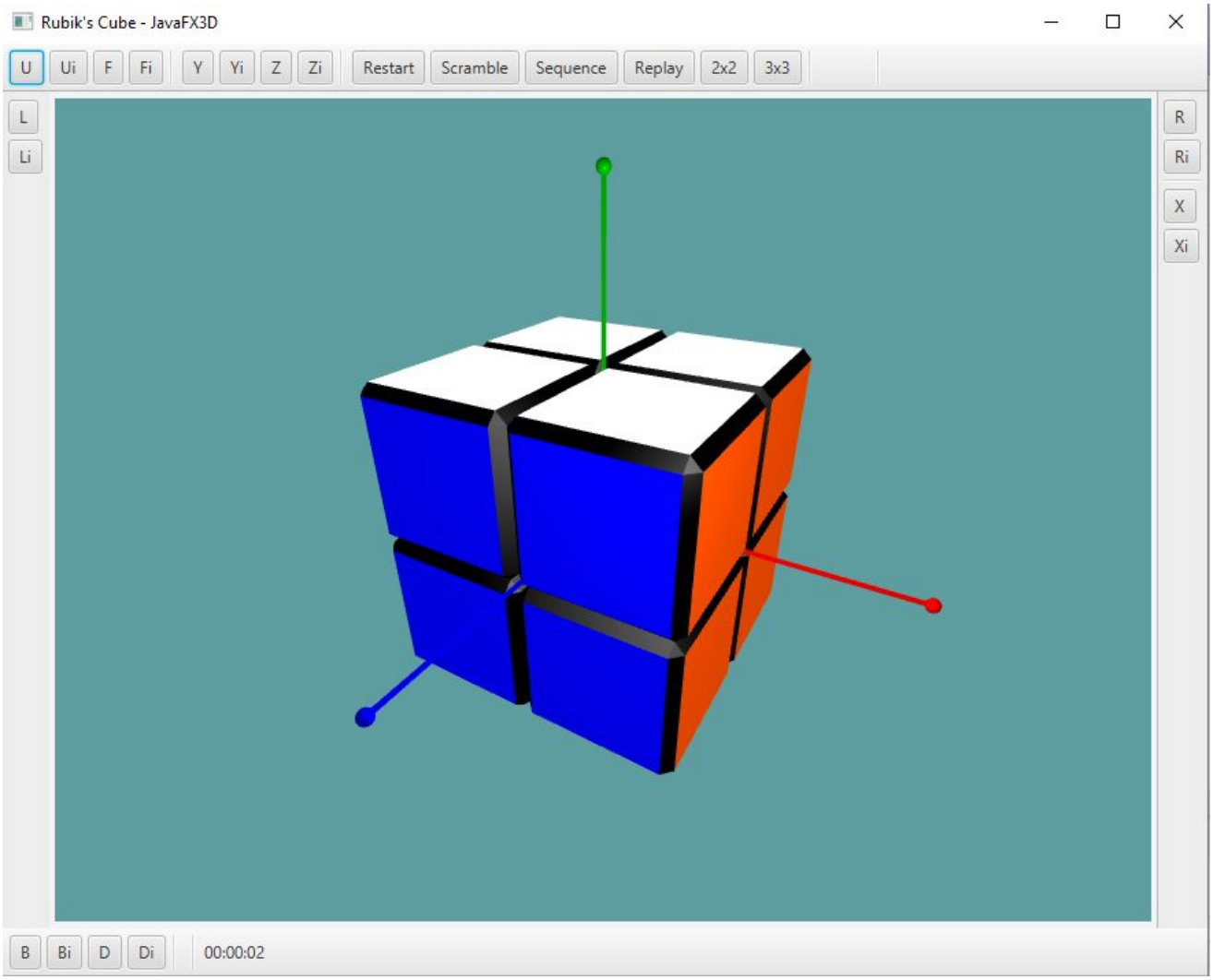
# Rubik’s cube




Stage réalisé par Thibaut BETEND en classe de L3 CDA à l’UBO
Supervisé par Monsieur Goulven GUILLOU

# *Sommaire*

# I) Présentation :

## 1) Objectifs :

L'objectif de ce projet est de créer un programme en java de manipulation de rubik's cube 2x2 en 3D. On pourra interagir avec celui-ci à l'aide de la souris, du clavier et / ou de boutons.

## 2) Pourquoi :

Ce projet était présenté parmi d'autres. Il m'a tout de suite attiré pour deux raisons :

- Le langage de programmation java est un de mes préférés et est celui que j'utilise le plus souvent dans mes projets personnels

- Le thème "rubik's cube" est un thème qui m'inspire beaucoup. J'en possède moi-même quatre de différentes sortes et je les manipule régulièrement.

## 3) Contexte :

Dans ce contexte un peu particulier du COVID-19, la plupart des entreprises n'était pas en mesure de recruter des stagiaires. C'est pourquoi nous réalisons un projet "engagé" par notre université.

# II) Cahier des charges :

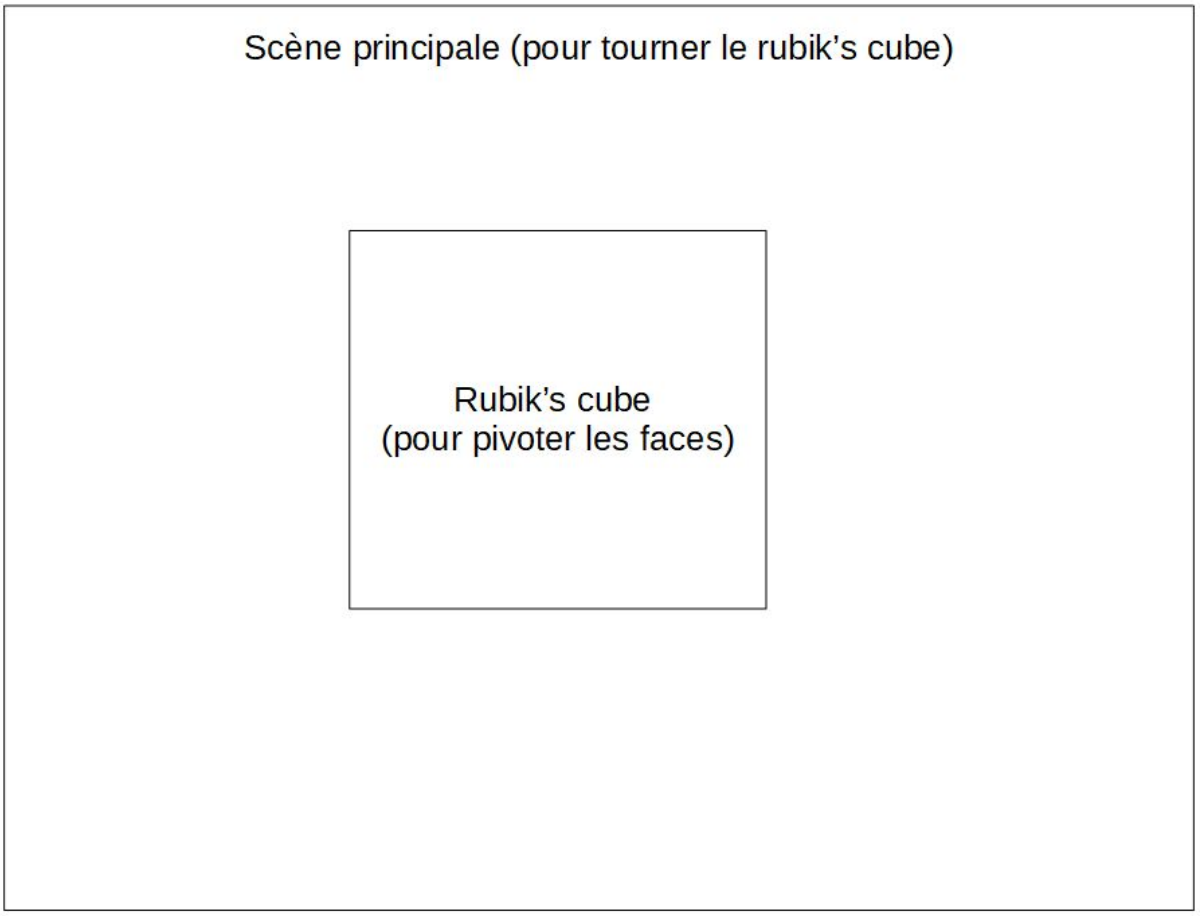
## 1) Objectifs :

L'objectif de ce projet est de créer un rubik's cube en 3D pouvant être manipulé à l'aide de la souris, du clavier et / ou de boutons. Nous commencerons par créer un rubik's cube 2x2. Celui-ci pourra donc tourner sur lui même et les différentes faces pourront bouger.

