur consulte son score final et peut rejouer ou quitter.

## B. Choix techniques

Le langage imposé était le C, que nous avions tous plus ou moins pratiqué durant nos différents cursus (Licence Informatique et DUT Informatique).

Etant tous sous Windows, nous avons choisi comme EDI Code::Blocks, n'en connaissant pas beaucoup d'autres à l'époque. Nous avons appris par la suite qu'il existait CLion, mais nous avions déjà configuré la librairie SDL1 sous cet environnement, il était un peu tard pour changer. De plus, un des membres du groupe avait déjà un peu d'expérience de SDL2 sous Code::Blocks, ce qui a influencé notre choix.

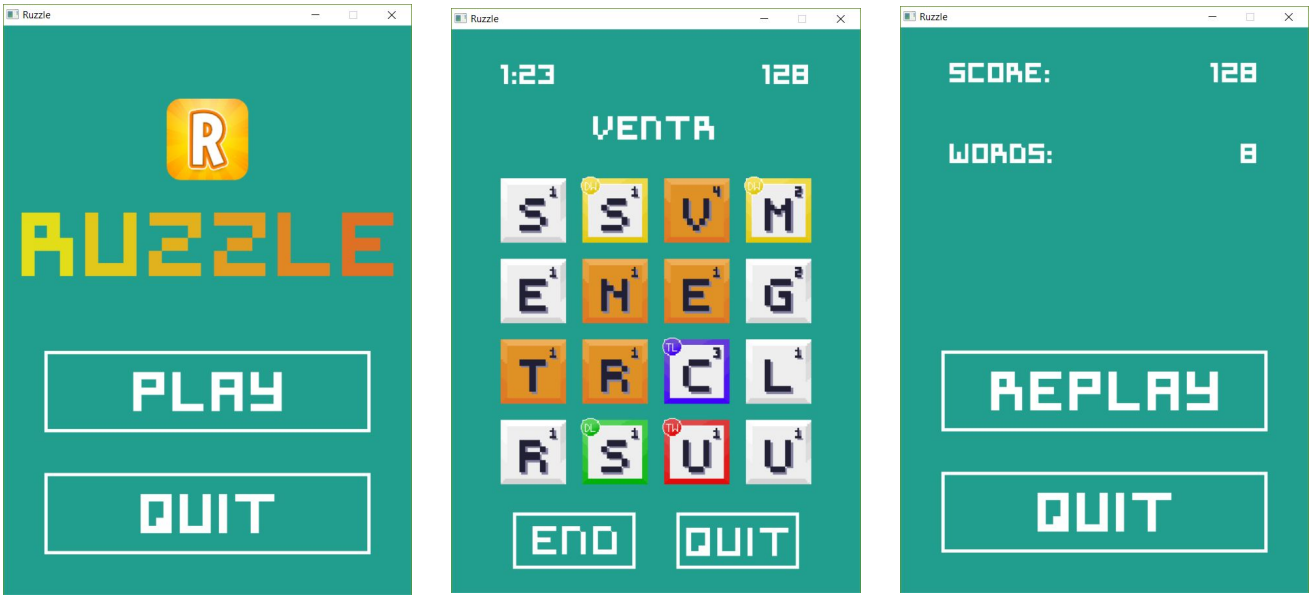
Concernant l'utilisation de Git, nous avons choisi d'installer GitKraken sur les conseils d'un autre membre du groupe qui en avait déjà fait usage. Le logiciel fournit une interface graphique qui remplace les commandes habituelles, ainsi qu'une visualisation claire des branches et des commits.

