

Ce document est l'un des livrables à fournir lors du dépôt de votre projet : 4 pages maximum (hors documentation).

Pour accéder à la liste complète des éléments à fournir, consultez la page [Préparer votre participation](#).

Vous avez des questions sur le concours ? Vous souhaitez des informations complémentaires pour déposer un projet ? Contactez-nous à info@trophees-nsi.fr.

NOM DU PROJET : RUBIX

> PRÉSENTATION GÉNÉRALE :

L'idée du projet est issue de notre intérêt prononcé pour les matrices vues en première en NSI. Nous cherchions ainsi un projet comprenant matrices, algorithmique, musique et programmation orientée objet – vue en terminale.

C'est en pensant à Alpha Go Zero, développé par Google, que nous avons eu l'idée de simuler un jeu et le résoudre. Le Rubik's Cube, inventé Ernő Rubik en 1974, est un casse-tête qui nous permettait de satisfaire nos goûts.

Nous avons donc imaginé « Rubix », un projet programmé en Python qui serait capable de modéliser un Rubik's Cube et de le résoudre. Le programme doit aussi pouvoir représenter sa résolution en 3D, pour indiquer à l'utilisateur comment le résoudre. Finalement, un membre de notre groupe, passionné de musique, nous a suggéré d'inclure de l'art aléatoire : générer une musique en fonction des mouvements effectués pour résoudre le cube.

Ce programme permet ainsi la résolution de cubes générés aléatoirement, via l'algorithme à deux phases de Herbert Kociemba, du Rubik's Cube 3 par 3. Sa résolution est animée via plusieurs mouvements en 3D, performés à l'aide des bibliothèques PyOpenGL et Pygame. Une interface graphique, construite à l'aide de la bibliothèque Tkinter, nous permet de centraliser les commandes du projet ; et de saisir un Rubik's Cube à la main.

Mais nous trouvions cette dernière étape de saisie fastidieuse et longue. Nous avons donc bâti un système de reconnaissance d'objets et de couleurs, permettant de scanner à l'aide d'une caméra, un Rubik's Cube 3 par 3. Pour cela, nous avons utilisé les bibliothèques OpenCV et Scipy.

> ORGANISATION DU TRAVAIL :

Notre équipe est constituée de cinq membres : M. Joris Bely, M. Nicolas Calderan, M. Sacha Guitteny, M. Lucas Loustalot et M. Luigi Tintinaglia.

Les tâches ont été réparties selon les compétences et les envies de chacun. Joris a travaillé sur la diffusion de musique, la conception et le montage de la vidéo de présentation du projet. Nicolas a créé l'interface utilisateur et rédigé les documents relatifs à la vidéo et à l'envoi du projet. Sacha a quant à lui plus particulièrement travaillé sur l'enregistrement et la cohérence des notes de musique. Lucas a créé l'interface 3D, les animations et l'a liée à l'interface utilisateur. Finalement, Luigi a dédié son travail à la résolution du Rubik's Cube, au scan par caméra, ainsi qu'à la structuration et la documentation du projet.

Nous avons utilisé une méthode « agile » pour notre projet. Chacun devait accomplir à de courtes échéances des tâches ; qui étaient ensuite mises en commun. Après quelques heures de travail sur les heures de spécialité NSI, le travail à la maison était réparti de la même façon, plus intensément sur les week-ends. Chaque lundi et mercredi matins, un point était effectué avant les cours de spécialité. Luigi et Lucas ont veillé à la cohérence et à l'intégrité des modifications effectuées avec les autres branches du projet.

Nous avons tous opté pour coder avec l'environnement de Visual Studio Code. En effet, nous avons stocké le projet sur GitHub ; et Visual Studio Code permet de faire un lien efficace avec la plateforme. Pour échanger le soir et les week-ends, nous utilisons un serveur Discord réunissant tous les membres du groupe.

> LES ÉTAPES DU PROJET :

L'idée du projet « Rubix » nous est apparu à la fin du mois de janvier. Sous son nom de code « RBX », nous avons commencé nos recherches durant la première moitié du mois de février. L'objectif était le suivant : trouver de nouvelles idées et établir rationnellement ce qui était réalisable ou non ; nous en parlerons plus en détail dans la partie « Fonctionnement et opérationnalité ». Nous voulions aussi un programme avec une réelle valeur ajoutée ; nous avons donc allongé notre temps de réflexion et de premiers essais jusqu'à la fin du mois de février ; ce qui nous a permis d'enregistrer les sons nécessaires et de commencer l'animation 3D.

