Rapport du Projet C – Rubik's Cube

Mathieu CHANTOT et Clément LE STRAT

EFREI L1 S2 – Mai 2021

Table des matières

1 Introduction

Le but de notre projet est de réaliser un programme en C permettant de manipuler un Rubik's cube. Le programme initialise un Rubik's cube que l'utilisateur peut mélanger, modifier et manipuler. De plus, un algorithme permet de résoudre le Rubik's cube en indiquant la liste des mouvements réalisés. Pour ce faire, l'utilisateur interagit avec un menu disponible en console. Une représentation du Rubik's cube en deux dimensions et en couleurs, est affichée à chaque étape.

Un "cubie" désigne un carré de couleur, il y en a neuf par face. Un cubie peut être un coin, une arête ou un centre, il dispose alors d'un nombre de "voisins" variable en fonction de son type. Soit deux voisins pour un coin, un voisin pour une arête, ou pas de voisins pour un cubie central.

Le programme a été développé sous Linux, et dépend donc des consoles utilisées par ce système, dans notre cas "konsole" (KDE), et Gnome Terminal (Gnome).

2 Présentation du projet et information fonctionnelle du programme

Le programme présente à l'utilisateur différentes fonctionnalités, en partant d'un Rubik's cube réalisé :

- "Mélanger le Rubik's cube", permet de réaliser automatiquement une suite aléatoire de vingt à trente mouvements sur le Rubik's cube.
- "Résoudre le Rubik's cube", lance un algorithme de résolution qui permet de le résoudre automatiquement. Les différents mouvements réalisés sont indiqués à l'utilisateur.
- "Faire vos propre mouvements", l'utilisateur peut réaliser manuellement les mouvements qu'il souhaite.
- "Placer les couleurs", permet à l'utilisateur de choisir pour chaque cubie et ses voisins une combinaison de couleurs disponibles. A chaque étape, la liste des combinaisons restantes est mise à jour de telle sorte que le Rubik's cube soit correctement complété.
- "Afficher le Rubik's", permet d'afficher à nouveau la représentation du Rubik's cube.
- Et enfin, il est possible de quitter le programme.

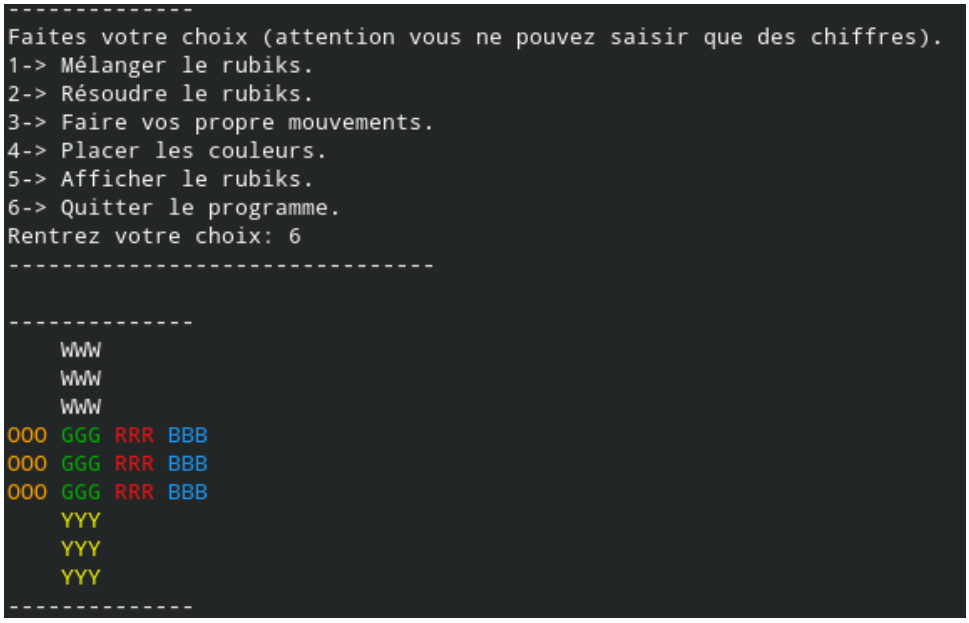
3 Description de l'interface

L'interface s'affiche en console et permet à l'utilisateur de choisir entre les différentes options offertes par notre programme avec, bien sûr, une saisie sécurisée qui ne permet pas à l'utilisateur de faire planter le programme ou de choisir des options inexistantes.

