Projet #2: Jeu 2048

Auteurs :

- Perez Svetlitsky, Alejandro, alejandroperezsvet@gmail.com, LDD IM Groupe 2
- Sánchez Salcedo, Andrés Felipe, sanchez.andresfelipe0310@gmail.com, LDD IM Groupe 1

Indice

I. Niveau 0 : Structure de base du projet

1. Structure du projet
2. Représentation des tuiles
3. Les fonctions reliées au mouvement

3.a. Fonctions vérification de mouvement

3.b. Fonctions pour mouvement simple

3.c. Fonctions ajout des éléments

II. Niveau 1 : Mode console amélioré

III. Niveau 2 : Compilation et utilisation de Git

1. Compilation du projet
2. Utilisation de Git/GitHub

IV. Niveau 3 : Implantation Interface Graphique

1. Échelles et mesures utilisées pour la fenêtre
2. Fonction pour dessiner rectangle avec du texte
3. Représenter les tuiles sur l'écran
4. Animations des tuiles

Introduction

Le jeu de **2048** se joue sur une grille **4x4** avec des tuiles représentant des puissances de deux. Le but est d'atteindre la tuile 2048 en déplaçant toutes les tuiles dans une direction *(droite, gauche, haut* ou *bas)*.

En tant que réalisateurs, nous avons été inspirés par l'idée de créer un jeu sur ordinateur enrichi d'une interface graphique. Pour ce faire, nous avons utilisé **C++,** langage de programmation couramment utilisé dans le domaine des jeux videós.

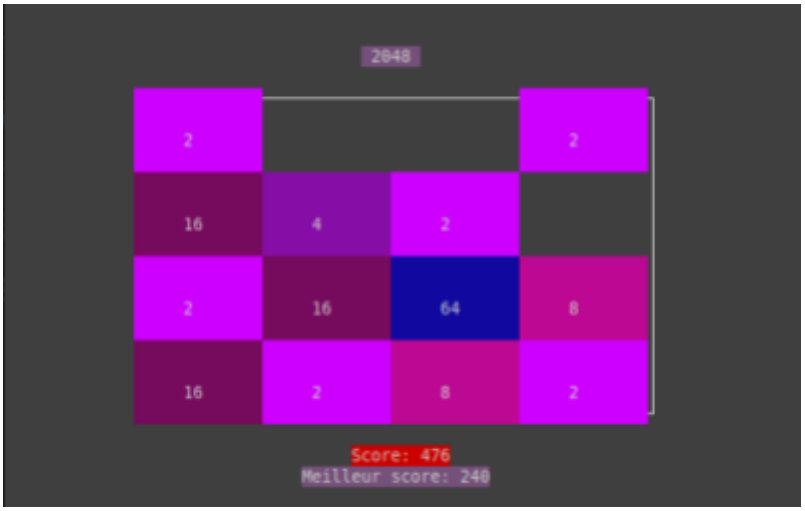
Le projet se décompose en trois grandes parties:

1) La Structure Générale de Base
2) L'Ajout de Fonctionnalités en Mode Console
3) L'Implantation d'Interface Graphique

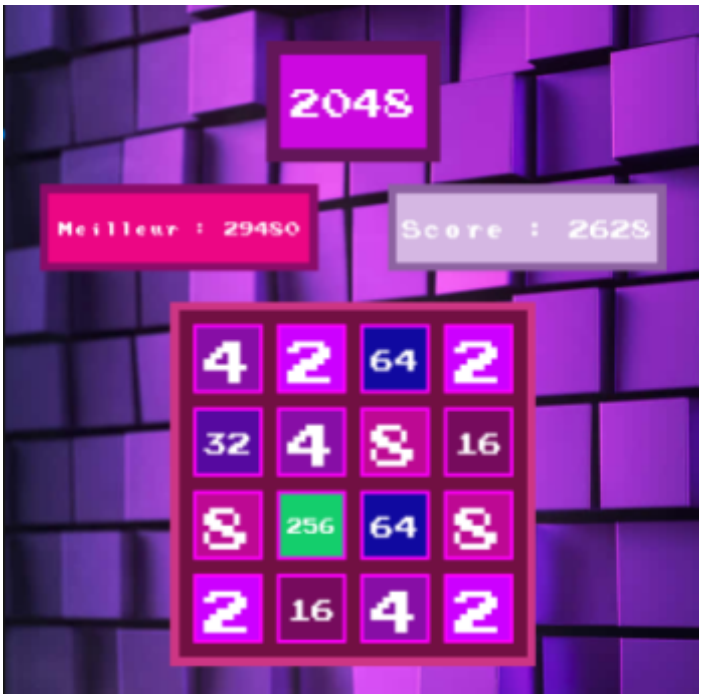
Chacune des parties est accompagnée de test permettant de valider le fonctionnement