## C. Choix graphiques

Nous avons choisi de réaliser trois interfaces : une pour le début du jeu, une pour le jeu et une pour la fin du jeu, à la manière du Ruzzle original. Voici une description de ces trois interfaces :

- Ecran de début : logo et nom du jeu, deux boutons, un pour jouer, l'autre pour quitter le jeu
- Ecran de jeu : temps restant et score en haut, mot en cours en-dessous, grille de lettres et deux boutons, un pour terminer la partie, l'autre pour quitter le jeu
- Ecran de fin : score final, nombre de mots réalisés et deux boutons, un pour rejouer, l'autre pour quitter le jeu

Interfaces :




Le logo utilisé est le logo officiel du jeu. Les boutons sont des images que nous avons réalisées (couleurs inversées ici). Exemple du bouton Play :


Pour chaque lettre, nous avons fait deux images : une pour la lettre non sélectionnée, avec son poids, et une pour la lettre sélectionnée, en orange. Nous avons aussi fait des images pour les bonus, en fond transparent et légèrement plus grandes, afin de les superposer à l'image de la lettre. Exemple pour la lettre "A" et le bonus "Double Word" :


Tout le reste (score, timer, mot courant…) est affiché par l'intermédiaire de la librairie TTF de SDL, qui permet d'écrire du texte à l'écran.

## D. Conventions de codage

Nous avons choisi de séparer notre code en unités logiques, selon leur rôle. Nous avons fait l'équivalent de classes en programmation orientée objet, comme nous l'aurions fait si le projet était en C++, par l'intermédiaires de structures. Cela a permis d'obtenir un code modulable et simple, bien organisé.

Dans chaque .h, nous définissions une structure et déclarions les fonctions associées à celle-ci. Pour chaque structure (sauf exception), nous avons créé sur le même modèle une fonction de création createStructure et une fonction de destruction freeStructure.

La première alloue l'espace nécessaire à l'objet, l'initialise (en créant éventuellement les pointeurs sur les objets que la structure possède via les méthodes createSousStructure), et renvoie un pointeur sur la structure ainsi complète.

La deuxième fonction libère l'espace mémoire alloué lors de la création (en appelant éventuellement les méthodes freeSousStructures des pointeurs créés auparavant), libère l'espace alloué pour la structure elle-même et met à NULL la valeur de chaque pointeur. Ceci est une sécurité qui assure que le pointeur ne sera pas utilisé sans être initialisé par la suite, sans qu'une erreur ne se produise.

Notre boucle de jeu principale comporte trois grandes étapes : la gestion des évènements, c'est-à-dire les inputs du joueur (clics de souris ici), la mise à jour des éléments à afficher, puis leur affichage.

Chaque élément affichable (grille, timer, bouton…) possède dont au minimum la fonction d'affichage, au maximum les trois mentionnées plus haut. Chacune des fonction a la même signature, à l'exception de leur nom qui suit une nomenclature afin de simplifier la lecture du code.

Nous avons réalisé un diagramme des "classes" (structures + fonctions associées), mais étant donné qu'il n'était pas très lisible inclus dans un document, nous l'avons laissé à votre disposition dans l'archive présente.

# II. Rôle et fonctionnement des principales entités

## A. Boucle de jeu

L'entité principale du jeu est le GameManager. C'est lui qui s'occupe de tout lancer, de gérer la boucle de jeu et d'appeler les fonctions adéquates selon la situation. Notre programme a été pensé de telle sorte que le code du main soit très réduit :

```
// Declare which function to call on program exit
atexit(cleanExit);

// Initialize SDL and read complete dictionary
initSDL();
readRootDictionary();

// Launch game loop
gameManager = NULL;
initGameManager();
gameLoop();

// Free every pointer
freeGameManager();

return EXIT_SUCCESS;
```

Son rôle est de déclarer la fonction à appeler à la fin du programme, que ce soit à cause d'une erreur ou dans le cas normal. Cette fonction sert à fermer les librairies ouvertes. Il appelle ensuite la fonction d'initialisation de SDL (et des autres librairies), il lit le dictionnaire (cf II. E.), initialise le gameManager et lance la boucle de jeu. Une fois que le programme sort de cette boucle de jeu, cela signifie la fin du jeu. Le main n'a donc qu'à lancer la libération en chaîne de tout l'espace mémoire alloué pour les pointeurs des divers composants.

La boucle de jeu quant à elle est aussi très simple, elle s'exécute comme suit :

```
Tant que l'étape actuelle du jeu n'est pas Quitter
       Si l'étape actuelle du jeu est Rejouer
              Réinitialiser le GameManager
       Fin si
       Récupérer tous les inputs du joueur
       Gérer ces inputs
       Mettre à jour chaque élément
       Afficher chaque élément
Fin tant que
```

La gestion des évènements, la mise à jour et l'affichage se font dans des fonctions séparées, et font chacune appel à la fonction adéquate : selon l'étape actuelle du jeu (cf II. G.), ce n'est pas la même interface qui doit être appelée.