Dans un premier temps, l'utilisateur devra choisir entre les options « mélanger le rubik's cube », « résoudre le Rubik's cube », « choisir ses propres mouvement », « placer vos couleurs », « afficher le Rubik's cube » ou encore « quitter le programme ».

Dans un second temps, si l'utilisateur choisit l'option « placer vos couleurs », il devra rentrer les couleurs des cubies et de ses voisins que le programme lui propose. Les couleurs choisies et leurs inverses sont ensuite supprimés des choix (par exemple les combinaisons blanc et orange, et orange et blanc seront supprimées toutes les deux si l'utilisateurs choisie l'une d'entre-elles) jusqu'à ce que tous les cubies soient entrés. Les entrées sont donc maitrisées, pour éviter que l'utilisateur puisse saisir des combinaisons inexistantes sur un vrai Rubik's cube. Mais attention le Rubik's cube "saisit" par l'utilisateur ne sera pas toujours réalisable car certaines associations de coins rendent la résolution du Rubik's cube impossible.

Si l'utilisateur choisit l'option « faire vos propres mouvements », il devra alors tout d'abords choisir quel mouvement (horaire ou antihoraire), il souhaite réaliser, et ensuite choisir sur quelle face ce mouvement sera appliqué (blanc, orange, vert, etc.).



4 Informations sur le programme

Le programme a été développé sous linux. Il utilise des bibliothèques couramment utilisées en C : stdlib.h , stdio.h et math.h.

Le code source se divise en plusieurs fichiers :

main.c: contient la fonction principale

draw.c/.h: contient les fonctions d'affichages du Rubik's cube en console. Différentes fonctions utilisant la bibliothèque nCurses ont été réalisées en vue d'un affichage en console utilisant cette librairie, mais cette fonctionnalité n'est pas encore disponible.

menu.c/.h: contient les fonctions d'interactions avec l'utilisateur, les différents menus et les fonctions qui permettent la personnalisation du Rubik's cube.

rubiks.c/.h: contient les fonctions permettant de créer et d'initialiser le Rubik's cube

move_rubiks.c/.h: contient les fonctions permettant de réaliser des mouvements sur le Rubik's cube ainsi que les algorithmes de résolution.

Les programmes ont été écrit en gardant en tête l'idée de bien séparer les logiques de saisies, et d'affichage des partie algorithmiques de résolutions, mélanges, saisies, etc. Par conséquent, il est possible de passer à nCurses sans tout réécrire.

La documentation du programme est disponible en annexe, elle est générée automatiquement à l'aide de Doxygen.

5 La représentation du Rubik's cube en c

Nous avons fait le choix de représenter le Rubik's cube sous la forme de structures contenant les différentes informations nécessaires. Dans cette partie, nous allons expliquer comment le Rubik's cube est initialisé lorsque le programme est lancé, mais nous allons aussi voir les différentes actions que rendent possibles cette représentation du Rubik's cube.

5.1 Définitions des structures

Dans un premier temps, on définit les différentes structures dont nous aurons besoin, il y aura ainsi les définitions de type énuméré « T_COLOR » (un énum qui définit les couleurs des cubies et des faces), « T_SIDE » (un énum qui définit la position des différentes faces), et « T_CUBIE_TYPE » (un énum listant les différents types de cubies : coin, centre, et arête), et les définitions de type structuré « neighbour » (qui permettra de stocker les informations des voisins d'un cubie), « cubies » (qui permettra de stocker les informations des cubies telles que sa position, son numéro relatif à la face, sa couleur, ses voisins, …)

et « rubiks_side » (qui permettra de stocker les informations concernant les faces, telles que la couleur de la face, les 9 cubies, …).

```
// Défnition de la structure représentant le rubiks et des informations nécessaires.


//Définition des couleurs des faces et des cubies.
définition de type enum {WHITE=0, ORANGE=1, GREEN=2, RED=3, BLUE=4, YELLOW=5, GREY = 6, NO_COLOR = 7} T_COLOR

//Définition des faces adjacentes à une face du Rubik's Cube.
définition de type enum {UP=0, RIGHT=1, DOWN=2, LEFT=3} T_SIDE

//Définition des types de chaque cubie.
définition de type enum  {CORNER=0, EDGE=1, CENTER=2} T_CUBIE_TYPE

//Définition des adjacents à un cubie
définition de type structure {
    T_COLOR num_side
    num_cubie : entier
} neighbour

//Définition des cubies, qui sont les petits cubes de couleur rattachés à une face
definition de type structure {
    x, y, num : entiers
    T_CUBIE_TYPE type
    T_COLOR color
    T_COLOR cubie_side
    neighbour neighbours[2]
} cubies

//Définition d'une face du Rubik's Cube
définition de type structure {
    T_COLOR neighbour_side[4]
    T_COLOR opposite_side
    T_COLOR side;
    cubies cubie[9]
} rubiks_side
```