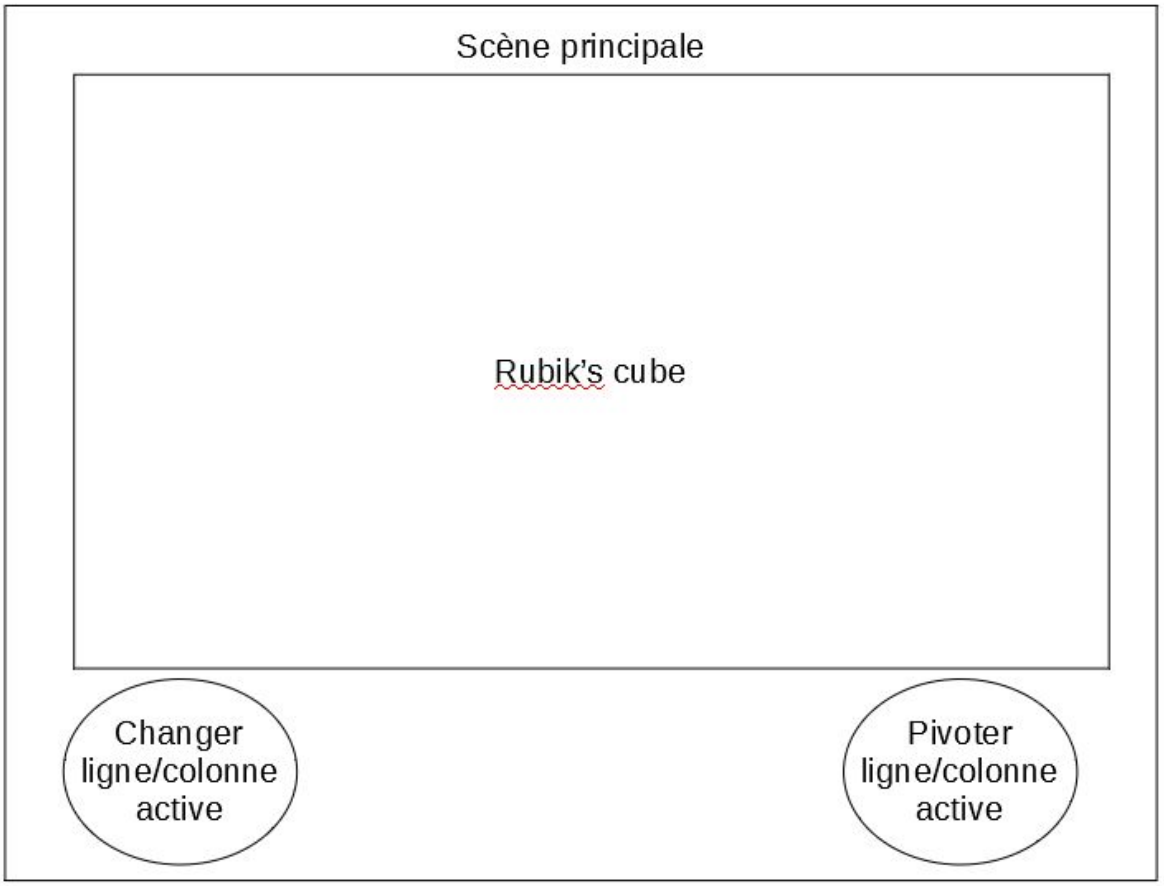
## 2) Maquettes :

Voici plusieurs sortes de maquettes pouvant être réalisées :

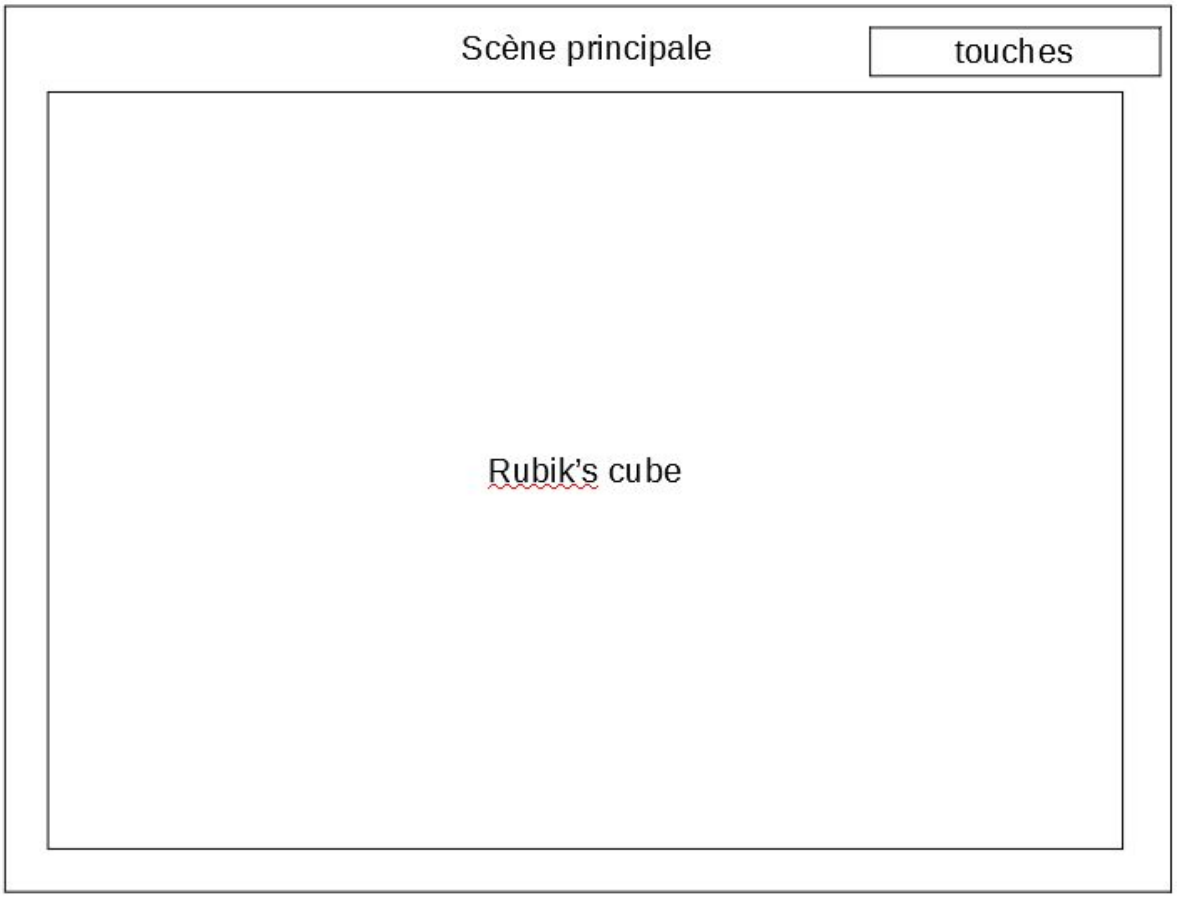
- Cette maquette permet la manipulation seulement à la souris. Pour tourner le Rubik's cube, il faut simplement "Drag" hors de celui-ci. Pour faire pivoter une face du Rubik's cube, il faut donc placer sa souris sur la face à pivoter et "Drag" dans le bon sens.



- Cette maquette permet d'interagir avec un rubik’s cube entièrement avec des boutons. Les boutons à gauche peuvent mettre en surbrillance les lignes et colonnes de cubes pour signifier qu’elles sont sélectionnées. Les boutons à droite peuvent tourner la ligne ou la colonne sélectionnée.



- Cette maquette permet d'interagir avec le rubik’s cube à l’aide du clavier. Les touches sont indiquées en appuyant sur un bouton ou une touche. Il sera donc possible de faire pivoter le rubik’s cube, choisir une ligne ou une colonne et la faire bouger au clavier.