Durant le mois de mars, c'est la résolution qui fut au cœur de nos réflexions : algorithme de Thistlethwaite, algorithme de Kociemba, « God's algorithm¹ » ou encore avec l'apprentissage par renforcement. Nous avons parallèlement imaginé et construit l'interface graphique, et réfléchi à la conception du logo du projet.

Finalement, durant le mois d'avril, nous avons travaillé à la structuration de notre projet : lier les différentes parties entre elles ; ainsi que les documenter de façon précise. C'est également durant cette période que nous avons eu l'idée de la reconnaissance d'objets, que nous avons implémenté mi-avril.

La seconde partie du mois d'avril fut consacrée à la rédaction des documents nécessaires au projet, à la conception et au montage de la vidéo. Parallèlement, nous avons effectué quelques améliorations suite aux nombreux tests que nous avons pu réaliser. Finalement, nous remercions Baptiste Montagne, camarade qui a réalisé notre logo grâce à ses talents de dessinateur.

¹ « Algorithme de Dieu » : celui qui résout le cube optimalement.

> FONCTIONNEMENT ET OPÉRATIONNALITÉ :

Le projet Rubix est entièrement fonctionnel. La reconnaissance d'objets et de couleurs, de même que la résolution et la modélisation 3D fonctionnent.

Pour nous assurer de son intégrité, nous avons testé notre programme sur différentes plateformes (Windows, MacOS, Raspberry Pi OS, Ubuntu) ; et nous l'avons fait tester par nos camarades et amis. Cela nous a permis d'optimiser la simplicité de notre programme et de corriger son ergonomie.

Au cours des trois mois de développement, nous avons rencontré plusieurs difficultés.

La première était celle de la méthode de résolution : nous voulions un algorithme très rapide et proche de l'algorithme optimal. Nous avons donc éliminé les algorithmes intuitifs et le « God's algorithm » pour leur lenteur. Nous avons essayé, durant plusieurs semaines, d'implémenter un algorithme d'apprentissage par renforcement, mais nous n'y sommes pas arrivés, à cause de notre manque de connaissances dans ce domaine. Ainsi, nous avons choisi un compromis : l'algorithme de Kociemba, version évoluée de l'algorithme à deux phases de Thistlethwaite ; qui résout très rapidement un Rubik's Cube en 19 coups en moyenne.

Ensuite, l'imprécision de la reconnaissance de couleurs que nous avons mis en place, à l'aide de bornes inférieures et supérieures pour identifier les couleurs (en HSV : Hue, Saturation, Value) ne fut efficace que dans certaines conditions : absence stricte de reflets, luminosité et balance des blancs stable... Nous avons donc opté pour l'utilisation d'un arbre de classification de la librairie Scipy, déjà entraîné ; mais qui ne s'avère pas toujours efficace. Nous avons donc mis en place une interface pour corriger le Rubik's Cube scanné si les couleurs scannées s'avéraient incorrectes.

> OUVERTURE :

Le projet Rubix a bien entendu plusieurs améliorations à recevoir. Premièrement, nous souhaitons optimiser la détection de couleurs en entraînant nous-même un modèle de classification. En effet, selon les différents environnements que nous avons testés, le Rubik's Cube détecté n'était pas toujours fidèle à la réalité.

Ensuite, nous avons imaginé lier ce projet avec de la robotique ; en construisant une machine qui serait capable de résoudre un Rubik's Cube réel, en le scannant notamment.

Nous aurions souhaité développer le rôle de la musique et de l'art aléatoire dans notre projet, en implémentant un algorithme d'harmonisation des notes ; ainsi qu'en élargissant les gammes représentées.

Nous sommes fiers du résultat obtenu par le projet ; mais nous aurions aimé changer quelques approches parfois inefficaces que nous avons eu. Nous pensons que nous aurions dû commencer par implémenter un algorithme simple, intuitif pour la résolution avant l'apprentissage par renforcement ; nous aurions pu ainsi approfondir la résolution techniquement.

De la même façon, nous aurions voulu implémenter les différents modules du programme (scan, animation 3D...) en programmation parallèle pour éviter les situations bloquantes ; quand bien même nous avons tenté de les minimiser.