Images

- Console



- Interface Graphique



I. Niveau 0 : Structure de base du projet

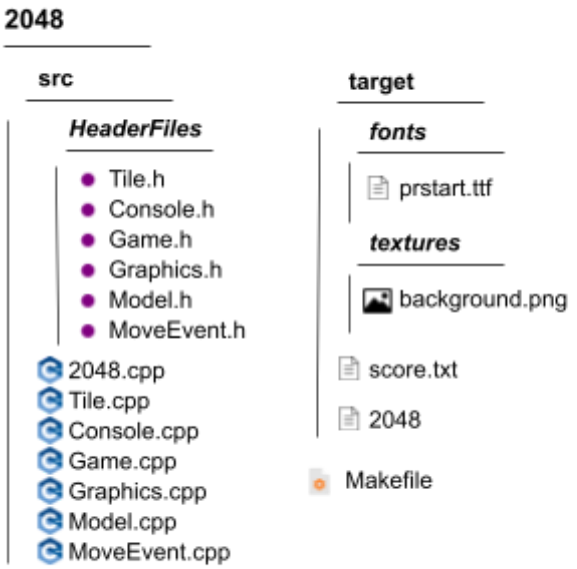


FIG. 1.1 FICHIERS DU PROJET

1. Structure du projet

Le projet est composé de **7** fichiers **C++**, dont l'un contient la fonction **main** pour démarrer le programme et les autres **6** correspondent aux différentes classes utilisées, chacune accompagnée de son fichier d'en-tête *(Header Files)* respectif.

Le répertoire **target** est utilisé pour stocker le fichier exécutable lors de la compilation, ainsi que d'autres fichiers nécessaires au programme.

De même, un **Makefile** est utilisé pour compiler le projet. On détaillera son fonctionnement postérieurement.

- Les fonctions de logique et des règles de Jeu se trouvent dans les fichier **Model.cpp** | **Model.h**, qui utilisent comme base la classe **Tile** définie dans les fichiers **Tile.cpp** | **Tile.h**.

- Les fichiers **Game.cpp** | **Game.h** permettent de gérer le jeu pour initialiser les variables à utiliser, et de même pour commencer ou finir le jeu.

- Les fichiers **Console.cpp** | **Console.h** et **Graphics.cpp** | **Graphics.h** possède respectivement les fonctionnalités pour afficher le jeu en mode console ou interface graphique avec l'utilisation des classes.

2. Représentation des tuiles

Pour représenter les tuiles de notre Jeu, nous allons utiliser une classe intitulé **Tile** *(Tuile en anglais)* dans laquelle on va garder toutes les variables de position de la tuile, donc les indices i, j dans le plateau du jeu, la valeur de la tuile mais aussi tous les mouvements faits par la tuile *(Cela nous sera utile dans le cas des animations)*.

Nous avons créé ainsi une liste dans laquelle on stocke toutes les classes des tuiles d'une valeur distincte de 0 : ses tuiles sont représentées par leur absence dans cette liste.

3. Les fonctions reliées au mouvements

3.a. Fonctions vérification de mouvement

Pour vérifier si un mouvement est possible nous parcourons les tuiles de la ligne/colonne concernée, deux à deux, dans la direction opposée au mouvement.