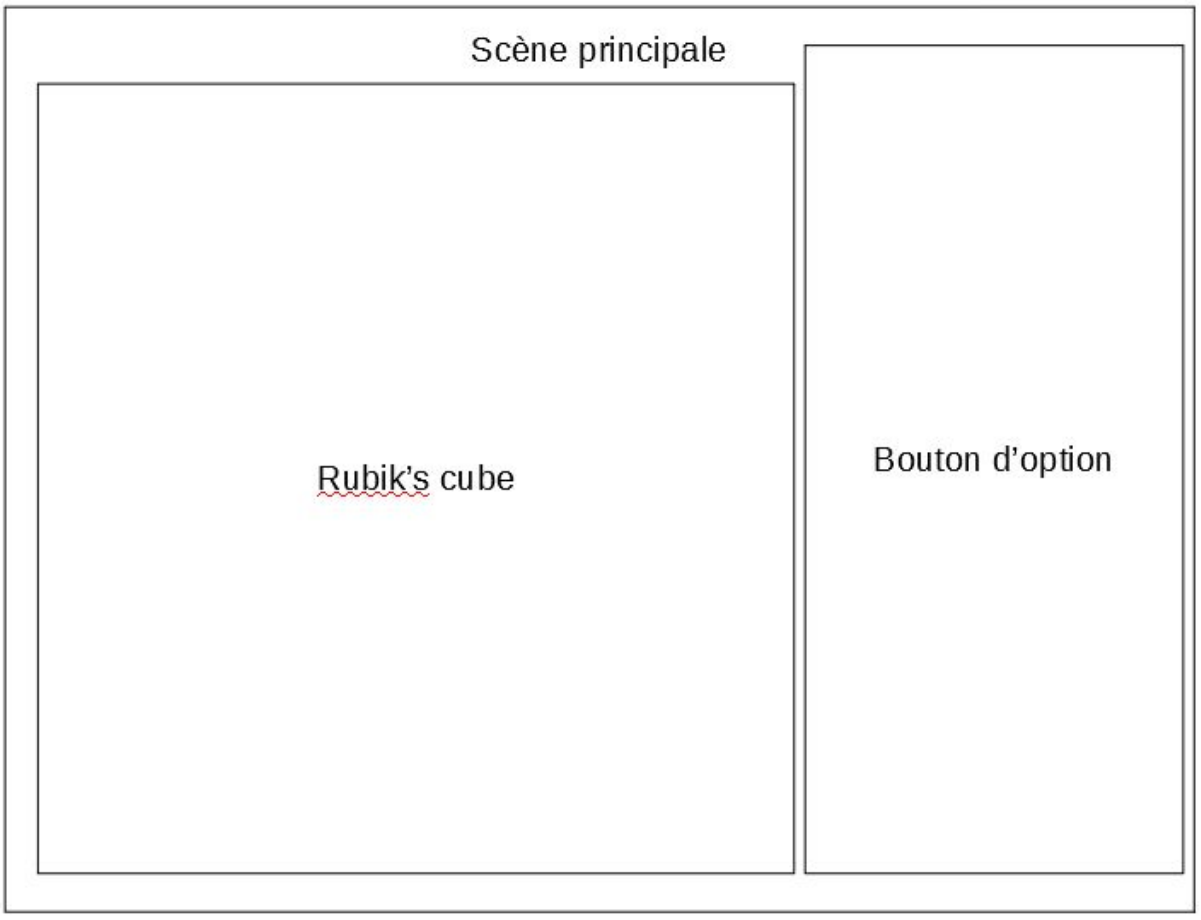

Nous partirons dans l’optique de la première maquette.

## 3) Caractéristiques techniques :

Pour réaliser ce projet, le choix d’une bibliothèque est nécessaire. J’ai donc opté pour la bibliothèque “JavaFX” qui est la plus commune de toutes. Ce sera donc une application java basée sur cette bibliothèque. Nous serons aussi amenés à utiliser le projet “3DViewer” et ControlsFx. Ce sera plus pour la partie esthétique.

## 4) Pour aller plus loin :

Le développement d'un menu pourra être réalisé comme suivant :




Ce menu pourra être lié à plusieurs types de rubik's cube : 2x2, 3x3…
Nous pourrons aussi ajouter une multitude d'options comme voir à travers le rubik's cube, changer les couleurs ect…

# III) Outils et technologies utilisés :

## 1) Outils :

- Eclipse :
  Cet environnement de développement est le meilleur outil que je connaisse pour la programmation en java. De plus, c'est celui que nous avons utilisé pendant nos cours.

- Sketchup :
  Outils de modélisation 3D essayé.

- Blender :
  Outils de modélisation essayé et utilisé.

- Navigateur web :
  Exclusivement pour des recherches sur le sujet du projet.

## 2) Langages et Bibliothèques :

- Java :
  Langage principal permettant le développement de cette application.

- JavaFx :
  Bibliothèque permettant la création d'applications client. Dans notre cas, elle est très utile au niveau du traitement de la 3D.

- 3D Viewer :
  Projet permettant d'importer et de traiter des objets 3D dans l'application.

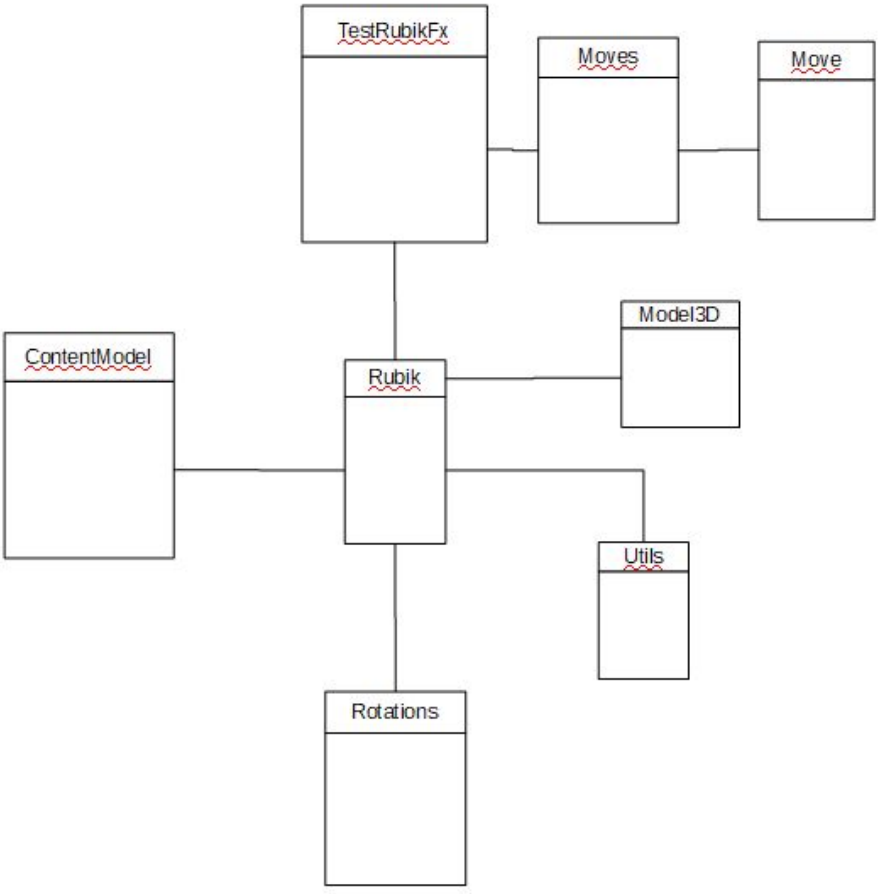
- ControlsFx :
  Projet Open Source permettant de créer des interfaces utilisateurs plus perfectionnées. Ce projet n'est pas utilisé dans la version "finale" du programme.

