Compte rendu projet développement

Rubik's Cube

Responsable du projet :
M.Carl Esswein

Étudiants :
Djawad M’DALLAH MARI / Nathan DENZLER

Sommaire

Introduction **3**

Cahier des charges **3**

Problème 3

Le besoin 3

Les contraintes 3

Conduite de projet **4**

Planning prévisionnel 4

Planning réel 4

Comptes rendus de réunion **5**

Mise en oeuvre **5**

Structure du projet 6

La structure de donnée mise en oeuvre 6

Les principales fonctions 8

Problèmes rencontrés 12

Tests **15**

Test des fonctions élémentaires 15

Test complet des coups 16

Test de mélange et de résolution 17

Bilan **17**

Introduction

Dans le cadre de notre cursus en informatique industrielle à Polytech Tours, nous sommes amenés à réaliser un projet de développement durant la première année. Ce rapport a pour but de présenter notre projet. Nous verrons dans un premier temps le cahier des charges, puis nous aborderons par la suite comment on a fait pour gérer toute la partie organisationnel du projet. Ensuite nous regarderons quelques solutions envisagés pour répondre au besoin, notamment la structure de donnée employé et les principales fonctions. Enfin, nous verrons quelques programmes de test et nous terminerons par un bilan de fin de projet.

Cahier des charges

Dans cette partie nous allons voir les éléments du cahier des charges établi en début de projet. Nous présenterons dans cette partie le problème initial, le besoin explicite du client ainsi que ses attentes.

Problème

Ici la problématique du projet se base sur le casse-tête très connu aujourd'hui appelé le Rubik's Cube. Ce casse-tête est apparu en 1974, inventé par l'architecte et professeur de design M.Erno Rubik. Elle s'est rapidement répandu sur toute la planète dès les années 80.

Le principe est simple, réussir à remettre chaques éléments du cube à sa place. Le cube ayant 6 faces et chaque faces étant divisée en 9 cubes miniatures, la tâche devient très vite complexe.

Le cube ayant des caractéristiques assez complexe (6 faces, 8 cubes de coin à 3 faces visible, 12 cubes d'arête à 2 faces visibles…) , l'idée ici est de développer un système similaire informatique, permettant de simuler les mêmes comportement que le Rubik's Cube.

Le besoin

La demande du client est de pouvoir mettre en place une structure de donné solide afin de préserver toutes les caractéristiques et contraintes imposées par le Rubik's Cube. En effet l'implémentation d'un tel système nécessite une grande réflexion au niveau de la structure de donnée employé. Celle-ci devrait permettre de réaliser les mouvements permis par le casse-tête tout en interdisant d'autres mouvements. Un grand travail de conception et de réflexion sur la structure de donné est donc la principale attente du client. La partie graphique de notre cube n'est pas demandé. (BONUS)

Les contraintes

Concernant le langage de programmation à utiliser, le client n'impose aucune contrainte. Néanmoins, il serait plus intéressant de partir sur un langage de programmation puissant et assez bas niveau comme le langage C afin de pouvoir contrôler chaque aspect

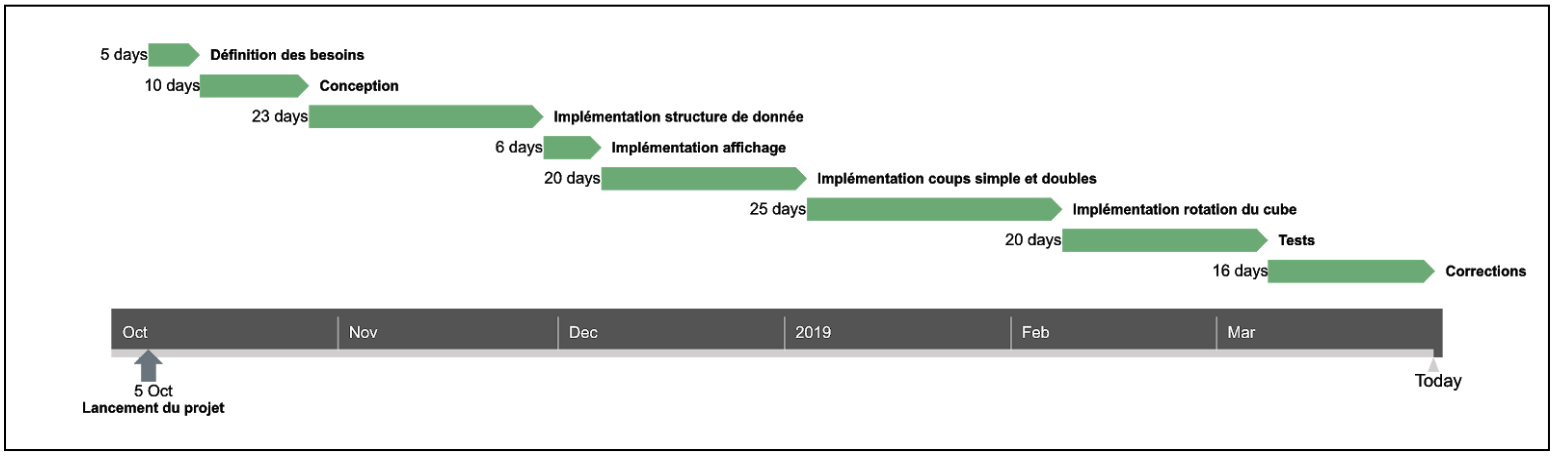
du programme et ainsi limiter l'utilisation de fonctions prédéfinis. Ceci rendrait le projet moins enrichissant pour notre apprentissage. Nous avons utilisé le langage C.