Les cases de départ et d'arrivée sont positionnées selon la direction du mouvement (par exemple, pour un mouvement à gauche, la case de départ est à droite et celle d'arrivée à gauche). Lors du parcours, deux cas sont possibles :

- Soit la première case (tuile de départ) est vide et la deuxième (tuile d'arrivée) est non vide, auquel cas la tuile d'arrivée peut être déplacée vers la première
- Soit les deux cases ont la même valeur et peuvent être additionnées

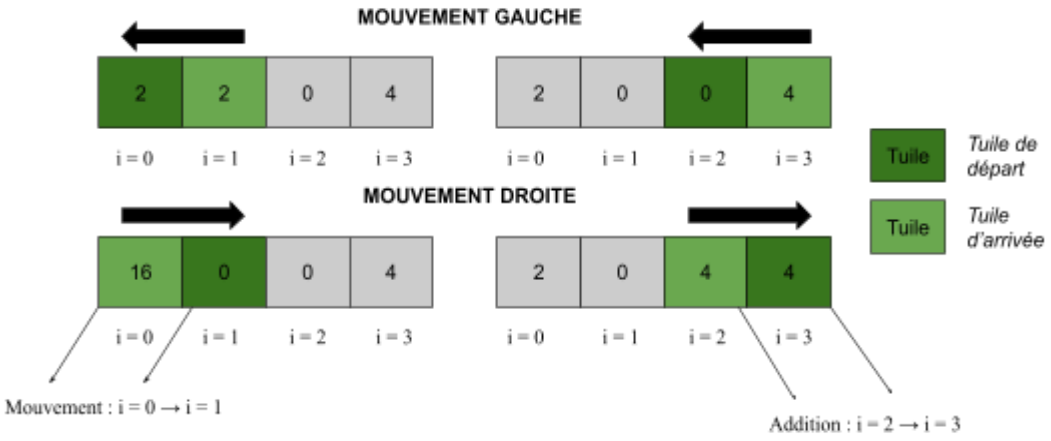


FIG. 1.2 EXEMPLE VÉRIFICATION DE MOUVEMENT

3.b. Fonctions de mouvement

Les fonctions de déplacement identifient les cases vides et déplacent les tuiles non vides vers ces cases vides selon la direction spécifiée (haut, bas, gauche, droite). Elles parcourent les lignes/colonnes et déplacent les tuiles dès qu'une case vide est trouvée.

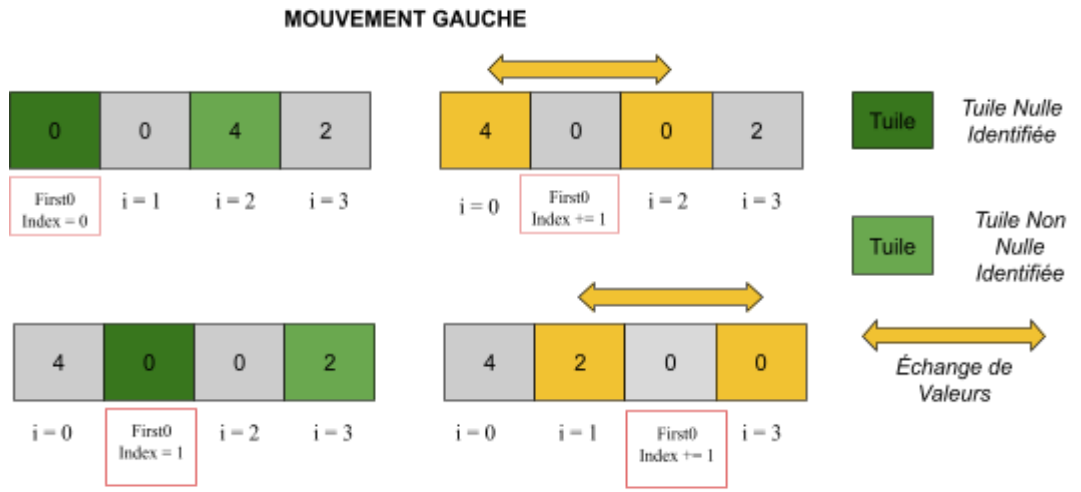


FIG. 1.3 EXEMPLE MOUVEMENT

3.c. Fonctions pour rajouter les éléments

On vérifie si deux tuiles peuvent être fusionnées. Si c'est le cas: on additionne les deux tuiles, on met à jour le score et on crée une animation de déplacement et de fusion. La tulle de départ est ensuite supprimée.