# IV) Développement de l'application :

## 1) Recherches :

Pour la première semaine du projet, j'ai surtout effectué beaucoup de recherches. J'ai donc pu me documenter sur les méthodes de création en 3D d'un rubik's cube et sur le langage le plus commun pour réaliser l'application. J'ai récupéré plusieurs liens me permettant de concevoir ce projet.
Tuto d'installation javaFX :
https://download.java.net/general/javafx/eclipse/tutorial.html#install
https://www.tutorialkart.com/javafx/install-javafx-in-eclipse-ide/
Code rubik's cube 3x3 javaFX : https://gist.github.com/jperedadnr/28534fcdce605b75382b
Tuto rubik's cube 3x3 3D javaFX :
http://jperedadnr.blogspot.com/2014/04/rubikfx-solving-rubiks-cube-with-javafx.html
ControlsFx : https://github.com/controlsfx/controlsfx
3D Viewer : http://openjdk.java.net/projects/openjfx/

Après avoir étudié le tutoriel pour la création du rubik's cube 3x3, j'ai pu réaliser ce diagramme de classe (Voir en annexe pour le diagramme complet) :



## 2) Rubik’s cube en javafx :

Pour la première version de mon rubik’s cube et suite à mes recherches, j’ai choisi de le réaliser entièrement en javaFX. Je me suis vite rendu compte que, pour réaliser des animations de rotation, la tâche allait être plus complexe que prévu avec cette méthode. De plus, comme chaque valeur était indiquée en dur dans mon programme, celui-ci n’était pas du tout optimisé. Voici une partie du programme montrant que la fabrication du rubik’s cube était faite à la main et donc pas optimisé pour le programme :

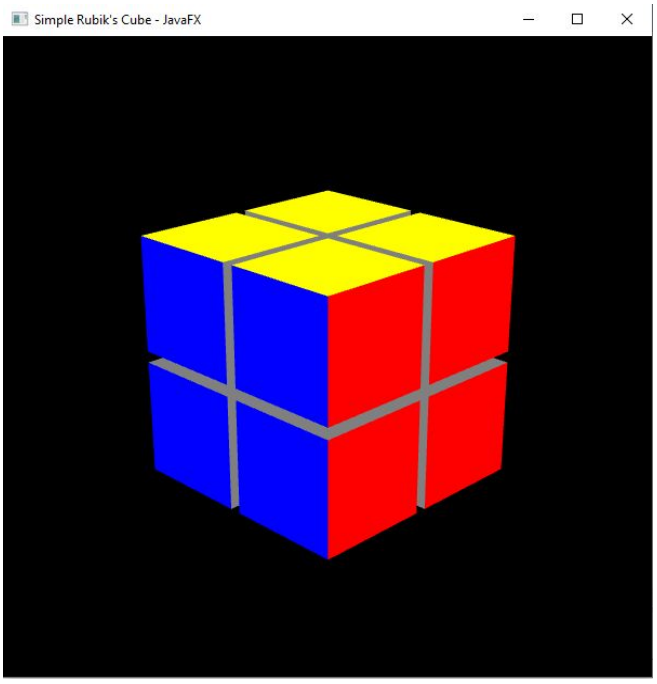
```
private static final int[] FLD  = new int[]{BLUE, GRAY, GRAY, GRAY, ORANGE, WHITE};
private static final int[] FRD  = new int[]{BLUE, RED, GRAY, GRAY, GRAY, WHITE};
private static final int[] FLU  = new int[]{BLUE, GRAY, YELLOW, GRAY, ORANGE, GRAY};
private static final int[] FRU  = new int[]{BLUE, RED, YELLOW, GRAY, GRAY, GRAY};

private static final Point3D pFLD   = new Point3D( posNeg,  pos, posNeg);
private static final Point3D pFRD   = new Point3D( pos, pos, posNeg);
private static final Point3D pFLU   = new Point3D( posNeg, posNeg, posNeg);
private static final Point3D pFRU   = new Point3D( pos, posNeg, posNeg);

private static final int[] BLD  = new int[]{GRAY, GRAY, GRAY, GREEN, ORANGE, WHITE};
private static final int[] BRD  = new int[]{GRAY, RED, GRAY, GREEN, GRAY, WHITE};
private static final int[] BLU  = new int[]{GRAY, GRAY, YELLOW, GREEN, ORANGE, GRAY};
private static final int[] BRU  = new int[]{GRAY, RED, YELLOW, GREEN, GRAY, GRAY};

private static final Point3D pBLD   = new Point3D( posNeg,  pos, pos);
private static final Point3D pBRD   = new Point3D( pos,  pos, pos);
private static final Point3D pBLU   = new Point3D( posNeg, posNeg, pos);
private static final Point3D pBRU   = new Point3D( pos, posNeg, pos);
```

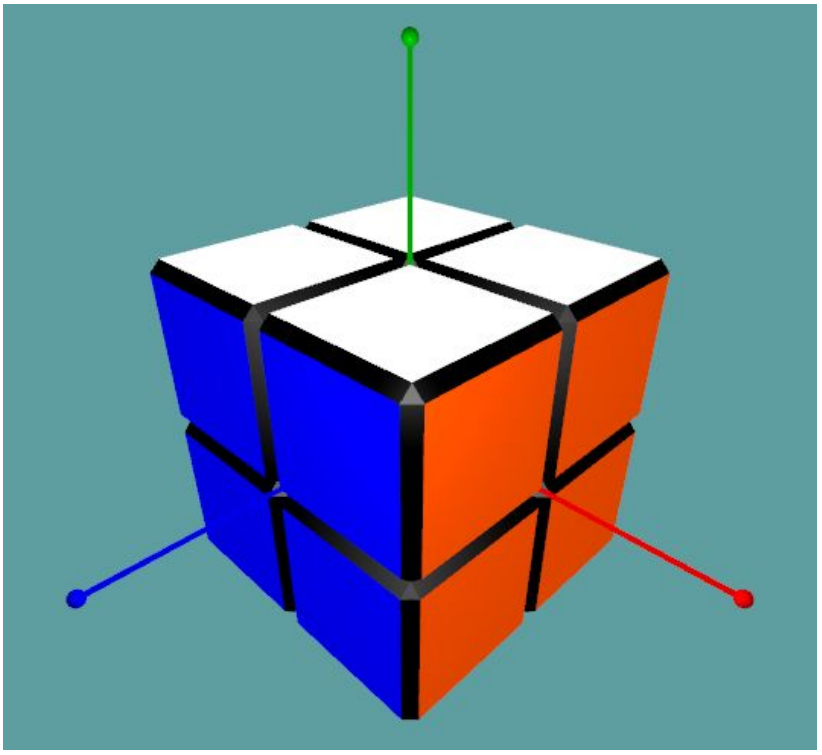
C’est pourquoi je l’ai abandonné.
Voici le rendu de la première version du rubik’s cube. Seul le rubik’s cube entier pouvait tourner.



## 3) Début de programme :

J'ai dû suivre le tuto recherché antérieurement. Ce tuto m'a permis d'obtenir trois axes (X, Y et Z) et de créer une scène correcte.

Voici les axes traversant le rubik's cube :



La classe ContentModel m'a donc permis de créer plusieurs effets sur mon application.
Pour commencer, la méthode "buildCamera" met en place la caméra de façon à, lorsque l'utilisateur lance le programme, avoir une position optimale du rubik's cube avec la face "avant" en bas à gauche (ici la face bleue).

La méthode "addLights" ajoute une lumière d'ambiance dans un angle qui est égal à : dimension de la scène * 0.6.
Pour finir, les autres méthodes sont des "handlers". Elles permettent des actions comme le déplacement de caméra (rotation du rubik's cube) et le zoom pouvant se faire de deux façons différentes : sur un "pad" d'ordinateur portable ou avec une souris.

## 4) Rubik’s cube 3D :

Pour la création du rubik's cube en 3D, j’ai utilisé plusieurs outils. Le but était de réaliser un rubik’s cube 2x2 en 3D contenu dans des fichiers .obj(forme) et .mtl(couleur).
J’ai donc pu obtenir un rubik’s cube 2x2 sans couleur en 3D. Le logiciel “Blender” m’a permis de le modifier et d’y apporter de la couleur et les cubes manquants.

Le logiciel est donc capable de lire le rubik’s cube avec une matrice :

```
private final int[][][] cube = { { { 50, 51, 52 }, { 49, 54, 53 }, { 59, 48, 46 } },
        { { 58, 55, 60 }, { 57, 62, 61 }, { 47, 56, 63 } },
        { { 67, 64, 69 }, { 66, 71, 70 }, { 68, 65, 72 } } };
```

Chaque nombre correspond à un cube du rubik’s cube bien précis. La première ligne correspond à la face avant d’en haut à gauche (Bleu/Rouge/Blanc) jusqu’en bas à droite (Bleu/Orange/Jaune). La deuxième ligne (inutile dans le cas du 2x2 mais utile dans le cas du 3x3) correspond à la face du milieu dans un ordre similaire à la première (de Blanc/Rouge jusqu’à Orange/Jaune). De même, la dernière ligne permet la gestion de la face arrière (de Vert/Rouge/Blanc à Vert/Orange/Jaune).