5.2 L'initialisation des structures / Création du Rubik's cube

Dans un second temps, on initialise « à la main » ces structures pour avoir toutes les informations dont l'on pourrait avoir besoin pour la suite dans les différents algorithmes :

```
//Initialisation du rubik's cube. Au départ, il est résolu.
Fonction rubiks_creation(rubiks_side *rubiks) {
    face, cubie : entiers

    // On va former chacune des faces et à chaque fois définir ses voisins et son opposé
    Pour (face = WHITE; face <= YELLOW; face++) Faire {

        rubiks[face].side = face
        // On considère que la face en cours de création est face à nous.
        // Et à chaque fois on détermine les adjacents de la face en cours de traitement, ainsi que sa face opposée

        Si (face == WHITE) Alors {
            rubiks[face].neighbour_side[UP] = BLUE
            rubiks[face].neighbour_side[RIGHT] = RED
            rubiks[face].neighbour_side[DOWN] = GREEN
            rubiks[face].neighbour_side[LEFT] = ORANGE
            rubiks[face].opposite_side = YELLOW
        }
        Si (face == ORANGE) Alors {
            rubiks[face].neighbour_side[UP] = WHITE
            rubiks[face].neighbour_side[RIGHT] = GREEN
            rubiks[face].neighbour_side[DOWN] = YELLOW
            rubiks[face].neighbour_side[LEFT] = BLUE
            rubiks[face].opposite_side = RED
        }
        Si (face == GREEN) Alors {
            rubiks[face].neighbour_side[UP] = WHITE
            rubiks[face].neighbour_side[RIGHT] = RED
            rubiks[face].neighbour_side[DOWN] = YELLOW
            rubiks[face].neighbour_side[LEFT] = ORANGE
            rubiks[face].opposite_side = BLUE;
        }
        Si (face == RED) Alors {
            rubiks[face].neighbour_side[UP] = WHITE
            rubiks[face].neighbour_side[RIGHT] = BLUE
            rubiks[face].neighbour_side[DOWN] = YELLOW
            rubiks[face].neighbour_side[LEFT] = GREEN
            rubiks[face].opposite_side = ORANGE
        }
        Si (face == BLUE) Alors {
            rubiks[face].neighbour_side[UP] = WHITE
            rubiks[face].neighbour_side[RIGHT] = ORANGE
            rubiks[face].neighbour_side[DOWN] = YELLOW
            rubiks[face].neighbour_side[LEFT] = RED
        }
        Si (face == YELLOW) Alors {
            rubiks[face].neighbour_side[UP] = GREEN
            rubiks[face].neighbour_side[RIGHT] = RED
            rubiks[face].neighbour_side[DOWN] = BLUE
            rubiks[face].neighbour_side[LEFT] = ORANGE
            rubiks[face].opposite_side = WHITE
        }


       // Initialisation des cubies de la face en cours de traitement. On parcourt donc le tableau de 0 à 8
       Pour (cubie = 0; cubie < 9; cubie++) Faire {

           // donner a chaque cubie la couleur de sa face
           rubiks[face].cubie[cubie].color = rubiks[face].side
           // donner à chaque cubie la couleur de sa face
           // On sauvegarde la face de rattachement du cubie en cours
           rubiks[face].cubie[cubie].cubie_side = rubiks[face].side

           // On défini le type de cubie en cours de traitement, c'est soit un coin, soit un côté, soit un centre
           Si (cubie == 0 ou cubie == 2 ou cubie == 6 ou cubie == 8) Alors {
               rubiks[face].cubie[cubie].type = CORNER
           }
           Si (cubie == 4) Alors {
               rubiks[face].cubie[cubie].type = CENTER
           }
           Si (cubie == 1 ou cubie == 3 ou cubie == 5 ou cubie == 7) Alors {
               rubiks[face].cubie[cubie].type = EDGE
           }

           // On sauvegarde le numéro de cubie
           rubiks[face].cubie[cubie].num = cubie

           // Et enfin, on défini ses coordonnées car elles nous servirons aux rendus à l'écran
           rubiks[face].cubie[cubie].x = rubiks[face].cubie[cubie].num % 3
           rubiks[face].cubie[cubie].y = (rubiks[face].cubie[cubie].num - rubiks[face].cubie[cubie].x) / 3

       }
    }

    // Et enfin on défini les faces adjacentes
    rubiks_neighbour(rubiks)
}
```