Envisager une pluralité des algorithmes employés pour la résolution aurait également été intéressant : utiliser des algorithmes intuitifs pour apprendre aux novices, jusqu'aux algorithmes utilisant les groupes et les permutations comme Kociemba ou l'algorithme de Dieu pour les experts.

Pour diffuser notre projet au grand public, nous envisageons de le porter dans plusieurs lycées où nous connaissons nombreux les amateurs de Rubik's Cube comme nous. Il serait également intéressant de faire tester notre projet par les participants à des championnats de Rubik's Cube, toujours intéressés par des méthodes de résolution parallèles. De même, faire découvrir à des jeunes au collège la NSI au lycée et le Rubik's Cube serait idéal dans le cadre de ce projet !

En dernier lieu, il serait judicieux de placer un QR code sur chaque boîte de Rubik's Cube contenant un lien de téléchargement du projet. Cela permettrait au nouvel acheteur d'apprendre à manipuler et résoudre son cube, une fois de nouveaux algorithmes implémentés, de façon interactive et ludique avec Rubix.

DOCUMENTATION

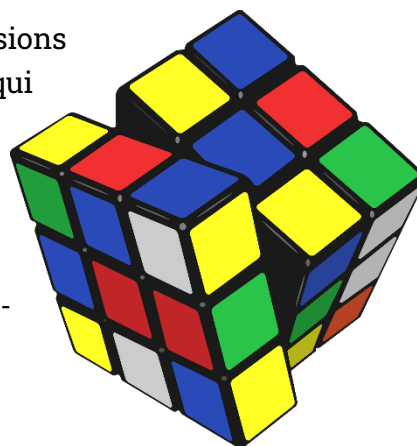
Toutes les images tierces utilisées dans cette partie sont libres de droit.

I) Les bases du Rubik's Cube

Le Rubik's Cube est un casse-tête géométrique à trois dimensions inventé par Ernő Rubik en 1974. Il comporte 26 petits cubes qui peuvent se déplacer sur toutes les faces.

Les cubes situés au centre des faces ne se déplacent pas.

Le mécanisme qui permet de faire déplacer les cubes sur les différents axes du cube a été breveté par le créateur du casse-tête lui-même.



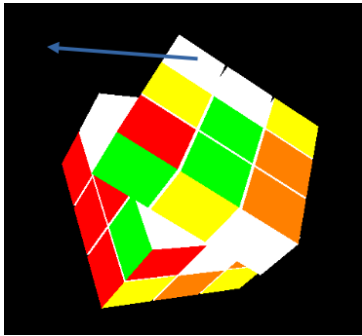
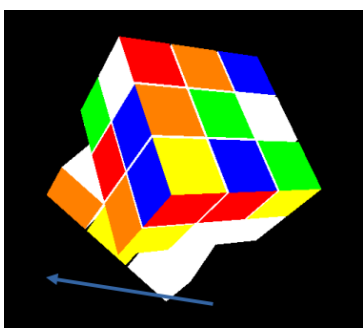
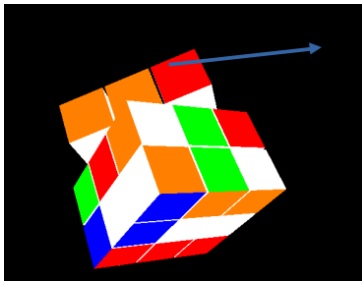
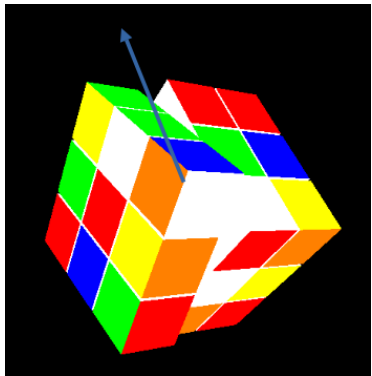
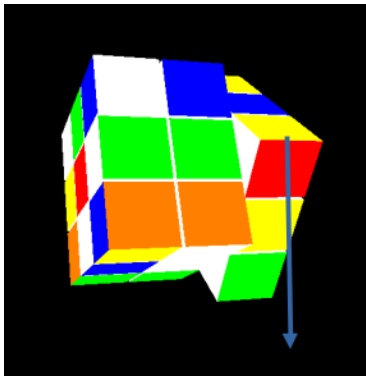
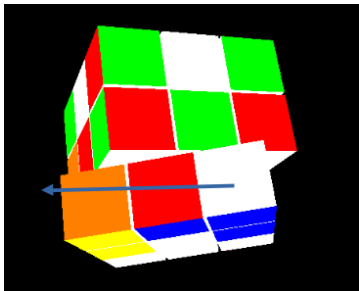
A. Formules et notations internationales

Pour performer des mouvements sur un cube, dans le but de le résoudre ou de le mélanger, des notations ont été instaurées pour les simplifier.