Conduite de projet

Dans cette partie nous allons voir les aspects de conduite de projet mise en place à travers notre planning prévisionnel. Nous comparerons par la suite le planning prévisionnel avec le planning réel.

Planning prévisionnel

Quelques jours après le lancement du projet au mois d'octobre, nous avons établi le planning prévisionnel ci-dessous:

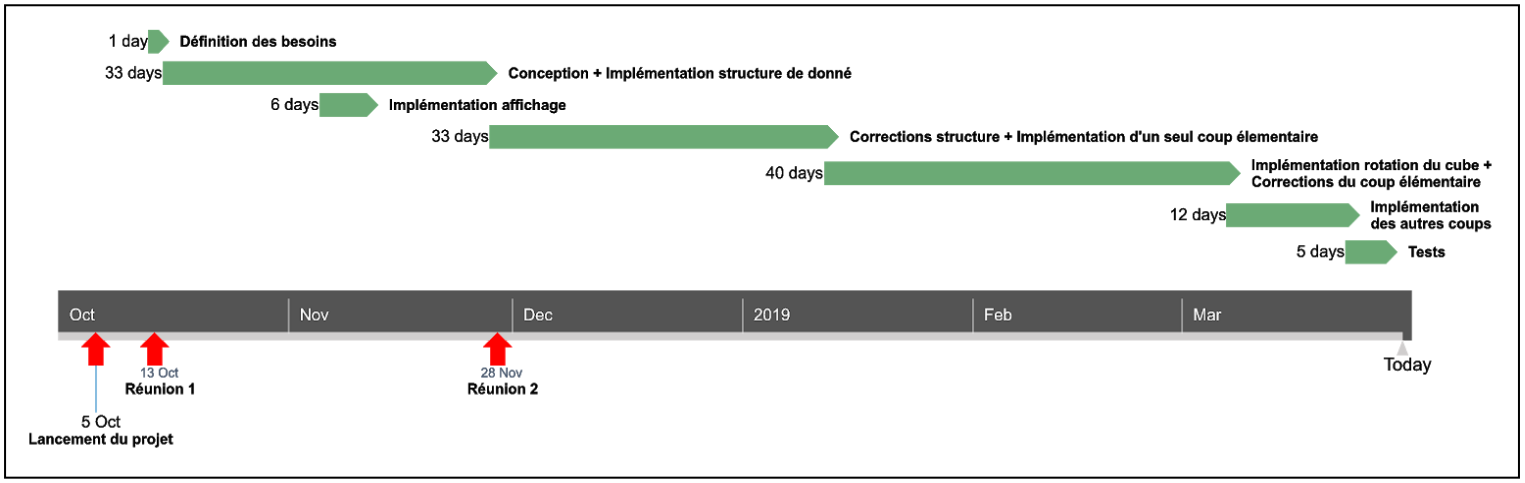


Nous avons tout d'abord identifier les potentielles tâches qui devront être réalisées tout en étant conscient que d'autres tâches peuvent se rajouter. Nous avons donc volontairement accordé beaucoup plus de temps pour les tâches d'implémentation que pour les autres types de tâche. En effet par exemple pour la définition des besoins du client, nous avions estimé que 5 jours étaient suffisante pour cette tâche. En revanche pour l'implémentation des fonctions importantes tels que la rotation du cube ou encore le codage des mouvements, nous avions octroyé beaucoup plus de temps. Nous avions également prévu au moins 15 jours de marge avant la fin du projet. Ces 15 jours nous permettrait de terminer les éventuelles tâches inachevées ou corriger les bugs éventuelles. Durant ces derniers jours nous avions prévu également de préparer notre soutenance de projet ainsi que le compte rendu.