## B. Gestion de l'affichage

L'affichage est regroupé dans 3 interfaces, comme expliqué plus haut. Ces interfaces contiennent tous les composants nécessaires à leur fonctionnement et à leur affichage : timer, logo, score, grille… Ces structures sont elles-mêmes composées de plus petites structures utiles, notamment d'une Texture. Cette structure permet de gérer la création, l'affichage et la destruction d'une image ou d'un texte. Cela facilite grandement leur manipulation. Les boutons par exemple, ne nécessitent que le chemin de leur image et leur position dans la fenêtre pour être créés et affichés.

## C. Gestion de la grille

### Grille

Pour stocker les lettres de la grille, nous avons choisi de faire un tableau (lignes) de tableaux (colonnes) de pointeurs sur lettres. Cela donne donc :

```
Letter*** grid;
```

Cela rend l'accès à chaque lettre, selon sa ligne et sa colonne (stockés dans la lettre) très facile, ou selon les coordonnées du clic souris. Le pointeur sur Lettre nous évite de copier les objets lettres à chaque passage de paramètre et accès dans la grille. De plus, cette grille est modulable : la taille de la grille (son nombre de cellules par ligne/colonne) est défini comme constante dans un fichier extérieur (cf II. G.).

### Génération de grille

En ce qui concerne la génération de la grille, une simple fonction prenant en paramètre la taille de la grille (carrée) permet de générer dans un fichier .txt une grille "pseudo-aléatoire" de la taille voulue. Pseudo-aléatoire car chaque lettre a un poids qui est inversement proportionnel à sa valeur en point. La grille possède aussi la valeur des bonus généré, eux aussi avec un poids. Cela permet d'obtenir des grilles avec plus de lettres courantes dans les mots français (e, a, s…), ainsi que peu de lettres ayant des bonus.

Exemple d'une grille générée :

```
10
[v,0][f,4][u,1][v,4][i,1][o,0][x,4][v,4][t,4][t,4]|
[h,4][i,4][i,4][d,4][h,4][i,4][h,4][v,4][b,4][p,4]|
[f,0][p,4][c,4][p,4][l,4][s,0][a,0][l,4][n,4][s,4]|
[c,0][t,4][c,4][e,4][z,4][d,4][i,4][s,4][a,4][q,4]|
[b,4][h,2][p,4][p,1][w,1][e,4][a,4][g,4][u,4][l,4]|
[d,4][i,4][o,4][r,4][b,4][g,4][m,0][c,1][n,1][c,4]|
[t,4][b,4][o,4][m,4][l,4][s,4][s,0][s,4][u,4][t,4]|
[l,4][b,4][m,3][s,4][o,1][l,4][r,4][r,4][n,4][u,3]|
[o,0][e,4][l,4][b,4][b,4][r,0][a,1][w,4][v,4][r,4]|
[g,4][l,4][b,4][m,4][r,4][u,4][i,0][o,4][l,4][e,0]|
```

Sur la première ligne, la fonction écrit la taille de la grille. Puis elle écrit pour chaque ligne de la grille les lettres sous le format [lettre, bonus], le bonus étant sa valeur dans l'enum définie dans le code (cf. I. D.). La fin de chaque ligne est notée avec le caractère "|".

## D. Gestion des mots et des lettres

### Mot

La structure Mot est en réalité une structure d'abstraction par rapport à la structure Liste qui elle, est une structure de donnée basée sur le concept simple d'une liste chaînée de Lettre (LinkedLetter). Un mot est donc en traduction, dans le programme, un enchaînement de lettre dans un sens précis. La raison de ce choix est simple : une liste chaînée se lit dans un sens, comme un mot, de ce fait cela semblait tout indiqué, d'autant plus que cela permet d'accéder à chacune des lettres du mot indépendamment. L'intérêt principal est le traitement qui est donc possible et facilité pour les différents algorithmes gérant la grille.