## 5) Les rotations “bouton” :

```
ToolBar tbTop=new ToolBar(new Button("U"),new Button("Ui"),new Button("F"),
                          new Button("Fi"),new Separator(),new Button("Y"),
                          new Button("Yi"),new Button("Z"),new Button("Zi"));
```

La création de “toolbar” a été nécessaire pour la visualisation correcte des mouvements du rubik’s cube. Cela a permis, pour chaque bouton appuyé, de faire tourner la face qui lui correspond dans le bon sens.

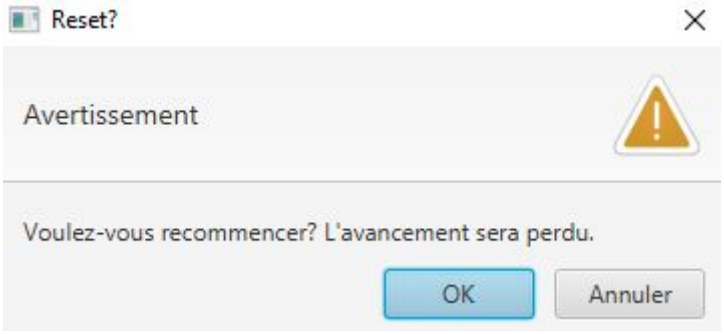
Les boutons envoient le sens de rotation du rubik’s cube à la méthode “rotateFace”. Ainsi, grâce à cette action et au “keyframe” qui crée l’animation, la face voulue se met en mouvement pendant 600 millisecondes.

# 6) Mélangeur, séquence et recommencer :

- Recommencer :
  Il a donc fallu créer un nouveau bouton pouvant réaliser cette action. La classe moves permet d'envoyer un message à l'utilisateur si le compteur de mouvements est supérieur à 0 (hors mélangeur et hors séquence). Le bouton, si l'utilisateur est d'accord pour recommencer (clique sur "ok" -> voir image ci-dessous), appelle la méthode "doReset". Celle-ci a pour fonction de remettre à 0 la caméra et le rubik's cube grâce à l'enregistrement de l'état "de base" :

```
mapMeshes.forEach((k,v)->mapTransformsOriginal.put(k, v.getTransforms().get(0)));
```

  ainsi que le futur compteur de mouvements et de temps.




- Mélangeur :
  Le mélangeur permet de réaliser 25 coups aléatoires pour mélanger le cube. Pour commencer, de même que pour le "Recommencer", il a fallu envoyer un message à l'utilisateur si les coups étaient supérieurs à 0 (hors mélangeur et hors séquence). Comme vous pouvez le voir, le mélangeur recommence le cube si des coups ont déjà été effectués :

```
Button bSc=new Button("Scramble");
bSc.setOnMouseClicked(e->{
    if(moves.getNumMoves()>0){
        Alert alert =
                new Alert(AlertType.WARNING,
                    "Voulez-vous mélanger le cube? " +
                    "L'avancement sera perdu.",
                     ButtonType.OK,
                     ButtonType.CANCEL);
        alert.setTitle("Mélangeur");

        Optional<ButtonType> result = alert.showAndWait();
        if (result.isPresent() && result.get() == ButtonType.OK) {
            rubik.doReset();
            doScramble();
        }
    } else {
        doScramble();
    }
});
```

Il suffit de créer une génération de 25 coups automatiques (méthode “doScramble” de rubik). Les mouvements sont enregistrés dans une variable :

```
private static final List<String> movements =
        Arrays.asList("F", "Fi", "F2", "R", "Ri", "R2",
                      "B", "Bi", "B2", "L", "Li", "L2",
                      "U", "Ui", "U2", "D", "Di", "D2");
```

Nous n’avons plus qu'à faire choisir au programme un chiffre entre 0 et 17 :
((int)(Math.floor(Math.random()*movements.size()))
Ils sont donc enregistrés dans le StringBuilder créé. La séquence sera lue grâce à la méthode “doSequence” expliquée dans le prochain point.

- Séquence :
  Du même type que les deux premiers, le bouton séquence a une particularité :
  Un textField permettant d’inscrire la séquence de coups voulus. Si l’utilisateur remplit et clique sur “ok”, le cube sera remis à zéro et “doSequence” sera appelé avec la séquence de l’utilisateur.
  Dans `doSequence(String list, boolean varSequence)`, les lignes suivantes permettent la compréhension et la mise en place de notation “correcte” :

```
List<String> asList = Arrays.asList(list.replaceAll("’", "i").replaceAll("'", "i").split(" "));

sequence=new ArrayList<>();
asList.stream().forEach(s->{
    if(!s.equals(""))
    {
        if(s.contains("2")){
            sequence.add(s.substring(0, 1));
            sequence.add(s.substring(0, 1));
        } else if(s.length()==1 && s.matches("[a-z]")){
            sequence.add(s.toUpperCase().concat("i"));
        } else {
            sequence.add(s);
        }
    }

});
```

Deux listeners sont nécessaires pour pouvoir lire entièrement la séquence. Le premier : `index.addListener((ov,v,v1)` permet l’arrêt de la séquence lorsque l’index arrive à la fin de celle-ci en supprimant le listener du deuxième.
Le deuxième : `onRotation.addListener(lis);` permet l’attente (rotation en cours) grâce au : `ChangeListener<Boolean> lis=(ov,b,b1)`.
Il reste plus qu’à sauvegarder les mouvements et à réaliser les mouvements avec :
`rotateFace(sequence.get(index.get()));`
et à incrémenter l’index pour chaque rotation.

# 7) Temps et compteur de mouvements :

- Compteur : Le compteur de mouvement est un simple label. Celui ci est alimenté par un Listener qui, pour chaque évolution d'un mouvement dans "moves" (ajout ou suppression de mouvement), permet au label de se mettre à jour constamment.

```
Label lMov=new Label();
rubik.getCount().addListener((ov,v,v1)->{
    bReset.setDisable(moves.getNumMoves()==0);
    bReplay.setDisable(moves.getNumMoves()==0);
    lMov.setText("Movements: "+(v1.intValue()+1));
});
```

Ce code permet aussi de désactiver les boutons reset et replay pour ne pas créer de problèmes.

- Temps : Cette fonctionnalité permet au joueur de savoir en combien de temps il a réalisé le rubik's cube. Celui ci s'arrête lorsque le joueur a terminé le rubik's cube. Les lignes ci-dessous permettent donc de créer, à partir d'un label, un timer qui s'allume au début du programme.

```
Label lTime=new Label();
lTime.textProperty().bind(clock);
tbBottom.getItems().addAll(new Separator(),lTime);

timer=new Timeline(new KeyFrame(Duration.ZERO, e->{
    clock.set(LocalTime.now().minusNanos(time.toNanoOfDay()).format(fmt));
}),new KeyFrame(Duration.seconds(1)));
timer.setCycleCount(Animation.INDEFINITE);
```

Voici une façon d'allumer le timer ou de permettre de le faire repartir de 0 (Bouton reset, mélange ...)

```
time=LocalTime.now();
timer.playFromStart();
```

## 8) Replay et enregistrement des mouvements :

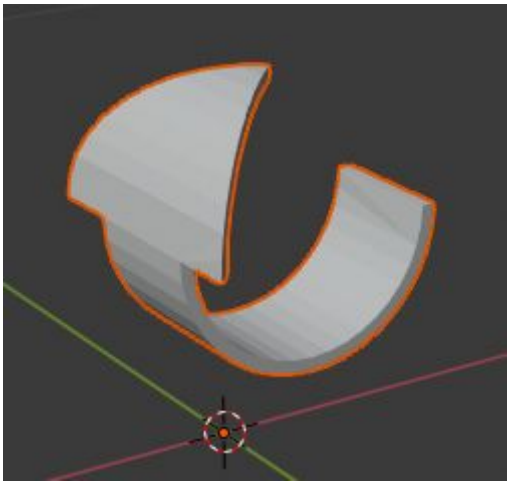
- Enregistrement :
  Pour chaque mouvement, deux classes ont été créées. Une qui enregistre seulement les mouvements (move) et une qui permet de regrouper tous les mouvements en liste dans l'ordre de réalisation.

- Replay :
  Le replay permet de revoir les mouvements réalisés à la main (hors séquence et mélangeur). La fonction doReplay de la classe principale envoie donc tous les mouvements à cette même fonction de la classe rubik. Cette méthode ressemble à "doSequence" vu plus haut et appelle donc la rotation pour chaque mouvement détecté dans la liste obtenue.