Les six mouvements de base sont :

F (Front)	B (Back)	U (Up)
L (Left)	R (Right)	D (Down)

Les mouvements dans le sens opposé se notent :

F'	B'	U'
		
L'	R'	D'
		

De même, voici la notation des mouvements doublés (on fait deux fois le même mouvement) : U2, D2, L2, R2, F2, B2

Il serait inutile de noter U'2 car le résultat de U'2 est le même que celui de U2.

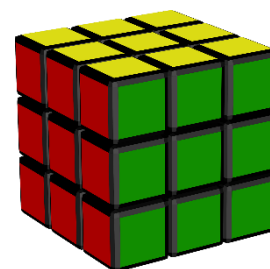
Nous avons donc utilisé 18 mouvements : U, D, L, R, F, B, U', D', L', R', F', B', U2, D2, L2, R2, F2, B2.

Il existe d'autres mouvements qui sont des suites de ces mouvements de base, pour les experts. Par exemple on note M = L' R X' avec X' qui fait tourner entièrement le cube vers le haut. En réalité, nous n'avons pas l'utilité de ces mouvements plus complexes.

Une formule, c'est un enchaînement de mouvements. Par exemple, « U' B F R2 » est une formule que l'on peut appliquer sur notre cube.

Pour la suite, et par abus de langage, nous dirons que la « face verte » du cube est la face dont la couleur du cube au centre est verte.

Pour appliquer ces formules, nous nous placerons toujours avec la face verte face à nous et la face jaune vers le *haut* (voir image ci-contre).



B. Des Rubik's Cube impossibles ?

Paradoxalement, il existe des Rubik's Cube qu'on ne peut pas résoudre. Ces derniers, que l'on appellera « impossibles », ont été modifiés par l'humain : un petit cube a été changé de place, etc...

Il existe des outils mathématiques pour vérifier la solvabilité d'un cube, utilisant notamment la théorie des groupes, en représentant chacune des rotations par une lettre. On peut ainsi modéliser l'ensemble des configurations d'un Rubik's cube par un groupe fini.

Ces méthodes pour vérifier la solvabilité d'un cube non-résolu sont :

- Il faut qu'il y ait exactement 8 coins distincts.
- Il faut qu'il y ait exactement 12 « facelets » distincts (cubes au milieu des arêtes du cube).
- Il faut qu'il y ait exactement un « facelet » de chaque couleur.
- Une arête doit être « flippée » : le « facelet » de cette arête doit être bien placé, mais mal orienté.
- Un coin doit être « twisté » : bien placé dans le cube, mais mal orienté.
- Deux coins ou deux arêtes doivent être échangés.

Nous nous en servons pour déterminer, lors du scan, s'il n'y pas eu d'erreur.

C. Algorithmes, mélange et « nombre de Dieu »

1. Les algorithmes intuitifs et communs

Ces méthodes de résolution sont souvent employées par les débutants pour résoudre un cube. Elles consistent, par exemple, à reconnaître une configuration du cube et appliquer une formule jusqu'à arriver à une autre configuration connue.

Quand bien même ces méthodes sont facilement applicables – comme le « Belge » - le nombre de coups employés est considérable ; et peut parfois dépasser les 50 mouvements.

De plus, la formule de résolution est construite au cours de la résolution, ce que nous ne voulions pas.

2. L'apprentissage par renforcement

La piste du Deep Learning nous paraissait avantageuse, notamment pour son efficacité et sa proximité avec l'apprentissage humain.

Néanmoins, nous avons manqué de connaissances dans le domaine pour implémenter les réseaux de neurones et les agents de la bibliothèque `keras-rl`.