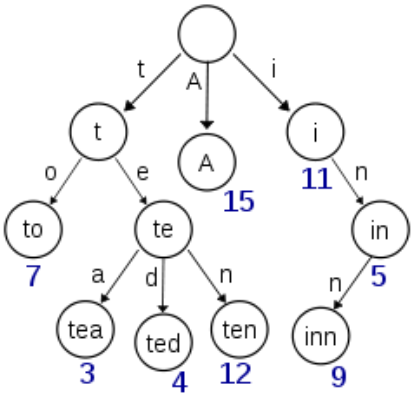
### Lettre

Quant aux lettres, nous avons choisi de regrouper la partie affichage de la partie technique : la structure contient aussi bien les informations nécessaires à son affichage (images de la lettre sélectionnée et non sélectionnée, bonus éventuel, position dans la fenêtre), que sa valeur, son caractère, etc. Il ne nous a pas semblé utile de séparer ces informations, étant donné que nous avions quasiment tout le temps besoin de ces deux types d'information quand nous accédions à une lettre donnée de la grille. Par contre, lors de la gestion du dictionnaire (cf II. E.), nous ne créons pas de nouvel objet lettre à chaque parcours de mot du dictionnaire. Cela serait fastidieux et inutile.

## E. Gestion du dictionnaire

### Trie Search Tree (Arbre préfixé)

Pour le dictionnaire, la solution la plus appropriée en terme de structure de données, nous a semblé, après réflexion, être un arbre préfixé. En effet, cette structure est réellement optimisée pour la gestion de beaucoup mots : à savoir que c'est ce type de structure qui est utilisée pour la prédiction dans les moteur de recherche comme Google. La structure est basique :




```
       typedef struct _trieNode
{
    struct _trieNode* children[ALPHABET_SIZE];
    bool endOfWord;
} TrieNode;
```

### Recherche des mots possibles dans la grille

La recherche des mots s'est organisée sous forme d'une méthode récursive parcourant chacune des lettres de la grille, puis créant un "mot" fictif pour chacune. Celui-ci prend la dernière lettre et observe les lettres alentours, et regarde par rapport au dictionnaire les lettres pouvant former un mot avec le mot fictif déjà rassemblé. Si une lettre est possible, elle est ajoutée au mot fictif, et la même méthode se réexécute avec cette dernière lettre.

La taille n'a pas d'importance : l'algorithme étant récursif, il s'adapte sans soucis. Seul le temps d'exécution diffère et il est très court. Les résultats sont enregistrés dans un nouvel arbre préfixé, plus court donc, permettant de comparer les mots entrés par le joueur le plus rapidement possible.

Exemple des première étapes de la première lettre d'une grille :

## F. Gestion du timer

Concernant le timer, nous pensions d'abord utiliser le SDL_AddTimer, mais cette fonction permet plutôt d'attendre une certaine durée avant de faire appel à une fonction callback. Nous voulions juste afficher le temps, puis quitter le jeu quand le temps était écoulé. L'usage de SDL_AddTimer pouvait être envisageable, mais nous avons préféré utiliser SDL_GetTicks qui permet d'obtenir le temps écoulé à partir du lancement du programme.

Nous avons récupéré de cette manière le temps à la création du timer :

```
        int startTime = SDL_GetTicks();
```

Puis pour calculer le temps qu'il restait avant la fin du jeu, nous avons fait la soustraction entre le temps retourné par le SDL_GetTicks courant et celui de départ. Le temps obtenu étant en millisecondes, nous l'avons converti en secondes :

```
        int currentTime = SDL_GetTicks();
int time = (currentTime - startTime) / 1000;
```

Enfin, pour l'affichage, il a suffit de faire différentes soustractions. Par exemple si le temps était inférieur à 50, il fallait afficher "1:" et 60 - temps, pour que le timer fasse un décompte.

```
        if (timer->time <= 50)
    sprintf(timeText, "1:%d", 60 - time);
```

### C. Autres fichiers

Nous avons également créé trois fichiers un peu particuliers :