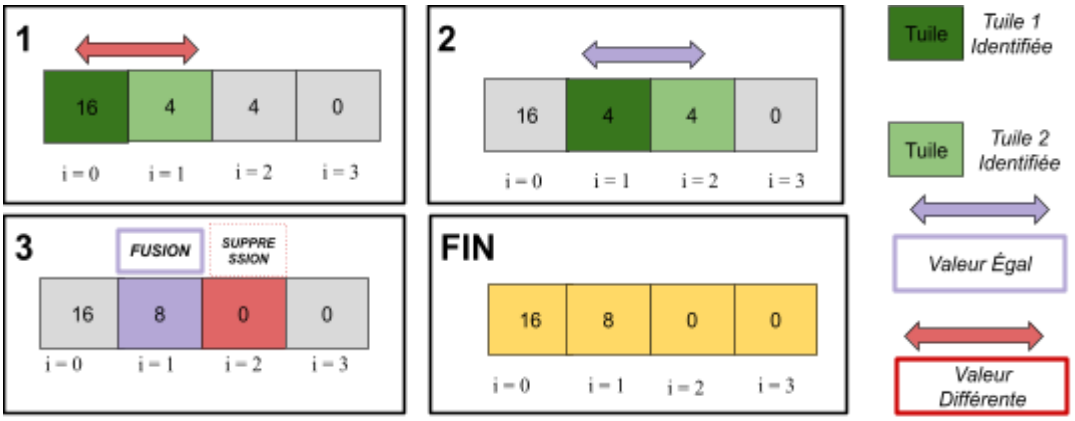


FIG. 1.4 EXEMPLE MOUVEMENT

II. Niveau 1 : Mode console amélioré

1. Ajout Couleur à la Console

Pour ce faire, nous avons utilisé la bibliothèque **ncurses**.

La première étape consiste à vérifier la compatibilité des couleurs avec le terminal et d'initialiser le mode couleur via **start_color().** Ensuite, des *paires* de couleur standard sont définies à l'aide de **init_pair().**

Pour les tuiles, des couleurs spécifiques sont définies en RGB avec **init_color()** et attribuées aux tuiles du jeu. Enfin, chaque couleur est associée à une paire, ce qui permet de combiner texte blanc sur un fond coloré.

2. Amélioration Jouabilité avec Ajout des Flèches

La fonction *isValideMove( )* vérifie si le déplacement demandé est valide avant de l'exécuter.

Tout d'abord, la fonction reçoit une touche en paramètre (variable *key*) et utilise une structure de contrôle appelée **switch** (ligne 91) pour identifier la flèche qui a été appuyée: *KEY_(DIRECTION)* pour la flèche de direction *DIRECTION.*

Pour chaque direction, sa fonction respective *canMove**Direction**()* est appelée pour vérifier si un mouvement est possible dans cette direction. Si le déplacement est valide, la fonction *move**Direction**()* est appelée pour déplacer les tuiles dans la direction choisie.

Si le mouvement a été effectué alors la fonction renvoie *true:* il s'agit d'un mouvement valide. Sinon elle examine la touche suivante ou, en dernier cas, renvoie false si aucun mouvement valide n'a été trouvé.

3. Amélioration Jouabilité avec Affichage Automatique du Plateau

La fonction *drawBoard( )* permet d'afficher le Plateau du Jeu de façon automatique.

Tout d'abord, elle dessine une bordure autour du Plateau en utilisant la fonction box (ligne 62). Ensuite elle parcourt chaque case du tableau (ligne 63-64). Pour chaque case valide (i.e. non vide), on récupère la valeur de la tuile et on détermine la couleur associée à cette tuile à l'aide de *findNumberColor(value).*

La fonction dessine la case avec la couleur respective et centre la valeur de la tuile dans la case en calculant sa position (ligne 71-72) selon la taille des cases (*cellHeight* et *cellWidth)* et les coordonnés (*startY* et *startX*) pris en paramètre.