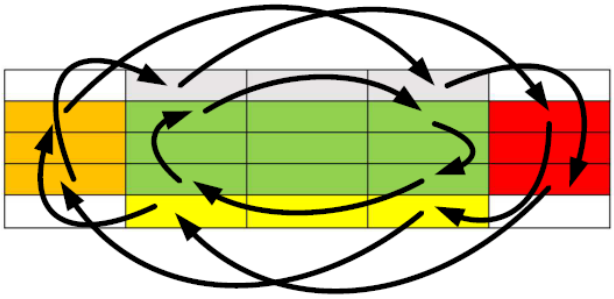
Pour finir, on initialise les voisins. Pour cela dans l'algorithme ci-dessous, on utilise la fonction « research_side » qui permet de trouver à partir de la position absolue d'une face, ainsi que la fonction « research_num » qui retourne le numéro du cubie selon la position relative de la face de son ou de ses voisins.

```
        //Initialisation des voisins.

        //Cette fonction permet d'attribuer à chaque cubie ses voisin.
        Fonction rubiks_neighbour(rubiks_side *rubiks) {

            // initialiser les voisins
            face, cubie : entiers

            // side relative : UP, DOWN, RIGHT, LEFT
            // side absolu : ORANGE, GREEN, BLUE...
            side_relative, side_abs : entiers

            Pour (face = WHITE; face <= YELLOW; face++) Faire {
                Pour (cubie = 0; cubie < 9; cubie++) Faire {
                    // si le cubie est de type edge, un seul voisin
                    Si (rubiks[face].cubie[cubie].type == EDGE) Alors {
                        //position relative de la face voisine, position absolu
                        // trouver à côté de quelle face (relative) le cubie est en fonction de ses coordonnées
                        // soit il a y = 0 (en haut)
                        Si (rubiks[face].cubie[cubie].y == 0) Alors {
                            side_relative = 0
                        }
                        // en bas
                        Sinon Si (rubiks[face].cubie[cubie].y == 2) Alors {
                            side_relative = 2

                        }
                            // au milieu : 2 possibilités
                        Sinon {
                            // il est a gauche
                            Si (rubiks[face].cubie[cubie].x == 0) Alors {
                                side_relative = 3
                            }
                                // il est a droite
                            Sinon {
                                side_relative = 1
                            }
                        }

                        // recuperer la side absolu
                        side_abs = rubiks[face].neighbour_side[side_relative]

                        // on met la side absolu comme side voisine du cubie
                        rubiks[face].cubie[cubie].neighbours[0].num_side = side_abs

                        // recuperer le num du cubie voisin
                        rubiks[face].cubie[cubie].neighbours[0].num_cubie = research_num(research_side(rubiks,face, side_abs), EDGE,0)

                        // mise a -1 du 2nd voisin, se chiffre indique qu'il n'y a pas de second voisuin (type edge)
                        rubiks[face].cubie[cubie].neighbours[1].num_side = -1
                        rubiks[face].cubie[cubie].neighbours[1].num_cubie = -1
                        // position initiale
                    }


            Si (rubiks[face].cubie[cubie].type == CORNER) Alors {

                // 2 voisins pour un corner
                side_relative2, side_abs2 : entiers

                // trouver à côté de quelles faces (relative) le cubie est en fonction de ses coordonnées
                Si (rubiks[face].cubie[cubie].y == 0) Alors {
                    Si (rubiks[face].cubie[cubie].x == 0) Alors {
                        side_relative = 3
                        side_relative2 = 0
                    } Sinon {
                        side_relative = 0
                        side_relative2 = 1
                    }
                }
                Sinon{
                    Si (rubiks[face].cubie[cubie].x == 0) Alors{
                        side_relative = 2
                        side_relative2 = 3
                    } Sinon {
                        side_relative = 1
                        side_relative2 = 2
                    }
                }

                // enregistrer le numéro absolu de ces faces
                side_abs = rubiks[face].neighbour_side[side_relative]
                side_abs2 = rubiks[face].neighbour_side[side_relative2]

                // On peut maintenant attribuer num_side (2 fois)
                rubiks[face].cubie[cubie].neighbours[0].num_side = side_abs
                rubiks[face].cubie[cubie].neighbours[1].num_side = side_abs2

                //On peut maintenant retrouver num_cubie grâce à research_side() et research_num()
                rubiks[face].cubie[cubie].neighbours[0].num_cubie = research_num(research_side(rubiks,face, side_abs), CORNER,research_side(rubiks,side_abs2, side_abs))
                rubiks[face].cubie[cubie].neighbours[1].num_cubie = research_num(research_side(rubiks,face, side_abs2), CORNER,research_side(rubiks,side_abs, side_abs2))
            }

            // dans le cas du centre on considère que son cubie voisin est le cubie opposé sur le rubiks cube
            // il est donc facile à définir et il n'y a pas de deuxième voisin
            Si (rubiks[face].cubie[cubie].type == CENTER) Alors S{

                rubiks[face].cubie[cubie].neighbours[0].num_side = rubiks[face].opposite_side
                rubiks[face].cubie[cubie].neighbours[0].num_cubie = 4
                // mise a -1 du 2nd voisin
                rubiks[face].cubie[cubie].neighbours[1].num_side = -1
                rubiks[face].cubie[cubie].neighbours[1].num_cubie = -1
            }
        }
    }
}
```