Planning réel

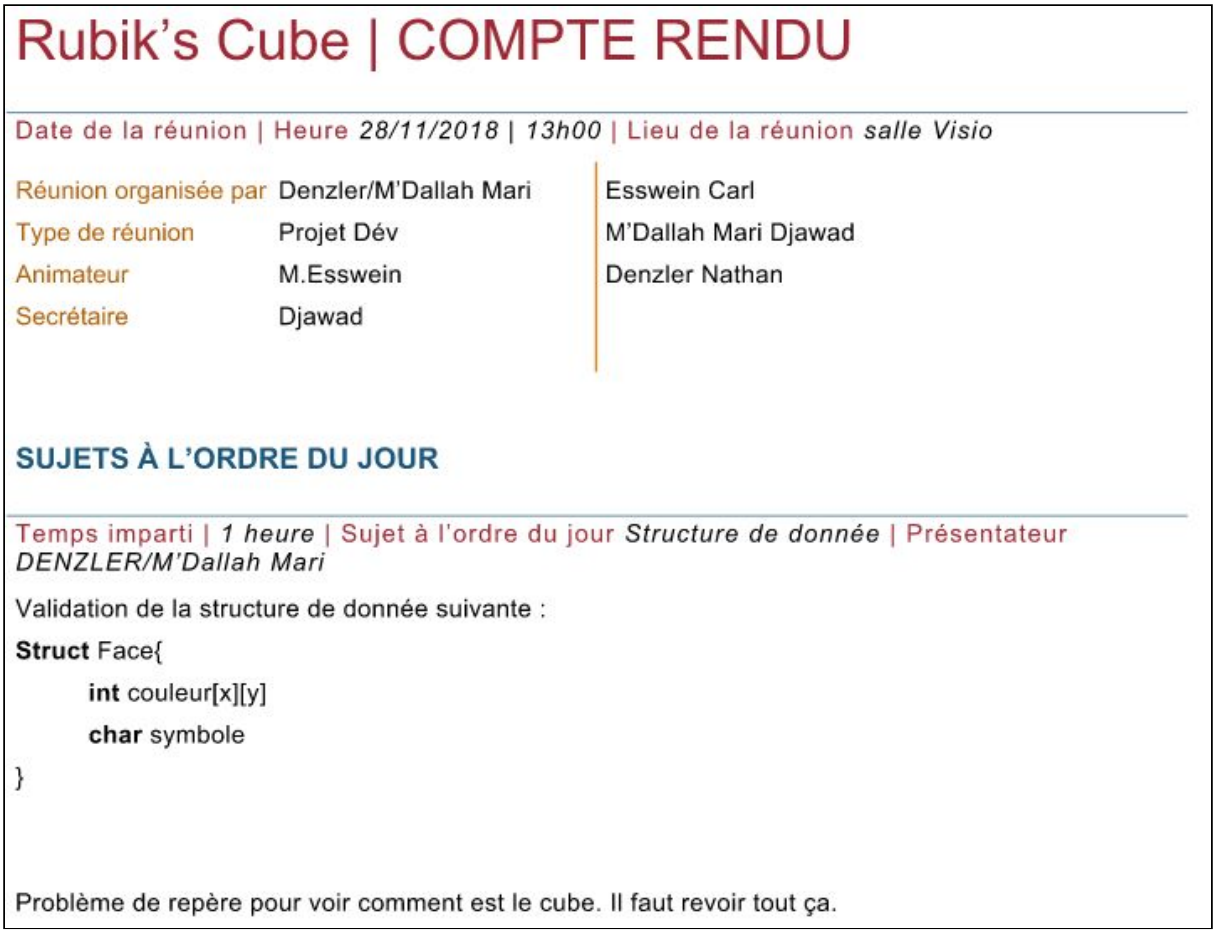
Il est évidemment impossible de respecter toutes les prévisions planifiées en début de projet. Une prévision au préalable permet néanmoins d'anticiper les grandes phases d'un projet et ainsi prévenir les problèmes d'organisation. Dans notre cas le déroulement du projet ne s'est pas effectuée à 100% tel que prévu. On a par exemple utilisé beaucoup plus

de temps pour faire certaines tâches et à l'inverse effectué certaines tâches en quelques jours. Ci-dessous le planning réel de notre projet.



Comme on peut le voir ici, certaines tâches ont été implémenté en même temps que d'autres. Par exemple lors de l'implémentation de la structure de donnée, le besoin de pouvoir afficher le cube était imminent. On a donc réalisé ces deux tâches en même temps. Également comme évoqué plus haut, on voit clairement ici que la phase de test de notre programme a été réduit de 10 jours. En effet l'implémentation des tests nous ont pas fait consommer beaucoup de temps. Ce qui nous a permit également de faires des corrections et des améliorations avant la deadline.

Comptes rendus de réunion



Rubik's Cube | COMPTE RENDU

Date de la réunion | Heure *28/11/2018* | *13h00* | Lieu de la réunion *salle Visio*

Réunion organisée par	Denzler/M'Dallah Mari	Esswein Carl
Type de réunion	Projet Dév	M'Dallah Mari Djawad
Animateur	M.Esswein	Denzler Nathan
Secrétaire	Djawad	

SUJETS À L'ORDRE DU JOUR

Temps imparti | *1 heure* | Sujet à l'ordre du jour *Structure de donnée* | Présentateur *DENZLER/M'Dallah Mari*

Validation de la structure de donnée suivante :