On utilise la commande *mvwprintw* pour afficher le contenu de chaque case. Enfin, après avoir dessiné toutes les cases, la fonction met à jour l'écran avec *wrefresh(win)* afin de refléter les modifications apportées au Plateau du Jeu.

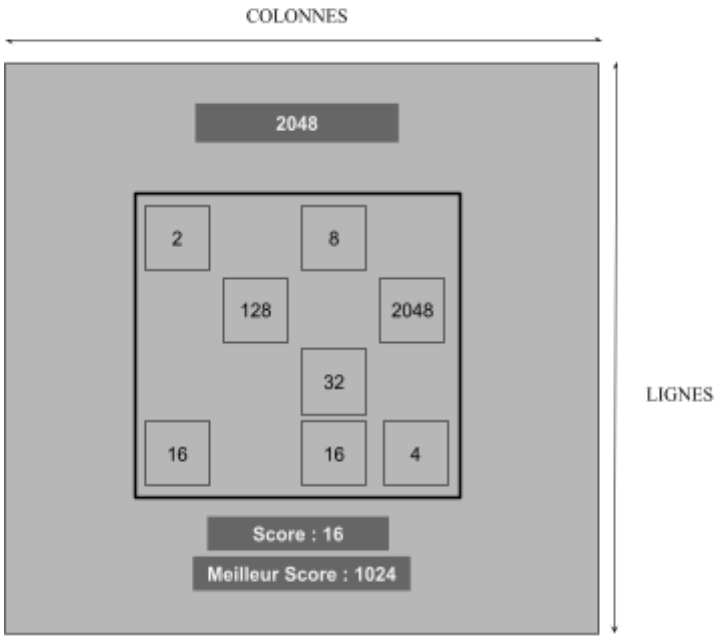


FIG. 2.1 MESURES POUR
INTERFACE GRAPHIQUE

4. **Calcul du score du Jeu avec structure de données qui associe le plateau et le Score.**

Pour des mouvements d'addition, on ajoute simplement le score précédent avec le numéro de la nouvelle case produite. La variable score est ainsi gardée dans la classe **Model**.

Le score est stocké dans un fichier nommée **score.txt**, et lors d'initialiser la classe **Game**, on a des fonctions pour extraire la valeur du score dans le fichier, et puis le passer en paramètre pour créer la classe **Model**.

III. Niveau 2 : Compilation et utilisation de Git

1. **Compilation du projet**

Ce Makefile est utilisé pour automatiser le processus de compilation du projet appelé *2048.*

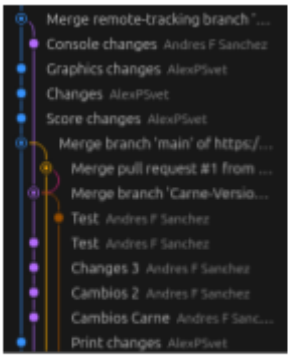
Il commence par identifier tous les fichiers sources *.cpp* situées dans le dossier *src* grâce à la commande *find* (ligne 13). Ces fichiers sont transformés en *fichiers objets .o* (ligne 14) qui sont à leur tour stockés dans le dossier target.

Le Makefile utilise des options spécifiques pour inclure des bibliothèques nécessaires au fonctionnement du programme, comme par exemple SFML ou ncurses (ligne 14). Ces bibliothèques permettent de gérer des aspects d'interface graphique et du terminal pour le jeu. Les fichiers objets *.o* sont ensuite combinés pour produire l'exécutable final placé dans le répertoire target.

Le Makefile inclut également des règles pour s'assurer que les répertoires nécessaires, comme *target*, soient créés avant la compilation (ligne 19 et 24).

Enfin, une commande *clean* (ligne 26) est définie pour supprimer tous les fichiers générés, y compris *target* et l'exécutable, afin de re-commencer la compilation depuis zéro.