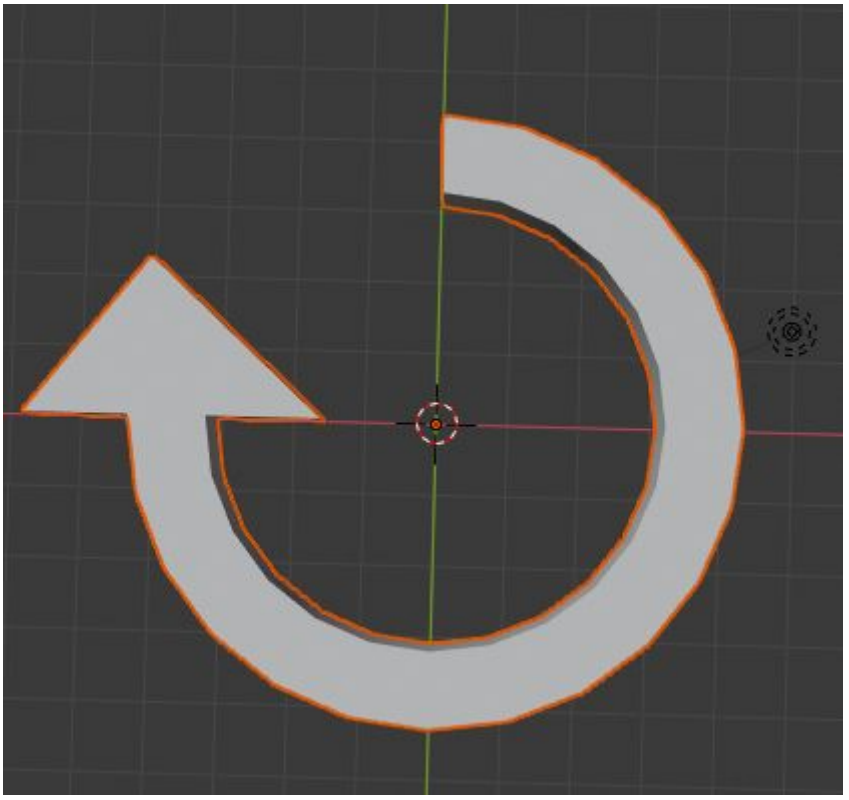
## 9) Prévisualisation des mouvements :

Voici les objets 3D permettant la prévisualisation du mouvement.

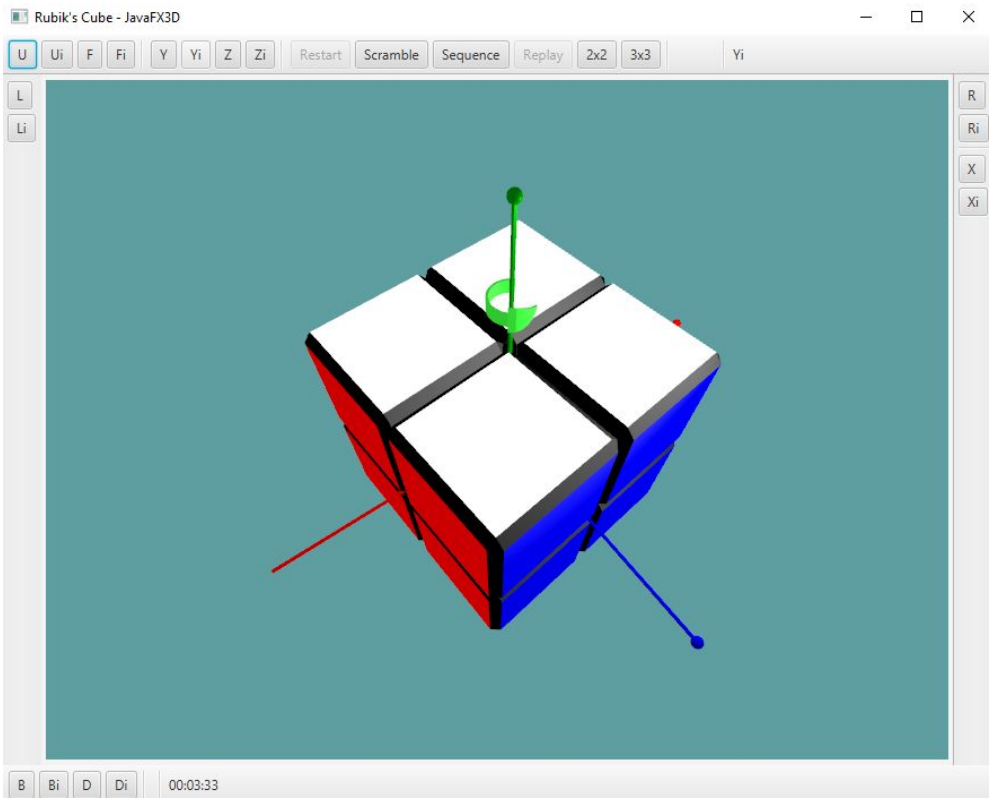
La première flèche montre la rotation du cube entier sur un des trois axes :



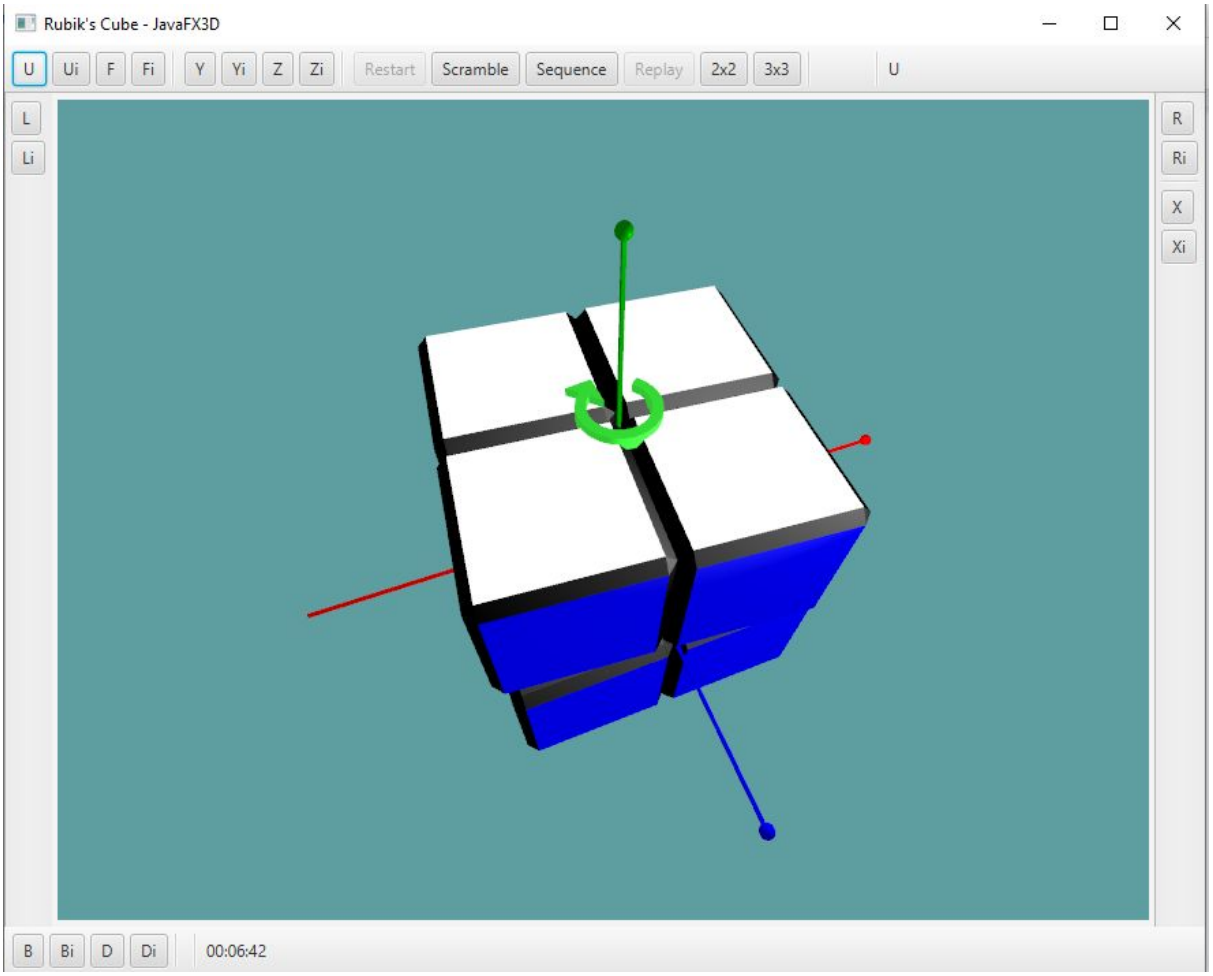
La seconde flèche montre la rotation d’une face du cube :