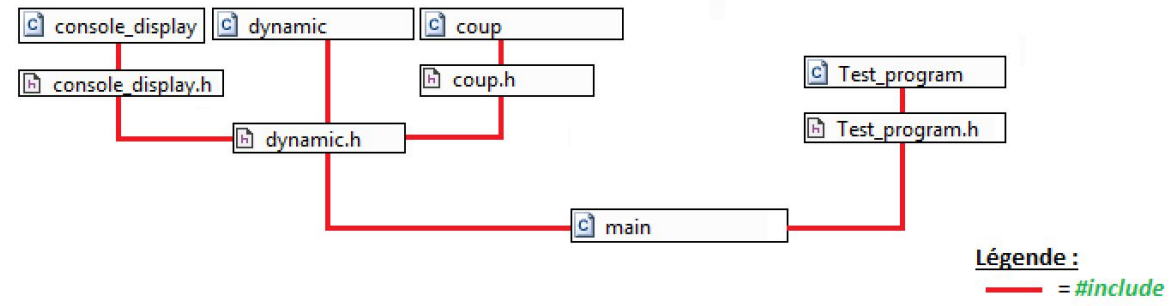
```
Struct Face{
        int couleur[x][y]
        char symbole
}
```

Problème de repère pour voir comment est le cube. Il faut revoir tout ça.

Mise en oeuvre

Cette section traite la partie technique du projet. On a sélectionné que les éléments importants qui méritent d'être expliqués afin de ne pas trop chargé ce compte rendu. Nous avons donc décidé de montrer tout d'abord la structure de notre projet afin de bien comprendre ce qui suit après. Ensuite nous vous aborderons la question de la structure de donnée mise en oeuvre. Nous mettrons par la suite en évidence les fonctions qu'on estime les plus importantes dans notre programme. Et enfin nous aborderons quelques problèmes qu'on a rencontré au cours de ce projet.

Structure du projet



Le diagramme ci-dessus illustre toutes les fichiers présent dans notre projet. Chaque fichiers sert à une/des tâche(s) spécifique(s).

Test_program : Contient des fonctions permettant de tester la robustesse du programme rubik's cube.

Dynamic : Contient la structure de donnée du Cube, les fonctions essentielles au fonctionnement du cube ainsi que les fonctions élémentaires tel que les mouvements de base.(Rotation du cube et un coup élémentaire Right())

Console_display : Contient les fonctions permettant l'affichage console du cube. (Peut être changé avec des librairies graphiques)

Coup : Contient tous les coups possible à une main sur un cube. Exemple coup droit, coup gauche...

Main : Permet de lancer les deux parties du programme (partie Programme rubik's cube et partie test du programme)