5.3 Réalisation d'un mouvement

Pour la fonction de mouvement, on modifie la couleur de chaque cubie et de ses voisins dans un ordre précis. Il est nécessaire d'enregistrer la couleur des premiers cubies "écrasés" pour pouvoir terminer le mouvement.

Ci-dessous la représentation de la rotation de la face verte en sens horaire le Rubik's cube étant résolu.



```
// Fonction de mouvements.

Fonction move_side_clockwise(rubiks_side *rubiks, int side, int add_to_history) {
    caratcère tab_face[][10] = {"BLANC", "ORANGE", "VERT", "ROUGE", "BLEU", "JAUNE"}

    tmp1, tmp2, tmp3 : entiers

    // On sauvegarde les données du coin 0 pour pouvoir les utiliser à la fin de l'algorithme, pour écraser le coin 2
    tmp1 = rubiks[side].cubie[0].color;
    tmp2 = rubiks[rubiks[side].cubie[0].neighbours[0].num_side].cubie[rubiks[side].cubie[0].neighbours[0].num_cubie].color
    tmp3 = rubiks[rubiks[side].cubie[0].neighbours[1].num_side].cubie[rubiks[side].cubie[0].neighbours[1].num_cubie].color

    // On commence par faire tourner les coins.
    move_corner(rubiks, side, 6, 0) // cette fonction ne sera pas présenté dans ce rapport pour des raisons de temps et de taille du rapport.
    move_corner(rubiks, side, 8, 6)
    move_corner(rubiks, side, 2, 8)

    // Et on finit par récupérer les données sauvegardées pour faire "tourner" le cubie[2]
    rubiks[side].cubie[2].color = tmp1
    rubiks[rubiks[side].cubie[2].neighbours[0].num_side].cubie[rubiks[side].cubie[2].neighbours[0].num_cubie].color = tmp2
    rubiks[rubiks[side].cubie[2].neighbours[1].num_side].cubie[rubiks[side].cubie[2].neighbours[1].num_cubie].color = tmp3

    // Maintenant, on fait tourner les arêtes
    // On sauvegarde les données de l'arête 1 pour pouvoir les utiliser à la fin de l'algorithme, pour écraser l'arête 5
    tmp1 = rubiks[side].cubie[1].color
    tmp2 = rubiks[rubiks[side].cubie[1].neighbours[0].num_side].cubie[rubiks[side].cubie[1].neighbours[0].num_cubie].color

    move_edge(rubiks, side, 3, 1) // cette fonction ne sera pas présenté dans ce rapport pour des raisons de temps et de taille du rapport.
    move_edge(rubiks, side, 7, 3)
    move_edge(rubiks, side, 5, 7)

    // Et enfin l'arête restante 5, deviens 0.
    rubiks[rubiks[side].cubie[5].neighbours[0].num_side].cubie[rubiks[side].cubie[5].neighbours[0].num_cubie].color = tmp2
    rubiks[side].cubie[5].color = tmp1

    // Et on ajoute le mouvement réalisé à l'historique des étapes, uniquement si ça nous a été demandé
    Si (add_to_history) Alors {
        Si (history == NULL) Alors {
            history = init_solution(tab_face[side])
        }
        Sinon {
            add_step_to_solution(history, tab_face[side])
        }
    }
}
```

Pour ce qui est du mouvement antihoraire, et afin de ne pas réaliser de duplication de code, on applique tout simplement trois fois la fonction « move_side_clockwise ».