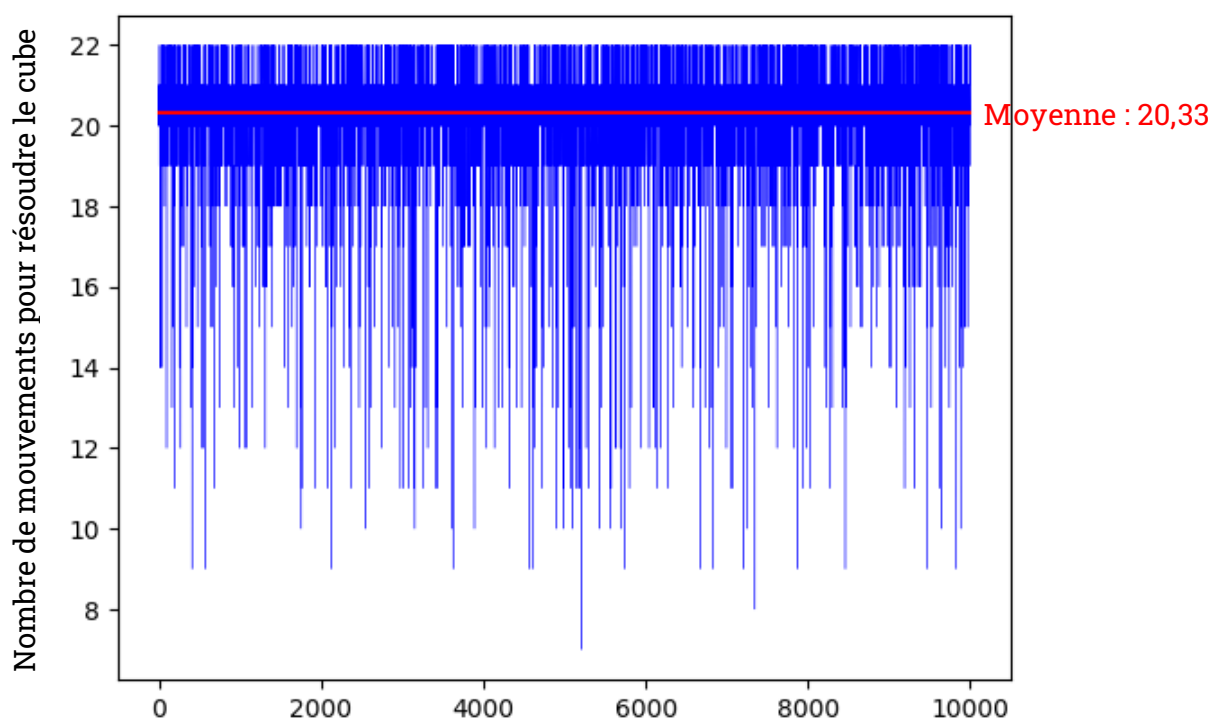
Nous avons tout de même réussi à entraîner un modèle, qui résolvait un Rubik's Cube, mais nous ne sommes pas arrivés à l'implémenter dans le projet, et à extraire la formule utilisée. En effet, le principal défaut des algorithmes utilisant les réseaux de neurone est leur opacité.

3. L'algorithme de Kociemba

Nous avons donc opté pour l'utilisation de l'algorithme à deux phases développé par Herbert Kociemba, chercheur allemand.

Cet algorithme utilise la théorie mathématique des groupes pour résoudre un cube très rapidement. Nous l'avons implémenté à l'aide de la bibliothèque `kociemba`.

Sur un échantillon de 10 000 cubes mélangés aléatoirement, la bibliothèque le résout en 20 mouvements en moyenne :



4. Le nombre de Dieu et mélange

Depuis 40 ans, la recherche du « nombre de Dieu » du Rubik's Cube a fait l'objet de nombreuses recherches. Ce nombre désigne le nombre de mouvements maximal pour résoudre n'importe quel des 43 252 003 274 489 856 000 cubes possibles.

Voici une brève histoire de la recherche du « nombre de Dieu » :

Date	Borne inférieure	Borne supérieure	Ecart	Chercheur(s)
Juillet 1981	18	52	34	Morwen Thistlethwaite
Décembre 1990	18	42	24	Hans Kloosterman
Mai 1992	18	39	21	Michael Reid
Mai 1992	18	37	19	Dik Winter
Janvier 1995	18	29	11	Michael Reid (par analyse de l'algorithme de Kociemba)
Janvier 1995	20	29	9	Michael Reid
Décembre 2005	20	28	8	Silviu Radu
Avril 2006	20	27	7	Silviu Radu
Mai 2007	20	26	6	Dan Kunkle et Gene Cooperman
Mars 2008	20	25	5	Tomas Rokicki
Avril 2008	20	23	3	Tomas Rokicki et John Welborn
Août 2008	20	22	2	Tomas Rokicki et John Welborn
Juillet 2010	20	20	0	Tomas Rokicki, Herbert Kociemba, Morley Davidson, et John Dethridge. Utilisation de la puissance de calcul d'ordinateurs de Google pour résoudre chaque position.