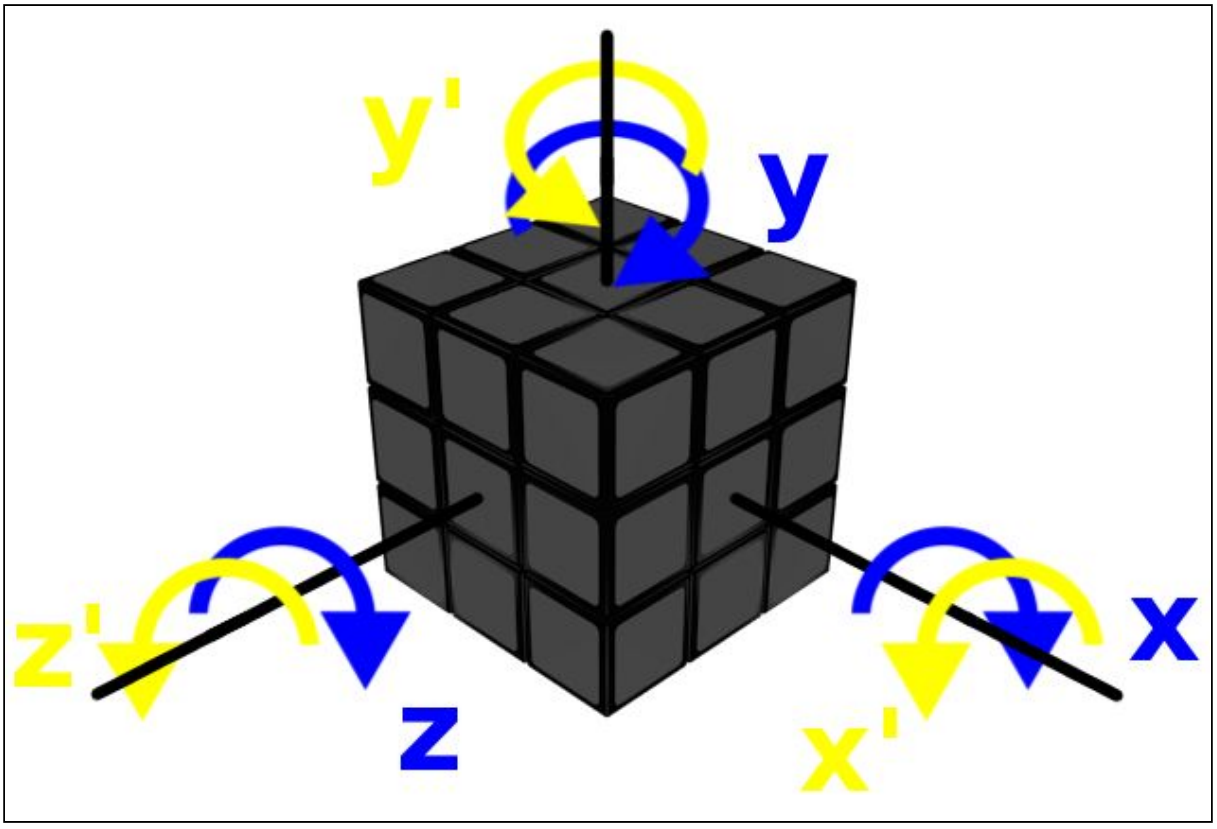
La structure de donnée mise en oeuvre

Avant toute chose il faut réussir à définir une structure de donnée qui reprend les caractéristiques du Rubik's Cube. Ceci peut très vite devenir assez complexe et la façon dont on défini notre structure de donnée va par la suite impacté la manière dont on programme ses comportements. Durant la phase de conception, on a réfléchi à plusieurs façons d'implémenter le cube. La solution qu'on a retenu au final n'est peut-être pas la plus optimale, mais est suffisante pour simuler le Rubik's Cube. Ci-dessous l'implémentation.

```
struct Face{
	int color[size][size]; //position sur la face d'un élément
	char symbol;
	int numFace;

};

struct Cube{ //Here structure of the cube
	struct Face face[nbface]; //For display

	struct Face FrontFace;
	struct Face TopFace;
	struct Face BackFace;
	struct Face RightFace;
	struct Face LeftFace;
	struct Face DownFace;
	struct Face BufferFace;
};

struct Cube cube; // Create Structure Cube
```

Nous avons décidés de séparer le cube en une double structures une définissant un cube et l'autre définissant les faces. Celle définissant les faces se compose ainsi : Il y a 6 faces pour la position dans l'espace et un tableau de face pour gérer l'affichage et l'affectation rapide des matrices grâce à des boucles. On instancie ensuite le cube.

Concernant la gestion des rotations du cube en entier, nous avons défini les axes de rotations comme suit :



Les principales fonctions

Ici nous mettons en évidence les principales fonctions de notre programme. Durant la phase de conception, nous avons pu remarquer qu'en réalité, il suffisait d'implémenter que quelques fonctions de base et puis par la suite les autres fonctions seront implémenter à partir de ces bases là. Par exemple, la fonction de rotation pour le mouvement de droite (présenté plus bas) va servir à implémenter tous les autres fonctions de ce type. Il suffira de refaire le même mouvement et y ajouter des changements de point de vue.

Initialisation du cube

Une des premières fonctions à être appelées dans notre programme est la fonction qui permet d'initialiser le cube. Ci-dessous l'implémentation de cette fonction.

```
void initialize_cube()
{
      //Affectation de couleur à chaque face
      cube.face[WHITE].symbol = 'W' ;
      cube.face[GREEN].symbol = 'G' ;
      cube.face[RED].symbol = 'R' ;
      cube.face[BLUE].symbol = 'B' ;
      cube.face[ORANGE].symbol = 'O' ;
      cube.face[YELLOW].symbol = 'Y' ;
      cube.face[BUFFER].symbol = 'X' ; //secure, permit to correct error
```

```
      //Affection à chaque vue (front, top,..etc) une face
      cube.FrontFace.numFace = RED ; // Define RED = 3
      cube.TopFace.numFace = WHITE ; // Define WHITE = 1
      cube.BackFace.numFace = ORANGE ; // Define ORANGE = 5
      cube.RightFace.numFace = BLUE ; // Define BLUE = 4
      cube.LeftFace.numFace = GREEN; // Define GREEN = 2
      cube.DownFace.numFace = YELLOW ; // Define YELLOW = 6
      cube.BufferFace.numFace = 0; //// Define BUFFER = 0

      fill_face(cube.FrontFace.numFace);
      fill_face(cube.TopFace.numFace);
      fill_face(cube.BackFace.numFace);
      fill_face(cube.RightFace.numFace);
      fill_face(cube.LeftFace.numFace);
      fill_face(cube.DownFace.numFace);
}
```

Dans le code ci-dessus on affecte les couleurs sous forme de symbole et également dans les faces, on a affecté les faces au démarrage pour que le cube ait une position de base dans l'espace. Celles-ci seront manipulées par la suite pour les rotations des faces. Puis on utilise la fonction de remplissage.

Remplissage du cube

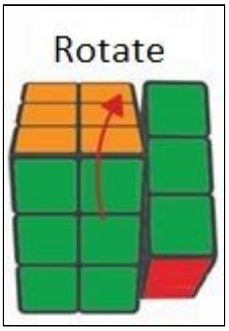
```
void fill_face(int Fce) // Create face of the cube and fill them with
color
{
      for(int i = 0; i < size; i++)
      {
            for(int j = 0; j < size; j++)
            {
                  cube.face[Fce].color[i][j] = Fce ;

            }
      }
};
```

C'est tout simplement une double boucle qui remplis la face que l'on appel. On remplit par rapport à la couleur de la face.