5.4 La résolution du Rubik's cube

La résolution du Rubik's cube se base sur différents algorithmes de résolution indiqués en annexe. Un travail de recherche a été nécessaire pour trouver à chaque fois des algorithmes plus facilement adaptable à la programmation. En effet ils sont destinés à être résolus par des humains et certaines étapes d'identification de l'algorithme à réaliser peuvent se révéler complexes. De plus, certains algorithmes imposent de tourner le Rubik's cube et de changer de face "front", ce qui n'est pas possible dans notre cas, nous avons donc dû adapter ces algorithmes.

Pour limiter le nombre de cas de figures à traiter, les algorithmes les plus polyvalents sont utilisés, ainsi ils conduisent souvent à la réalisation de mouvements inutiles ou inefficaces. Par exemple, même dans le cas où le Rubik's cube est déjà résolu, l'algorithme réalisera de nombreux mouvements.

Pour permettre au programme de restituer la liste des mouvements réalisés, ceux-ci sont comptabilisés au fur et à mesure par les fonctions de mouvements, si elle en reçoit la consigne en paramètre. La liste des mouvements est réalisée dynamiquement sous la forme d'une liste chaînée qui s'agrandit au fur et à mesure des étapes réalisées.

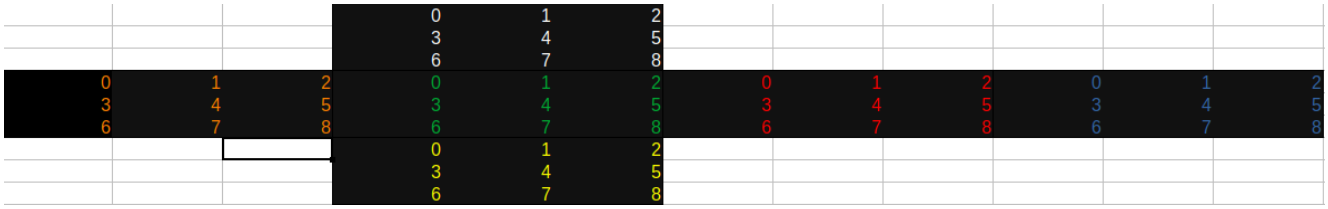
Enfin, les étapes de résolution sont affichées en console avant le Rubik's cube sous la forme d'une couleur indiquant la face à tourner et d'un " ' " dans le cas d'une rotation antihoraire.

5.5 L'affichage

Le programme d'affichage d'un Rubik's cube est simplement constitué de différents "printf()" affichant le Rubik's cube. Etant donné les contraintes d'affichage en deux dimensions du Rubik's cube, nous avons écarté l'utilisation de boucles particulières.

```
--------------
Faites votre choix (attention vous ne pouvez saisir que des chiffres).
1-> Mélanger le rubiks.
2-> Résoudre le rubiks.
3-> Faire vos propre mouvements.
4-> Placer les couleurs.
5-> Afficher le rubiks.
6-> Quitter le programme.
Rentrez votre choix: 6
------------------------------

--------------
    WWW
    WWW
    WWW
OOO GGG RRR BBB
OOO GGG RRR BBB
OOO GGG RRR BBB
    YYY
    YYY
    YYY
--------------
```

Le cube est représenté à plat, sur la base de la référence suivante, que nous avons réalisé sous Excel:



Les faces sont référencées selon leur couleur et leur position. La face blanche est la face "0" (en haut), la face orange est la première à gauche dans la partie "centrale" du cube, sa référence est "1", et ainsi de suite jusqu'à la face jaune qui est à l'opposé de la face blanche, donc tout en bas, et sa référence est donc "6".

Comme on peut le voir sur ce même schéma, chaque face possède 9 cubies, qui ont leur propre référence de 0 à 9.
Par conséquent, on peut référencer un cubie à partir de sa face, et son numéro sur la face, comme des coordonnées (face, numéro). Cela nous permet de référencer facilement les différentes pièces de notre cube afin de les traiter mais aussi pour nous permettre de les afficher simplement, y compris si l'on développe les affichages à travers nCurses ou d'autres solutions graphiques.

Pour afficher la lettre correspondant à la couleur de la face, une liste de référence est établie pour récupérer le caractère correspondant au numéro de la couleur. L'affichage est colorisé en utilisant des caractères ANSI et sont prévus pour des consoles pouvant afficher plusieurs centaines de couleurs, ce qui n'est pas le cas de toutes (cela fonctionne bien sous Linux, KDE ou Gnome, mais nous ne pouvons pas être certains que cela puisse fonctionner sur tous les systèmes d'exploitation). Chaque caractère possède donc son propre code couleur basé sur la référence : https://en.wikipedia.org/wiki/ANSI_escape_code.