Source : <http://www.cube20.org/> (traduit de l'anglais), Thomas Rokicki, Herbert Kociemba, Morley Davidson, et John Dethridge.

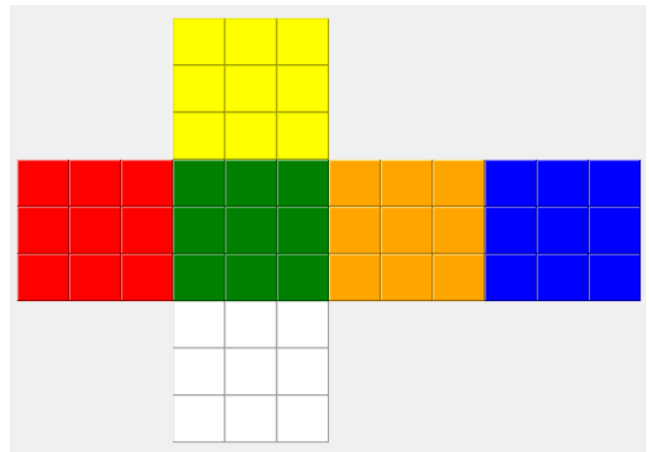
Le « God's number » du Rubik's Cube est donc de 20. Chaque position du cube requiert donc 20 mouvements ou moins pour être résolue.

Lorsque nous mélangeons aléatoirement notre Rubik's Cube, nous utilisons ainsi 24 mouvements tirés aléatoirement (bibliothèque random) : il serait inutile d'en effectuer plus, chaque cube pouvant être résolu en 20 mouvements !

D. Représentation informatique en deux dimensions et matrices

Pour représenter un Rubik's Cube, nous l'avons projeté en deux dimensions de la manière montrée ci-contre.

Nous avons ensuite associé à chaque couleur, une lettre : y, r, g, o, b ou w pour jaune, rouge, vert, orange, bleu, blanc respectivement.



```
[[['y', 'y', 'y'],
  ['y', 'y', 'y'],
  ['y', 'y', 'y']],
 [['r', 'r', 'r'],
  ['r', 'r', 'r'],
  ['r', 'r', 'r']],
 [['g', 'g', 'g'],
  ['g', 'g', 'g'],
  ['g', 'g', 'g']],
 [['o', 'o', 'o'],
  ['o', 'o', 'o'],
  ['o', 'o', 'o']],
 [['b', 'b', 'b'],
  ['b', 'b', 'b'],
  ['b', 'b', 'b']],
 [['w', 'w', 'w'],
  ['w', 'w', 'w'],
  ['w', 'w', 'w']]]
```

Nous avons

représenté chaque face du cube par une matrice carrée d'ordre 3, soit par un tableau à deux dimensions de 3 lignes et 3 colonnes, constitué de caractères.

Nous avons ensuite construit un tableau des 6 tableaux associés à chaque face dans l'ordre suivant : jaune, rouge, vert, orange, bleu, blanc.

← *La représentation du cube résolu en Python.*

La bibliothèque `pycuber` nous a ensuite permis d'appliquer des formules à un cube, et nous avons créé plusieurs fonctions permettant d'échanger avec cette bibliothèque, en la liant avec notre représentation en tableau de matrices.