Le mouvement élémentaire de base

Ici on a le mouvement élémentaire qui va être la base de tous les autres mouvements. En effet à travers un seul mouvement de rotation simple (voir figure

ci-dessous) on arrive à obtenir tous les autres mouvements par combinaison de celle-ci avec des changements de point de vue.



Ci-dessous l'implémentation de ce mouvement.

```
void rotates() //Movement right()
{

	int FrontV=cube.FrontFace.numFace; //COPY of num face in int
	int TopV=cube.TopFace.numFace;
	int BackV=cube.BackFace.numFace;
	int RightV=cube.RightFace.numFace;
	int LeftV=cube.LeftFace.numFace;
	int DownV=cube.DownFace.numFace;

	int j=2;
	for(int i = 0 ; i<size ; i++)
	{
	cube.face[BUFFER].color[i][2] = cube.face[FrontV].color[i][2] ;
	}
	for(int i = 0 ; i<size ; i++)
	{
	cube.face[FrontV].color[i][2] = cube.face[DownV].color[i][2] ;
	}
	for(int i = 0 ; i<size ; i++)
	{
	cube.face[DownV].color[j][2] = cube.face[BackV].color[i][0] ;//inv
	j--;
	}
	j=2;
	for(int i = 0 ; i<size ; i++)
	{
	cube.face[BackV].color[i][0] = cube.face[TopV].color[j][2] ;//inv
	j--;
	}
	for(int i = 0 ; i<size ; i++)
	{
	cube.face[TopV].color[i][2] = cube.face[BUFFER].color[i][2] ;
```

```
        }

        butter_effect_rotate(RightV);
}
```

Nous commençons par récupérer dans la structure l'emplacement de chaque face pour pouvoir affecter les bonnes faces. C'est à dire front/top/ back s'affiche dans l'autre sens / down.

Les changements de point de vue Rx() / Ry() / Rz()

```
void ry()
{
        cube.BufferFace.numFace=cube.FrontFace.numFace;
        cube.FrontFace.numFace=cube.RightFace.numFace;
        cube.RightFace.numFace=cube.BackFace.numFace;

        cube.BackFace.numFace=cube.LeftFace.numFace;
        cube.LeftFace.numFace=cube.BufferFace.numFace;


effect_butter_opposite(cube.TopFace.numFace,cube.DownFace.numFace);

}
```

Le principe est le même c'est des échanges de face grâce au buffer. Elle sont rentré directement dans la structure du cube. Ce qui est bien pris en compte lorsque l'on utilise les autres fonctions qui nécessite la position du cube celle-ci est renseigné dans la structure même du cube.

Tous les mouvements

Il nous suffisait maintenant pour coder les coups d'utiliser sous forme de combinaisons les fonctions qui suivent : rotates(), rx(), ry() et rz()

Résumé sous forme d'un tableau :

Coup	Combinaison
void right()	rotates()
void left()	ry() + ry() + right() + ry() + ry()
void top()	rz() + right() + rz() + rz()+ rz()
void down()	rz() + rz() + rz() + right() + rz()

void front()	ry() + ry() + ry() + right() + ry()
void back()	ry() + right() + ry() + ry() + ry()
void double_right()	left() + rx() + rx() + rx()
void double_left()	right() + rx()
void double_top()	down() + ry()
void double_down()	top() + ry() + ry() + ry()
void double_front()	back() + rz()
void double_back()	front() + rz() + rz() + rz()
void middle_x()	top() + down_inv()
void middle_y()	right() + left_inv()
void middle_z()	front() + back_inv()

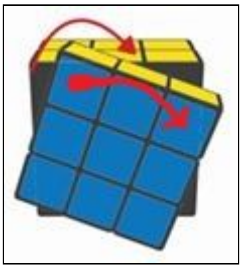
Ici n'apparaisse pas les coups inverse, mais par définition un coup inverse c'est 3 fois un coup normal.

Problèmes rencontrés

Dans la présentation des fonctions principales plus haut. Certaines fonctions ont dû être modifié suite à quelques problèmes qu'on a rencontrés. Dans cette partie nous allons voir quels sont ces problèmes et quels ont été les solutions trouvées pour y remédier.

Dans la fonction rotates

C'est ici que nous avons rencontré notre première difficulté. Nous avions pas anticipé l'effet que le coup avait sur la face droite. Ce qui explique qu'à la fin de l'implémentation de la fonction on appelle une autre fonction `butter_effect_rotate(RightV);` qui vient corriger ce problème. Illustration du problème ci-dessous:



Ci-dessous l'implémentation de la fonction correctives.