5.6 Les menus

L'affichage de l'interface a déjà été décrit, mais nous n'avons pas encore abordé la sécurité de la saisie. En effet, nous avons fait le choix de ne pas utiliser la fonction scanf car, bien que pratique, n'est pas du tout sécurisée (en fonction de ce qui est saisi par l'utilisateur, nous pouvons faire planter le programme et la gestion des erreurs aurait été plus compliquée). Après des recherches sur des solutions alternatives, nous avons plutôt décidé de nous orienter vers la fonction getchar, car elle permet une saisie plus maitrisée, caractère par caractère.

En effet, de par son fonctionnement, la fonction getchar nous permet de régler le problème d'un utilisateur saisissant par erreur des caractères, non prévus, lorsqu'on lui demande de rentrer uniquement des entiers. Par contre, getchar ne renvoie pas directement les caractères saisis mais leur code ASCII. Cela oblige à réaliser des conversions, mais cela nous permet de valider les entrées des utilisateurs plus facilement sans faire planter le programme (comme par exemple, un scanf qui prévoit de recevoir un entier, alors que l'utilisateur saisi une chaîne).

De plus, il nous a suffi de coder une fonction très simple vérifiant que les choix de l'utilisateurs étaient bien autorisés.

6 Les résultats obtenus

Toutes les fonctionnalités du programme ont été testés au fur et à mesure de leur réalisation. En le parcourant à de nombreuses reprises, toutes les erreurs identifiées ont été corrigées. Nous avons utilisé le système de débogage de Clion notamment pour nous aider à comprendre les différentes erreurs de référence que nous avions pu faire au fur et à mesure de nos développements.

Finalement, le programme fonctionne comme prévu, mais n'utilise pas de bibliothèque particulière pour afficher les menus et les résultats. Cette partie est basique, mais elle a permis de valider le fonctionnement des différents algorithmes assez facilement.

7 Difficultés rencontrées

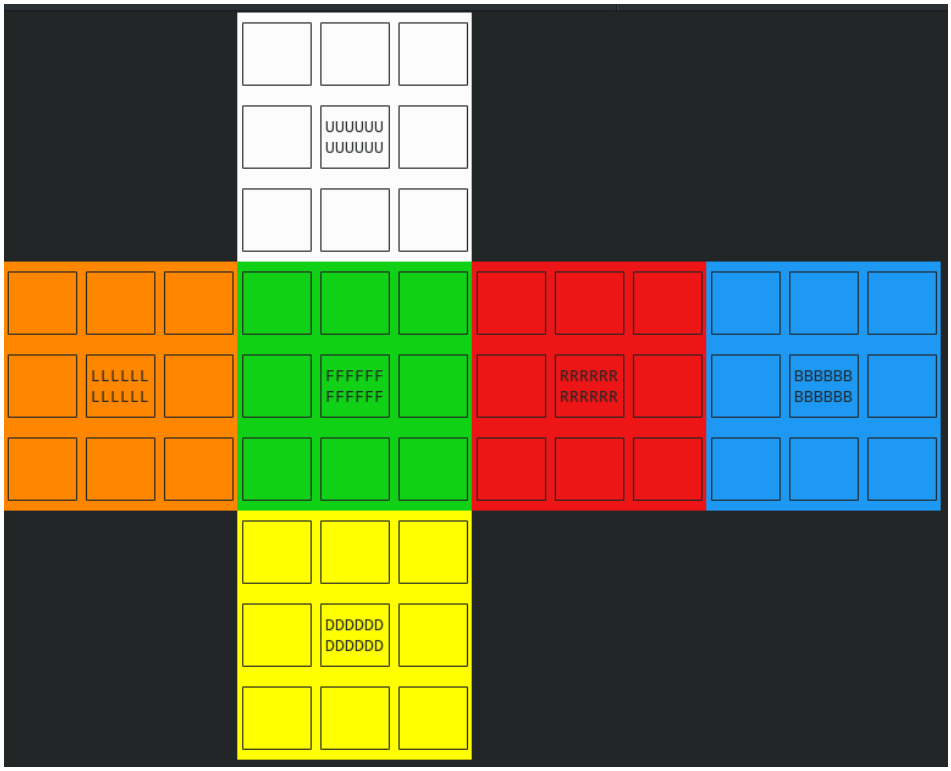
Lors de ce projet, la principale difficulté que nous avons rencontrée fut la complexité du code en lui-même. En effet, le développement et le travail de réflexion sur les différents choix et possibilité que nous offraient le sujet de ce projet étaient important. Il nous a fallu beaucoup de réunions en présentiel ou en distanciel pour parvenir à trouver des options qui nous convenaient à tous les deux. C'est probablement la structure représentant le Rubik's cube qui nous a pris le plus de temps, mais cela était essentiel, car de cette structure dépendait tout le reste du programme.