II) Notice d'utilisation et démonstration

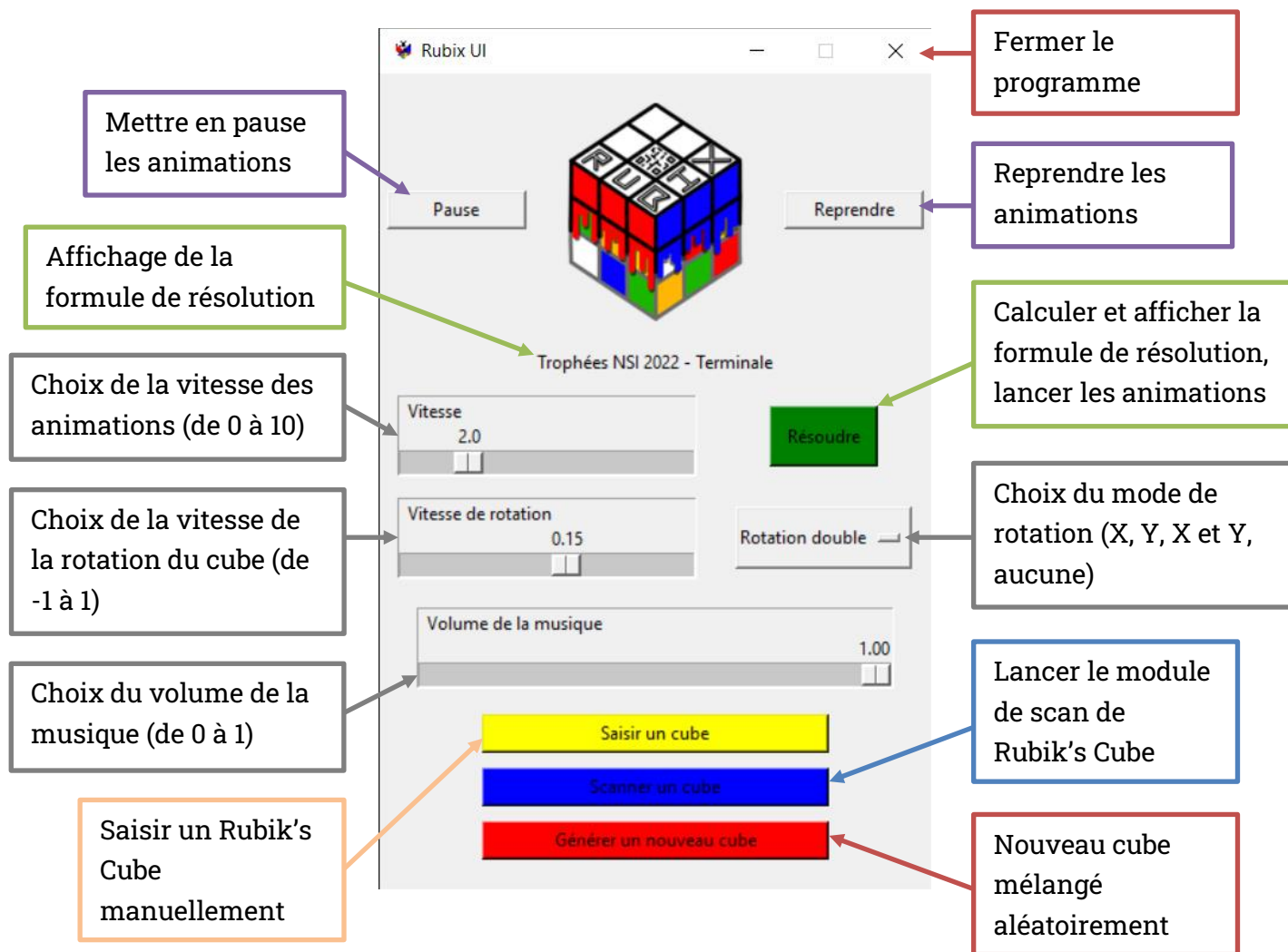
Les captures d'écran ont été réalisées sous Windows 10.

Pour lancer le programme, il suffit d'exécuter Python dans le dossier « sources » et lancer « `main.pyw` ».

A. Lancement du programme et présentation de l'interface graphique

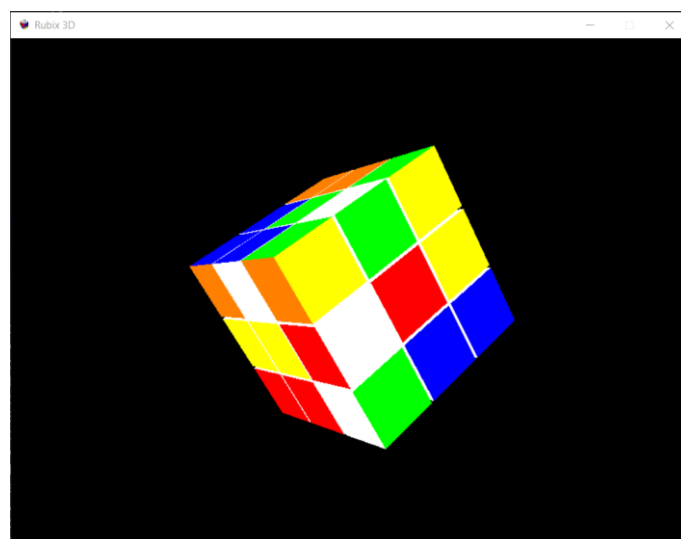
L'interface graphique « Rubix UI » se lance dès l'ouverture du programme et centralise les commandes entre les différentes fonctionnalités du programme : animation 3D, résolution, scan, musique. La fenêtre n'est pas redimensionnable et peut avoir un aspect différent selon le système d'exploitation utilisé.