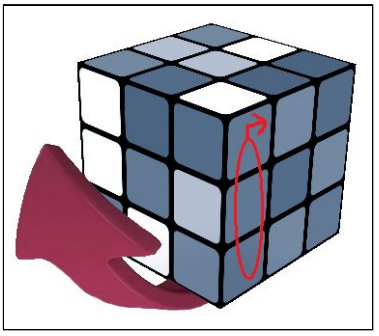
```
void butter_effect_rotate(int R)
{
      for(int i=0;i<size;i++){
```

```
            for(int j=0;j<size;j++)
            {
               cube.face[BUFFER].color[i][j] = cube.face[R].color[i][j] ;
            }
      }

      int j= 2 ;
      for(int i=0;i<size;i++){
      cube.face[R].color[0][i] = cube.face[BUFFER].color[j][0] ;
      cube.face[R].color[j][0] = cube.face[BUFFER].color[2][j] ;
      cube.face[R].color[2][j] = cube.face[BUFFER].color[i][2] ;
      cube.face[R].color[i][2] = cube.face[BUFFER].color[0][i] ;
      j--;
      }
}
```

Création d'une fonction pour un effet logique mais pas pensé tout de suite.
On copie la face concernée dans un buffer et il nous reste plus qu'à manipuler les élements pour les placer au bon endroit.

Dans les fonctions de rotations du cube (changement de vue)

Le problème ici s'effectue lors d'un changement de vue quel qu'en soit l'axe (x, y ou z). Dans le cas d'une rotation sur l'axe X par exemple, le front passera en top, le top en back, le back en down et le down en front. Cependant nous avions oublié que cette rotation impactait également les côtés qui restent à leurs places (dans notre exemple right et left). Illustration du problème ci-dessous.



- Les faces étaient toutes bonnes seulement, certaines étaient mal orientées.
  Il y avait également un effet que l'on a appelé effet papillon sur les deux faces qui s'opposaient.

Pour corriger nous avons créé 2 fonctions pour faire les réglages d'orientation. Ces fonctions tournent à 90° la face ou 180°. Sur le principe c'est le même code que pour l'effet papillon.

Pour l'effet papillon sur les faces opposées on a créé :

```
void ninety(int Fce)
{
   for(int i=0;i<size;i++){
            for(int j=0;j<size;j++){
                  cube.face[BUFFER].color[i][j]                          =
cube.face[Fce].color[i][j] ;
            }
      }

      int j= 2 ;
      for(int i=0;i<size;i++){
            cube.face[Fce].color[0][i] = cube.face[BUFFER].color[j][0] ;
            cube.face[Fce].color[j][0] = cube.face[BUFFER].color[2][j] ;
            cube.face[Fce].color[2][j] = cube.face[BUFFER].color[i][2] ;
            cube.face[Fce].color[i][2] = cube.face[BUFFER].color[0][i] ;
            j--;
      }
}
```

```
void effect_butter_opposite(int positive_sense,int
negative_sense)//effect to rotate opposite face
{
///1st Face positive
 for(int i=0;i<size;i++){
      for(int j=0;j<size;j++){
cube.face[BUFFER].color[i][j]=cube.face[positive_sense].color[i][j] ;
      }
 }
      int j= 2 ;
      for(int i=0;i<size;i++){ //repartition comme il faut
      cube.face[positive_sense].color[0][i] =
cube.face[BUFFER].color[j][0] ;
      cube.face[positive_sense].color[j][0] =
cube.face[BUFFER].color[2][j] ;
      cube.face[positive_sense].color[2][j] =
cube.face[BUFFER].color[i][2] ;
      cube.face[positive_sense].color[i][2] =
cube.face[BUFFER].color[0][i] ;
      j--;
      }

      ///2nd Face negative
      for(int i=0;i<size;i++){
```

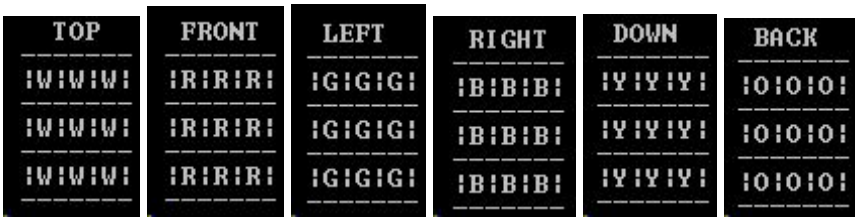
```
      for(int j=0;j<size;j++){
            cube.face[BUFFER].color[i][j] =
cube.face[negative_sense].color[i][j] ;
      }
      }
      j= 2 ;
      for(int i=0;i<size;i++){ //repartition comme il faut
      cube.face[negative_sense].color[0][j] =
cube.face[BUFFER].color[j][2] ;
      cube.face[negative_sense].color[j][2] =
cube.face[BUFFER].color[2][i] ;
      cube.face[negative_sense].color[2][i] =
cube.face[BUFFER].color[i][0] ;
      cube.face[negative_sense].color[i][0] =
cube.face[BUFFER].color[0][j] ;
      j--;
      }
}
```

Tests

Le programme de test compte 55 points de vérification dans nos test :

- le premier est la position du cube dans l'espace.
- Les 54 autres sont les 54 éléments du cube (6 faces*9 éléments).

Nous partons d'un cube fini et nous faisons tous les test 1 par 1 avec le même cube qui n'est pas formaté après chaque tests.
Tous les tests se feront à la suite dans une position précise comme ci-dessous dans l'espace. Puis nous testerons entre 4 000 et 400 000 fois chaque fonctions avec la même programme de test. Nous sommes conscients qu'à la moindre erreur, cela impactera les autres fonctions.
Après avoir utilisé un multiple de 4 sur toutes nos fonctions de "coup", nous devons retomber sur un cube fini et dans la position initiale. Nous verrons un test qui mélange le cube 400 000 et nous arriverons à le reconstruire avec les méthodes inverses.