Même une fois ces idées trouvées, le travail était loin d'être encore fini. En effet, le développement en lui-même fut assez compliqué, de par les nouvelles problématiques soumises pas le sujet, telles que la représentation d'un Rubik's cube dans un code en c, comment remplir cette représentation, comment s'en servir pour la suite, etc, mais aussi parce que nous étions novices sur ce langage. Le langage C n'est pas d'un abord aussi "facile" que Python, et le code peut devenir très vite assez touffu. Nous avons retravaillé à plusieurs reprises les commentaires, pour que nous puissions plus facilement nous y retrouver.

Mais heureusement, le travail de groupe ne fut pas un souci pour ce projet, nous avons commencé à travailler dès que nous avons reçus de sujet et nous avons travaillé régulièrement car nous désirions tous les deux obtenir un résultat qui nous satisfasse. Il nous semble que notre travail sur ce projet a été beaucoup plus efficace que sur les autres projets réalisés jusqu'à maintenant. Nous nous sommes facilement accordés, nous avons travaillé de concert et toujours en accord. Nous avons utilisé plusieurs outils (Github, Clion, et Discord) qui nous ont bien aidé à travailler ensemble lorsque nous ne pouvions pas travailler en présentiel.

8 Améliorations possibles

Concernant les améliorations possibles, la plus évidente serait la mise en place d'un menu et de l'affichage du Rubik's cube avec la bibliothèque nCurses ou bien d'autres solutions dont nous n'avons pas encore idée. Nous avions testé nCurses, et nous avions une idée assez précise de la manière de réaliser l'affichage du cube, mais il nous aurait manqué du temps et les fonctionnalités de menus.



Nous pourrions aussi revoir la façon dont nous avons modélisé la gestion des cubies et de leurs voisins. Par exemple, nous aurions pu utiliser des listes chainées qui auraient peut-être simplifié certaines autres fonctions. Mais au début du projet, nous n'avions pas abordé ce sujet, nous ne l'avons utilisé que plus tard pour gérer les étapes de résolution d'un Rubik's cube. Il serait aussi possible de revoir les algorithmes de résolution. Maintenant que toutes les briques sont présentes, il pourrait donc être assez facile de faire évoluer ce projet vers quelque chose de plus "joli", ou de plus performant. Par ailleurs, nous avons pu constater qu'il est possible de rentrer dans une boucle infinie lorsqu'un Rubik's cube ne peut pas être résolu. Dans ce cas, il faudrait utiliser une variable comptant le nombre d'itérations pour s'arrêter au bout d'un nombre prédéfini. Malheureusement, cela sous-entend une réécriture de la plupart des fonctions afin de véhiculer cette information et arrêter différentes boucles. Cela n'est probablement pas la solution la plus propre, ni la plus efficace mais c'est celle qui nous demanderait le moins de travail. L'autre solution serait de se baser sur la liste chaînée des étapes de résolution afin de les compter et de s'arrêter. Mais cela ne change rien au fait que nous serions obligés de réécrire la plupart des fonctions, sans compter le fait que cette liste n'est créée que lorsqu'on lance la fonction de résolution et pas dans tous les autres cas.

9 Conclusion

Pour conclure, ce projet s'est très bien déroulé malgré la quantité de travail, le degré d'investissement demandé et la nécessité de travailler à deux et pas plus. Nous avons surmonté ces difficultés grâce à une commune entente, des outils tels que « code with me » de Clion, Discord et son partage d'écran ou encore Github, mais aussi grâce au fait que nous utilisions tous les deux Linux comme système d'exploitation (ce qui évite pas mal de problèmes tout de même). Ainsi, nous avons pu aboutir à un résultat satisfaisant de notre point de vue, et respectant le cahier des charges. Notre programme remplit toutes les exigences, bien qu'il mérite encore quelques améliorations tels que la révision de la mise en place des voisins en utilisant les listes chainées, ou encore l'utilisation de la bibliothèque « ncurses » pour les menus et l'affichage du Rubik's cube. Enfin, la réalisation de notre premier projet en C nous a permis de mettre en perspective les apprentissages de ce semestre et relever les défis qui se sont présentés à nous.

ANNEXES

Voir fichier "Doc/Rapport programme rubiks - Annexes - Mathieu CHANTOT - Clément LE STRAT.pdf"