- Enums.h : Ce fichier contient les définition des deux énumérations utiles à divers endroits de notre code. L'enum Step définit les différentes étapes du jeu : BEGIN pour l'écran de démarrage, GAME pour l'écran de jeu, END pour l'écran de fin, REPLAY pour indiquer qu'il faut relancer le jeu, QUIT pour indiquer qu'il faut quitter le jeu. L'enum Modifier définit les différents bonus associés aux lettres, comme dans le jeu initial. Nous avons choisi de définir ces énumérations dans un fichier séparé et non dans un .h d'une structure afin qu'elles soient sans dépendances à d'autres fichiers. Ainsi, chaque fichier peut inclure Enums.h sans danger, et accéder aux énumérations.
- Utils.h et Utils.c : Ce fichier regroupes des constantes nécessaires au jeu et des fonctions utilitaires. Les constantes sont par exemple la longueur et la largeur de la fenêtre, le nombre de lignes/colonnes de la grille ou la taille de l'image d'une lettre en pixels. Cela permet ensuite de définir d'autres constantes plus pointues (espacement entre chaque lettre, position de la grille) de manière relative et proportionnelle à ces constantes définies en dur. La fenêtre de jeu est donc relativement modifiable. Les fonctions utilitaires sont des fonctions de tirage aléatoires, que ce soit d'entiers, de lettres ou de bonus (utilisées dans la fonction de génération de grille, cf II. C.).

# III. Organisation & difficultés rencontrées

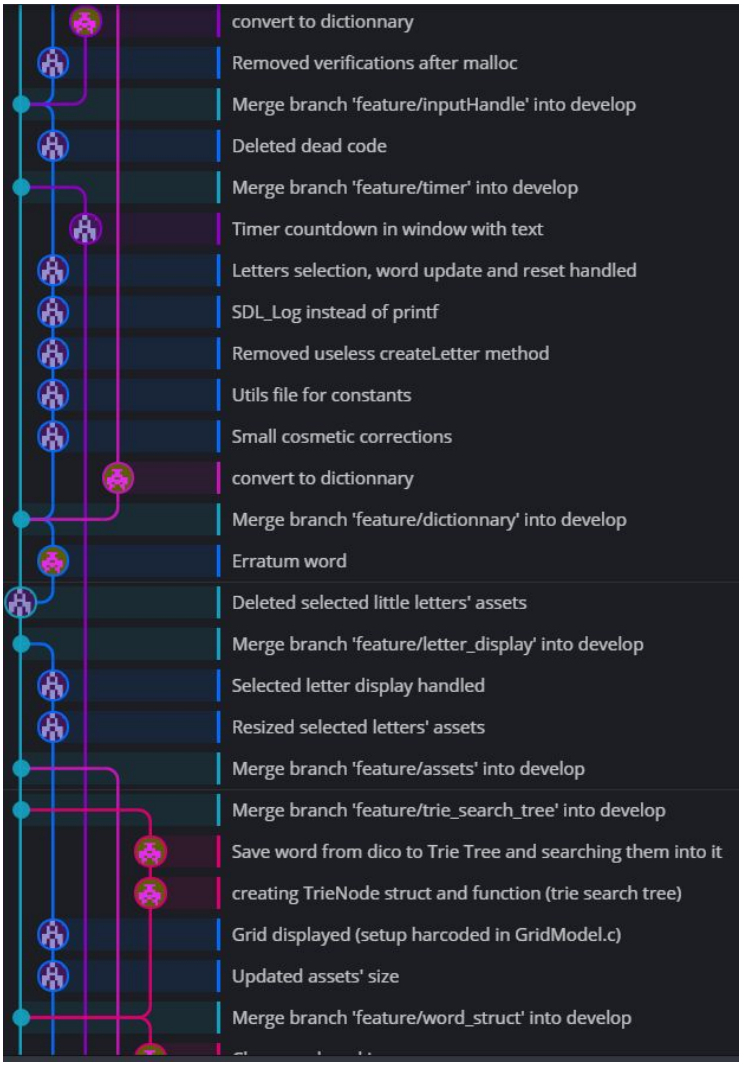
## A. Organisation

Dès la première semaine, nous avons fait une esquisse de la structure du code, avec un mini diagramme des classes. Cela nous a permis de dégager les principales problématiques et entités, de définir les grandes tâches et de les diviser en sous-tâches en fonction de leur priorité et de leurs dépendances, ainsi que de répartir ces tâches selon les envies et capacités de chacun.