2. Utilisation de Git/GitHub



IMG. 2.1 MODIFICATIONS GIT EXEMPLE

Pour l'organisation du projet, nous avons décidé d'utiliser **GitHub** pour ainsi avoir l'interface web qui nous permet de gérer les utilisateurs, l'accès au projet mais aussi pour voir les fichiers en ligne.

Git est intégrée avec **GitHub**, donc on dispose de toutes ses fonctionnalités pour faire les modifications au projet chacun de nous. On dispose de deux branches principales, pour que chacun puisse faire des modifications au projet, et puis mettre les deux versions dans une version finale.

IV. Niveau 3 : Implantation Interface Graphique

1. Échelles et mesures utilisées pour la fenêtre

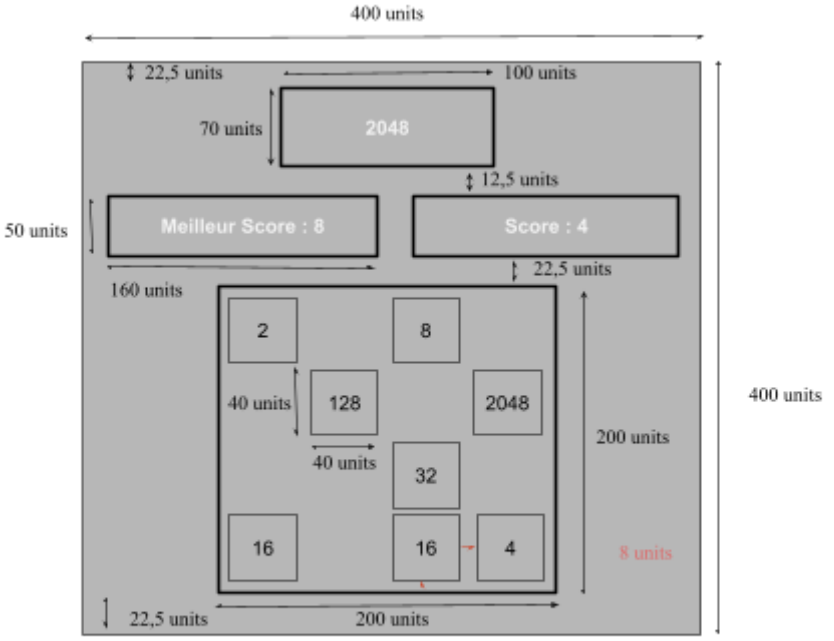


FIG. 4.1 MESURES POUR INTERFACE GRAPHIQUE

Pour la fenêtre, nous avons fait le choix de qu'elle soit une fenêtre carrée. Par exemple de 600 x 600 pixels.

Ainsi, pour que la fenêtre puisse s'adapter selon la résolution carré choisie, nous avons créé une **échelle d'unités**, dans laquelle on divise par **400** la longueur et largeur de la fenêtre.

Cela permet de conserver les mêmes tailles et ordre des éléments selon la résolution choisie.

2. Fonction pour dessiner rectangle avec du texte

Nous avons différentes parties du jeu dans lesquelles on veut dessiner des **rectangles** avec du **texte** à l'intérieur : les informations du jeu tels que le score, le meilleur score, les tuiles avec du texte variable, car il change de taille selon le score, meilleur score ou le numéro de la tuile.

Nous avons ainsi implanté une fonction qui dessine un rectangle de taille fixe, et selon le nombre de caractères dans le texte, il lui donne une taille pour que le texte reste à l'intérieur du rectangle de manière centré.

Le schéma suivant permet ainsi de calculer les coordonnées et longueurs du rectangle selon une bordure spécifique.