Test des fonctions élémentaires

```
int verif_state()// FINISH CUBE
{
	int verif=0;
	if ((cube.FrontFace.numFace == RED) &
	(cube.TopFace.numFace == WHITE) &
	(cube.BackFace.numFace == ORANGE) &
	(cube.RightFace.numFace == BLUE) &
	(cube.LeftFace.numFace == GREEN) &
	(cube.DownFace.numFace == YELLOW) ){
	verif++;
	}//OK FONCTIONNE

	for(int F =1 ; F<nbface ; F++)
	{
	for(int i = 0; i < size; i++)
	{
		for(int j = 0; j < size; j++)
		{
			if(cube.face[F].color[i][j] == F)
			{
				verif++;
			}
		}
	}
	}
	if(verif==55)
	{
	printf("OK : %d/55\n",verif);
	return 0;
	}
	else{

	printf("ERROR verification = %d/55\n",verif);
	return -1;
	}

}
```

Test complet des coups

```
Test_mode
Verif initialize() : OK : 55/55
recurence 4000 : rx() : OK : 55/55
recurence 4000 : ry() : OK : 55/55
recurence 4000 : rz() : OK : 55/55
recurence 4000 : right() : OK : 55/55
recurence 4000 : left() : OK : 55/55
recurence 4000 : top() : OK : 55/55
recurence 4000 : down() : OK : 55/55
recurence 4000 : front() : OK : 55/55
recurence 4000 : back() : OK : 55/55
recurence 4000 : right_inv() : OK : 55/55
recurence 4000 : left_inv() : OK : 55/55
recurence 4000 : top_inv() : OK : 55/55
recurence 4000 : down() : OK : 55/55
recurence 4000 : front_inv() : OK : 55/55
recurence 4000 : back_inv() : OK : 55/55
recurence 4000 : double_right() : OK : 55/55
recurence 4000 : double_left() : OK : 55/55
recurence 4000 : double_top() : OK : 55/55
recurence 4000 : double_down() : OK : 55/55
recurence 4000 : double_front() : OK : 55/55
recurence 4000 : double_back() : OK : 55/55
recurence 4000 : double_right_inv() : OK : 55/55
recurence 4000 : double_left_inv() : OK : 55/55
recurence 4000 : double_top_inv() : OK : 55/55
recurence 4000 : double_down_inv() : OK : 55/55
recurence 4000 : double_front_inv() : OK : 55/55
recurence 4000 : double_back_inv() : OK : 55/55
recurence 4000 : middle_x() : OK : 55/55
recurence 4000 : middle_y() : OK : 55/55
recurence 4000 : middle_z() : OK : 55/55
recurence 4000 : middle_x_inv() : OK : 55/55
recurence 4000 : middle_y_inv() : OK : 55/55
recurence 4000 : middle_z_inv() : OK : 55/55
```

Test de mélange et de résolution

On mélange 400 000 fois le cube de la manière suivante.

```
400000 recurence ry()+right() : ERROR verification = 21/55
```

TOP	FRONT	LEFT	RIGHT	DOWN	BACK
O W B	W R B	W G R	W B O	B G O	W O G
W W W	O R R	B G G	G B B	O Y R	O O Y
G W R	R Y G	G Y Y	Y Y Y	R B O	B R Y

Après avoir fait l'opération inverse 400 000 fois on retombe bien sur notre cube complet et résolue.

```
400000 recurence inv right()_inv()+ry_inv() : OK : 55/55
```

TOP	FRONT	LEFT	RIGHT	DOWN	BACK
W W W	R R R	G G G	B B B	Y Y Y	O O O
W W W	R R R	G G G	B B B	Y Y Y	O O O
W W W	R R R	G G G	B B B	Y Y Y	O O O

Bilan

Nous obtenu le résultat demandé dans le cahier des charges. Nous sommes globalement satisfait du projet et de la manière dont nous l'avons abordé. Nous avons acquis des compétences, notamment en langage C. Ces connaissances pourront être réutilisés dans divers projet à l'avenir.

Pour conclure, ce projet nous a permis d'apprendre la gestion de projet et le travail de groupe. Nous avons donc dû nous organiser. C'est à dire que nous avons partagé le travail en plusieurs partis afin de gagner du temps. Nous nous sommes par la suite réunis plusieurs fois afin de mettre en commun nos travaux. Celle-ci passe par la création d'une première version, et des idées de développement que nous avons mis en oeuvre. De plus, nous avons fait une deuxième version qui implémente plus d'option que la première sur lequel est implémenté notre travail de groupe. Nous nous sommes ensuite rencontrés pour corriger nos bugs Et faire la 3ème version. Par ailleurs, lors de ce travail nous avons remarqué qu'au final, nous avons eu une bonne cohésion de groupe.