Devant l'ampleur du travail qui nous attendait, nous avons choisi de venir aux séances chaque semaine, et non une semaine sur deux. Nous pensons qu'il aurait été très compliqué de faire sans, du moins cela aurait peut-être pris plus de temps si nous avions travaillé chacun chez soi.

Git a beaucoup aidé dans la gestion du partage des tâches. Nous avons pris l'habitude dès le départ de créer une branche par feature, et qu'une seule personne travaillerait dessus. En cours de développement, si cette branche nécessitait un code d'une autre branche qui venait d'être mergée dans develop, nous faisions simplement un rebase. A la fin, nous mergions et supprimions la branche de la feature associée. Cela nous a donné un arbre de commits et de branches plutôt clair et fluide, exceptées les quelques erreurs inévitables d'utilisation de git.

Exemple d'arbre :



## B. Difficultés rencontrées

Tout d'abord, et comme beaucoup de groupe nous pensons, nous avons eu énormément de difficultés avec l'installation des librairies SDL2/TTF/IMG sous Windows. L'EDI Code::Blocks n'a peut-être pas non plus facilité les choses, malgré la quantité de recherches que nous avons faites sur ce sujet. A force d'essais, de changement d'ordre des librairies, de changement de paramètres, de réinstallations, nous avons fini par le faire fonctionner, mais régulièrement quelque chose était modifié aléatoirement, et le programme ne retrouvait plus une librairie. Nous avons littéralement passé des heures là-dessus, et c'est du temps perdu où nous aurions pu coder et améliorer notre code. Nous avons fini par trouver un projet complet, qui incluait ces 3 librairies et dont les options de compilation et l'organisation du dossier fonctionnaient.

La seconde principale source de difficultés est le langage C utilisé. En effet, il est vraiment difficile de bien coder en C, et la moindre erreur d'inattention, d'ignorance ou approximation peut se révéler fatale. Il faut savoir exactement ce qu'on fait, quelles en sont les conséquences et comment cela fonctionne pour éviter les erreurs. De plus, des choses basiques dans tout autre langage un peu plus haut niveau deviennent longues et compliquées en C. La manipulation des chaînes de caractères par exemple, en particulier la concaténation, est une vraie torture. Nous avons dû faire face à des problèmes sans fin d'inclusions circulaires de .h, et ce malgré les forward declaration, car il fallait lire les directives #include dans l'ordre de parcours des fichiers depuis le main.c afin de savoir quels fichiers étaient compilés dans quel ordre. Nous avons eu des problèmes que nous avons fini par résoudre et que nous n'avons toujours pas compris, notamment la réception d'un signal SIGTRAP sur un free(), ce qui nous a amené à nous renseigner sur les undefined behaviour.

En résumé, beaucoup de difficultés très longues à résoudre qui auraient été presque toutes inexistantes en C++ par exemple.

# IV. Conclusion

En conclusion, nous avons globalement et malgré toutes nos difficultés beaucoup apprécié ce projet. Il nous a permis de coder un jeu dans le cadre de l'université, pour commencer, ce qui constitue une passion pour au moins deux des trois membres de ce groupe. Nous avons découvert ou redécouvert SDL2, qui reste tout de même une bonne initiation à tout ce qui est affichage. Nous avons manipulé l'environnement Code::Blocks et connaissons mieux son fonctionnement pour linker une librairie, ainsi que son debugger qui nous a été très utile. Nous avons beaucoup amélioré notre niveau en C, du moins il nous semble. Enfin, cela a été un challenge, et nous sommes fiers et heureux du résultat, même si nous aurions aimé avoir plus de temps pour le peaufiner et ajouter des fonctionnalités (affichage des 5 mots qui ont rapporté le plus de points à la fin, grille aléatoire à chaque début de partie...) ou améliorer certains comportements existants.