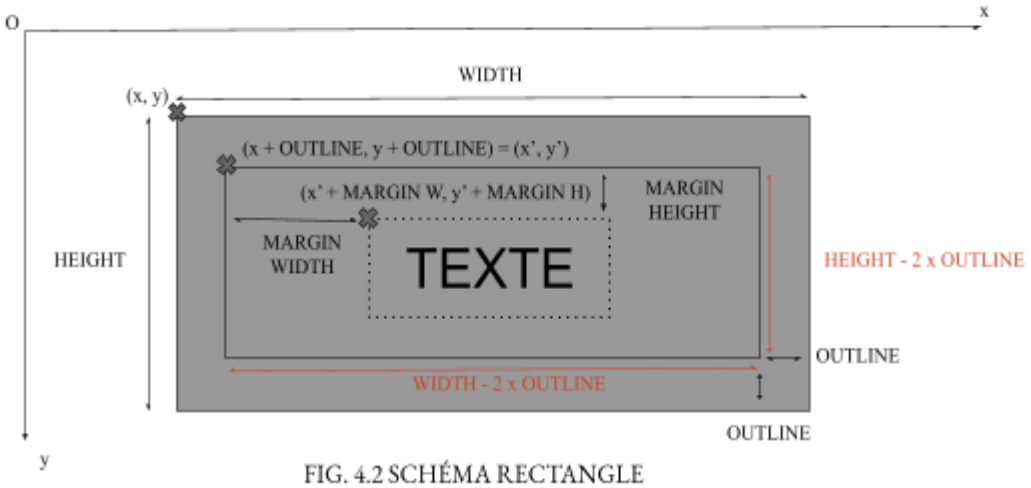


FIG. 4.2 SCHÉMA RECTANGLE AVEC DU TEXTE

Pour le texte, on veut calculer la taille occupé pour un caractère nommée **charSize**, nous avons le nombre de caractères **nbChar**, et pour chaque composante **widthText**, **heightText** et **charSizeWidth**, **charSizeHeight**, on calcule :

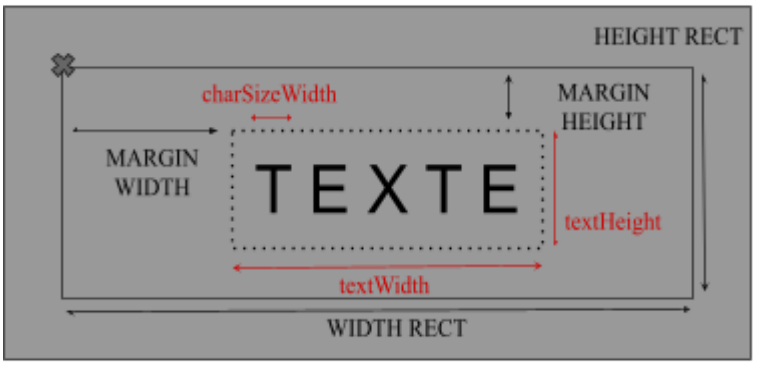


FIG. 4.3 CALCUL TAILLE D'UN CARACTÈRE DU TEXTE

CALCULS

$$\textbf{textWidth} = \textbf{widthRect} - 2 \times \textbf{marginWidth}$$
$$\textbf{textHeight} = \textbf{heightRect} - 2 \times \textbf{marginHeight}$$

$$\textbf{charSizeWidth} = \textbf{widthText} \;/\; \textbf{nbChar}$$
$$\textbf{charSizeHeight} = \textbf{heightText}$$

Ainsi, on prend le plus petit des tailles entre **charSizeWidth** et **charSizeHeight**, et on donne cette taille à chaque caractère du texte.

3. Représenter les tuiles sur l'écran

On dessine les tuiles en utilisant la fonction pour les dessiner, mais pour calculer leurs coordonnées, nous devons faire la chose suivante :