Voici une implémentation de la rotation axiale du cube :



Voici une implémentation de la rotation d'une face du cube :




La prévisualisation est permise par la variable "onPreview" et "bPreview". Si l'utilisateur choisit de tourner une face, cette ligne autorise le rubik's cube à réaliser une prévisualisation du mouvement, c'est à dire, faire une rotation de 5 degrés au lieu de 90 degrés :

```
double angEnd=(bPreview?5d:90d)*(btRot.endsWith("i")?1d:-1d);
```

Pour la création de l'animation, un "keyframe" a été réalisé par le tutoriel avec l'angle ainsi que le sens de rotation. Il détermine aussi le temps de rotation grâce à une autre lambda expression :

```
(Duration.millis(onScrambling.get() || onReplaying.get()?200:(bPreview?100:600)),
```

## 10) Rotation par souris :

La rotation par souris est faite grâce à un EventHandler<MouseEvent> eventHandler. La gestion se fait en trois étapes : le clic, le mouvement de souris et le relâchement du clic.

- le clic :
  Cette partie réalise un arrêt de la rotation de caméra. Le programme détecte le cube et la position de la souris et l'enregistre. Ce qui permet de détecter la face grâce à la méthode de la classe Utils : getPickedRotation();

- le mouvement :
  Cette partie autorise la prévisualisation du mouvement de la face ou du rubik's cube. Le programme détecte donc la direction dans laquelle l'utilisateur veut tourner la face ou le cube.

- le relâchement :
  Lors du relâchement, cette partie du programme détecte s'il y a un déplacement ou non (partie mouvement). Cela permet donc d'appeler soit la fonction de rotation, soit la fonction qui fera remettra le cube à la bonne place (annulation du mouvement de prévisualisation).

## 11) Changement de Rubik's cube :

Ce bouton, permettant le passage au rubik's cube 3x3, ré-ouvre l'application avec le nouveau paramètre (ici, monCube qui contient Cube3.obj). Lorsque le programme est arrêté, il est donc possible de l'ouvrir une autre fois avec le code "Platform.runLater".

```
Button bCube3=new Button("3x3");
bCube3.setOnMouseClicked(e->{
    monCube = "Cube3.obj";
    stage.close();
    Platform.runLater( () -> {
        try {
            new TestRubikFX().start( new Stage(), monCube );
        } catch (Exception e1) {
            // TODO Auto-generated catch block
            e1.printStackTrace();
        }
    } );
});
```

La méthode appelée est juste ici pour sauvegarder la taille du cube (ici Cube3.obj) et relancer le programme avec "start(stage);"

```
public void start(Stage stage, String taille) throws Exception {
    monCube = taille;
    start(stage);
}
```

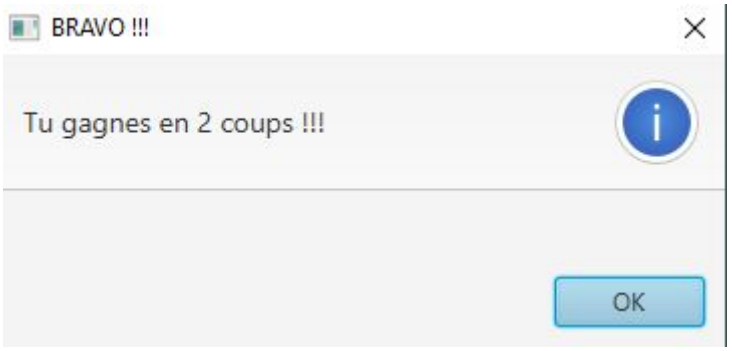
Par défaut, lors de l'exécution du programme, on pourra afficher le cube en 2x2 :

```
if(monCube.equals(null)||monCube.equals(""))
{
    monCube = "Cube2.obj";
}
```

## 12) Résolution du rubik’s cube :

Pour chaque rotation, l’ordre du cube est vérifié pour voir s’il est résolu ou non. Cette vérification est réalisée par la méthode “checkSolution” de la classe utils. Elle permet de regarder chaque partie du cube pour calculer s’il est remis dans une position où chaque cube du rubik’s cube est dans le bon ordre.

Pour finir, si le programme détecte que le rubik’s cube est résolu, il l’écrira, stoppera le temps et affichera un pop-up informatique comme quoi l’utilisateur a gagné. De plus, une musique de victoire sera jouée grâce à la variable “mediaPlayer”.




```
rubik.isSolved().addListener((ov,b,b1)->{
    if(b1){
        sound = new Media(new File(musicFile).toURI().toString());
        mediaPlayer = new MediaPlayer(sound);
        mediaPlayer.play();

        lSolved.setVisible(true);

        timer.stop();
        Alert alert =
                new Alert(AlertType.INFORMATION,
                    "",
                     ButtonType.OK);
        alert.setHeaderText("YOU WIN !!!");
        alert.setTitle("CONGRATULATION !!!");
        alert.show();
        moves.setTimePlay(LocalTime.now().minusNanos(time.toNanoOfDay()).toNanoOfDay());
        System.out.println(moves);
    } else {
        lSolved.setVisible(false);
    }
});
```

## 13) Difficultés :

- Changement de Bibliothèque :
  Pendant la réalisation du tutoriel, j'ai pu constater que la bibliothèque "ControlsFx" ne fonctionnait plus et faisait de grosses erreurs sur mon programme. En effet, elle appelait une classe de JavaFx qui n'existe plus dans la version actuelle. ControlsFx était principalement utilisé pour des pop-ups. J'ai réalisé beaucoup de recherches et passé beaucoup de temps sur une adaptation de ces même pop-ups en JavaFx.

- Passage du 3x3 en 2x2 :
  Le passage du rubik's cube de sa taille 3x3 en 2x2 et l'adaptation du programme a été une de mes plus grandes difficultés. Pour commencer, je ne connaissais pas grand chose en modélisation 3D. J'ai dû me former sur ce sujet pour pouvoir créer moi même le cube 2x2.

# V) Conclusion :

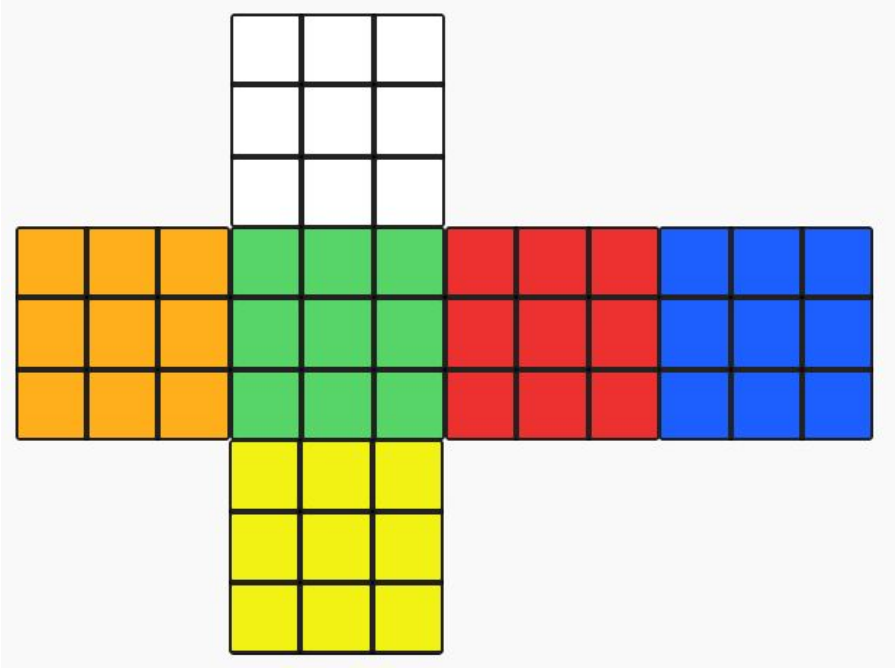
## 1) Intérêt du projet :

- Auto-Formation :
  Ce projet m'a permis d'obtenir des compétences de programmation diverses. En effet, j'ai pu me former sur différents sujets.
  Le premier est le JavaFX. Connaître une bibliothèque pouvant gérer des objets 3D pourra me permettre de créer des applications (et notamment des jeux) de plus en plus poussées.
  Le second est la modélisation 3D. J'étais déjà intéressé par cette spécialité. J'ai donc pu me plonger plus en profondeur dans ce domaine vaste.