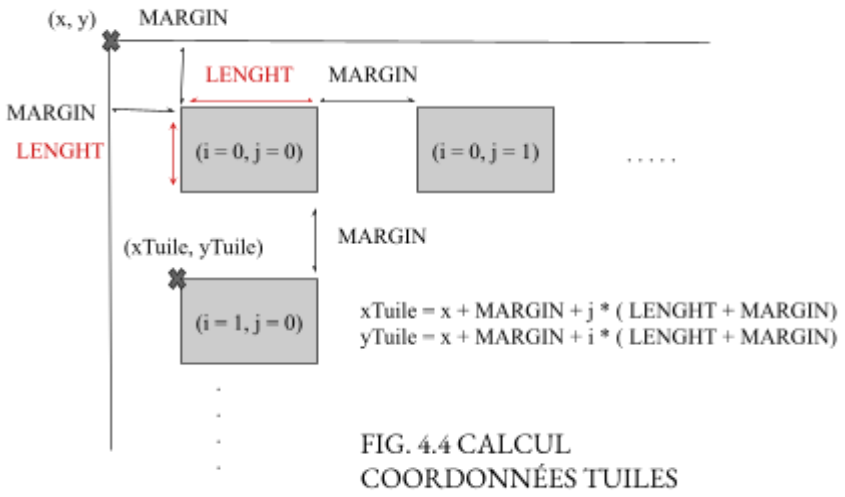


FIG. 4.4 CALCUL COORDONNÉES TUILES

Nous devons ainsi lorsqu'on calcule la coordonnée d'une tuile de rajouter tout au début l'espace *(margin)* entre le rectangle représentant le plateau et les tuiles, et puis la somme j, i fois de la longueur du carré représentant la tuile et l'espace entre les tuiles pour les coordonnées x, y

4. Animations des tuiles

Pour faire les animations des tuiles, on modifie à chaque affichage d'image dans la fenêtre du jeu les coordonnées des tuiles en animation. Pour faire cela, pour chaque tuile on stocke une liste des classes **MoveEvent** pour savoir quels mouvements effectuer. On modifie ainsi les coordonnées des cases pour les déplacer petit à petit du point de départ au point d'arrivée.

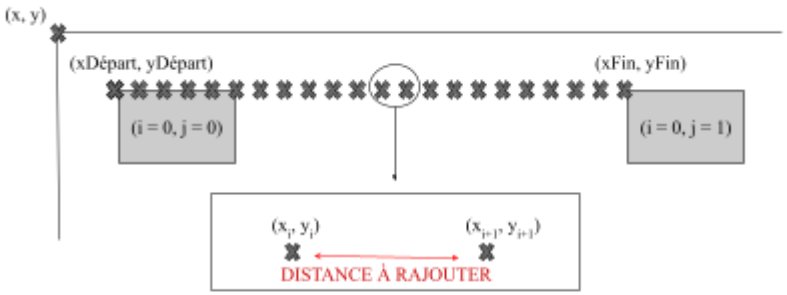


FIG. 4.5 SCHÉMA ANIMATIONS

On **augmente / diminue** les coordonnées dans la composante x pours des mouvements horizontales et même principe pour la composante y avec les mouvements verticaux. Cela crée ainsi l'effet de mouvement animé lorsqu'on modifie petit à petit les coordonnées des tuiles.

Conclusion

Ce projet nous a permis d'étaler nos connaissances informatiques, notamment sur l'environnement pour réaliser le projet en se familiarisant avec le système d'exploitation linux mais aussi les logiciels tels que Visual Studio Code pour écrire du code, Git et GitHub pour garder le projet et le partager, ainsi que le MakeFile pour compiler l'exécutable finale.

D'autre part, nous avons appris non seulement à ajouter des bibliothèques mais aussi à apprendre leur utilisation, tels que **ncurses** pour la terminale et **SFML** pour une interface graphique.

Nous avons pris à peu près **35-40** heures pour finaliser le projet, ce qui inclut le travail individuel de chacun d'entre nous mais aussi le travail qu'on a mis pour travailler ensemble.

Même si on aurait voulu arriver à la partie finale de créer une application mobile pour iOS et Android, nous avons presque accompli tous les objectifs qu'on s'est proposés.

Les difficultés qu'on a retrouvé au cours de la réalisation c'est l'implantation des animations en raison qu'on devait changer la structure de base du projet du stockage des tuiles. De même, on a aussi dû apprendre les concepts par rapport à l'utilisation des **Header Files** et aussi des classes pour regrouper des fonctions et variables selon un propos.

Finalement, pour le MakeFile on a dû de même explorer toute la syntaxe et sémantique du code pour le comprendre, ce qui a posé aussi un défi pour nous.