- Accomplissement personnel :
  La modélisation et la création de jeux virtuels ont toujours été un domaine pour lequel je porte un grand intérêt. Ce projet m'a permis de créer mon deuxième jeux vidéo. Je pense aussi le continuer dans un futur proche et lui rajouter des améliorations.

## 2) Ajout d'une procédure de résolution automatique :

Pour réaliser une résolution automatique, nous devons créer une matrice permettant la visualisation au programme de chaque face. Il pourra donc analyser chaque face comme dans l'image suivante :

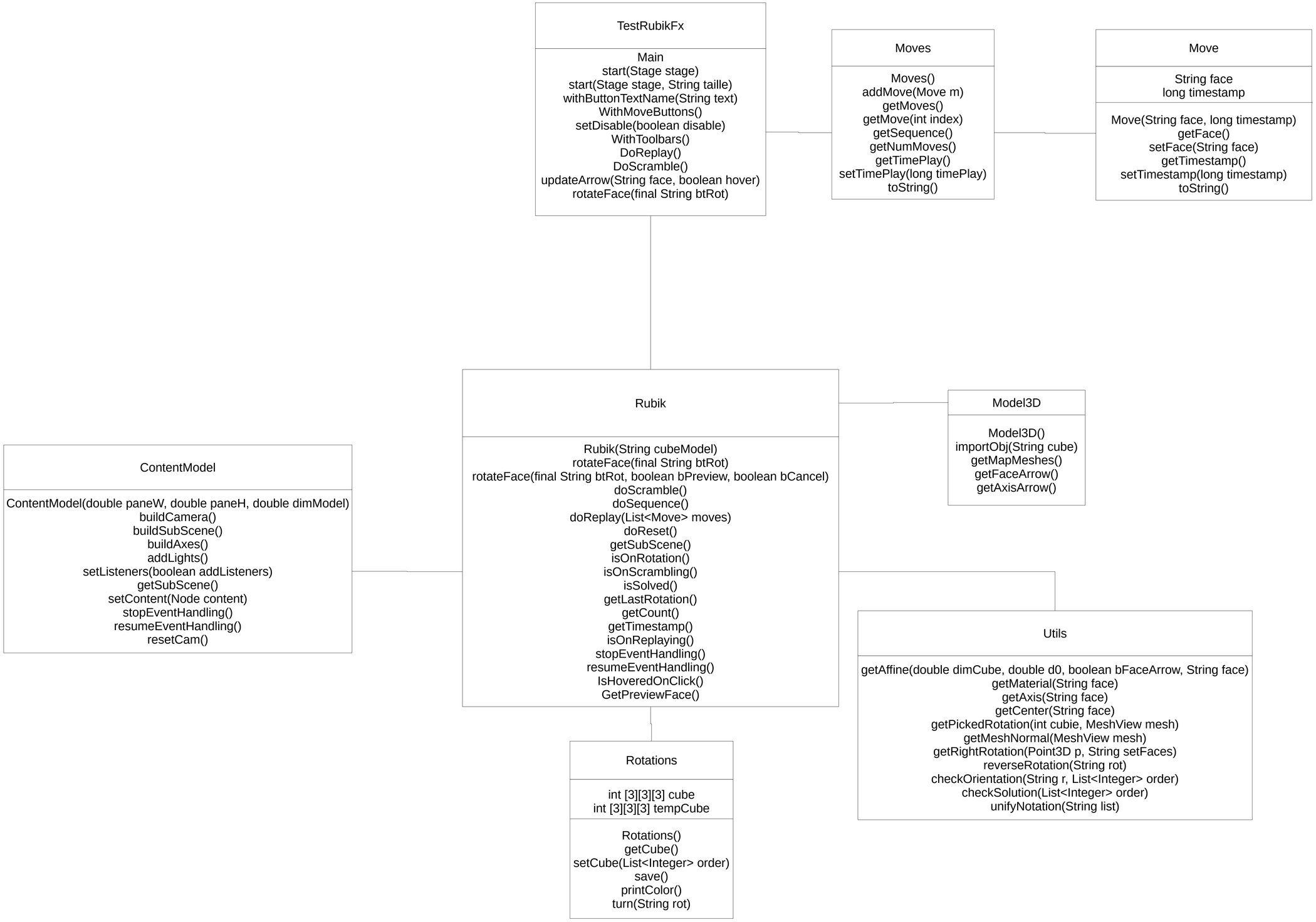


Voici une représentation de la matrice :

```
private final String[][][] color = {
        { { "50 B" }, { "51 B" }, { "52 B" }, { "49 B" }, { "54 B" }, { "53 B" }, { "59 B" }, { "48 B" },
                { "46 B" } },
        { { "50 W" }, { "51 W" }, { "52 W" }, { "58 W" }, { "55 W" }, { "60 W" }, { "67 W" }, { "64 W" },
                { "69 W" } },
        { { "67 G" }, { "64 G" }, { "69 G" }, { "66 G" }, { "71 G" }, { "70 G" }, { "68 G" }, { "65 G" },
                { "72 G" } },
        { { "59 Y" }, { "48 Y" }, { "46 Y" }, { "47 Y" }, { "56 Y" }, { "63 Y" }, { "68 Y" }, { "65 Y" },
                { "72 Y" } },
        { { "50 R" }, { "49 R" }, { "59 R" }, { "58 R" }, { "57 R" }, { "47 R" }, { "67 R" }, { "66 R" },
                { "68 R" } },
        { { "52 O" }, { "53 O" }, { "46 O" }, { "60 O" }, { "61 O" }, { "63 O" }, { "69 O" }, { "70 O" },
                { "72 O" } } };
```

Elle permet de visualiser le numéro d'un des cubes du rubik's cube ainsi que la couleur de la face visée. Il est donc possible de reproduire les mouvements du rubik's cube en obtenant la place de toutes les faces de celui-ci.

# ANNEXES


TestRubikFx
Main
start(Stage stage)
start(Stage stage, String taille)
withButtonTextName(String text)
WithMoveButtons()
setDisable(boolean disable)
WithToolbars()
DoReplay()
DoScramble()
updateArrow(String face, boolean hover)
rotateFace(final String btRot)
Moves
Moves()
addMove(Move m)
getMoves()
getMove(int index)
getSequence()
getNumMoves()
getTimePlay()
setTimePlay(long timePlay)
toString()
Move
String face
long timestamp
Move(String face, long timestamp)
getFace()
setFace(String face)
getTimestamp()
setTimestamp(long timestamp)
toString()
Rubik
Rubik(String cubeModel)
rotateFace(final String btRot)
rotateFace(final String btRot, boolean bPreview, boolean bCancel)
doScramble()
doSequence()
doReplay(List<Move> moves)
doReset()
getSubScene()
isOnRotation()
isOnScrambling()
isSolved()
getLastRotation()
getCount()
getTimestamp()
isOnReplaying()
stopEventHandling()
resumeEventHandling()
IsHoveredOnClick()
GetPreviewFace()
Model3D
Model3D()
importObj(String cube)
getMapMeshes()
getFaceArrow()
getAxisArrow()
ContentModel
ContentModel(double paneW, double paneH, double dimModel)
buildCamera()
buildSubScene()
buildAxes()
addLights()
setListeners(boolean addListeners)
getSubScene()
setContent(Node content)
stopEventHandling()
resumeEventHandling()
resetCam()
Rotations
int [3][3][3] cube
int [3][3][3] tempCube
Rotations()
getCube()
setCube(List<Integer> order)
save()
printColor()
turn(String rot)
Utils
getAffine(double dimCube, double d0, boolean bFaceArrow, String face)
getMaterial(String face)
getAxis(String face)
getCenter(String face)
getPickedRotation(int cubie, MeshView mesh)
getMeshNormal(MeshView mesh)
getRightRotation(Point3D p, String setFaces)
reverseRotation(String rot)
checkOrientation(String r, List<Integer> order)
checkSolution(List<Integer> order)
unifyNotation(String list)