Egalement, la fenêtre d'animation 3D « Rubix 3D » se lance dès l'ouverture du programme, et montre un cube non résolu généré aléatoirement qu'il est possible de résoudre, comme le montre le schéma à la page suivante.



B. Générer et résoudre un Rubik's Cube

Pour générer un Rubik's Cube mélangé de façon aléatoire, il suffit de cliquer sur le bouton rouge « Générer un nouveau cube », dans la fenêtre « Rubix UI ». Alors, le cube modélisé en 3D est modifié dans la fenêtre « Rubix 3D ».



Pour résoudre ce cube nouvellement généré, il suffit de cliquer sur le bouton vert « Résoudre » dans la fenêtre principale. La formule de résolution apparaît ainsi en dessous du logo du projet, et les animations débutent.

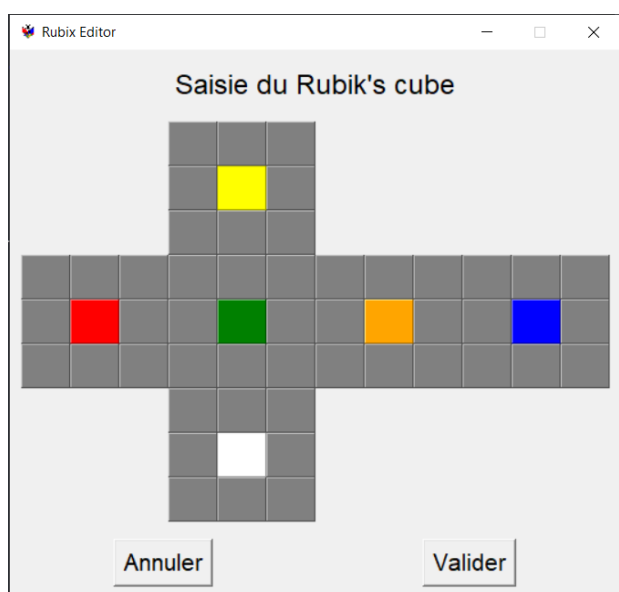
Vous pouvez choisir la vitesse des animations et de rotation.

De plus, la musique générée par la résolution du cube est jouée, selon la vitesse des animations et le volume demandé.

C. Saisir un Rubik's Cube

Vous pouvez, en cliquant sur le bouton jaune « Saisir un cube », configurer manuellement un Rubik's Cube que le programme doit résoudre.

Après avoir cliqué sur le bouton, une nouvelle fenêtre intitulée « Rubix Editor » s'ouvre. La fenêtre « Rubix 3D » se ferme.



Au centre de l'éditeur, vous pouvez choisir la couleur de chaque case grise en cliquant plusieurs fois dessus. Les centres des faces ne peuvent pas changer de couleur ; ils vous indiquent dans quel sens saisir votre cube.

Le bouton « Annuler » permet de revenir au menu principal et réouvre l'animation 3D.

Le bouton « Valider » permet de confirmer votre saisie et de vérifier son intégrité (si

le Rubik's Cube est solvable). Dans le cas contraire, il vous sera proposé de re-saisir le cube.

Si votre cube est valide, l'éditeur se ferme et l'animation 3D se réouvre avec votre cube modélisé. Vous pouvez ainsi le résoudre et accéder aux réglages vus précédemment.

D. Scanner un Rubik's Cube

Vous pouvez en cliquant sur le bouton bleu « Scanner un cube », lancer le module de scan. La fenêtre « Rubix 3D » se ferme, l'interface graphique se met en attente de scan.

Après d'éventuelles demandes d'autorisations pour accéder à la caméra, une nouvelle fenêtre « Rubix Scanner » se lance.

Vous pouvez à tout moment quitter le mode scan en appuyant sur la touche Q de votre clavier.

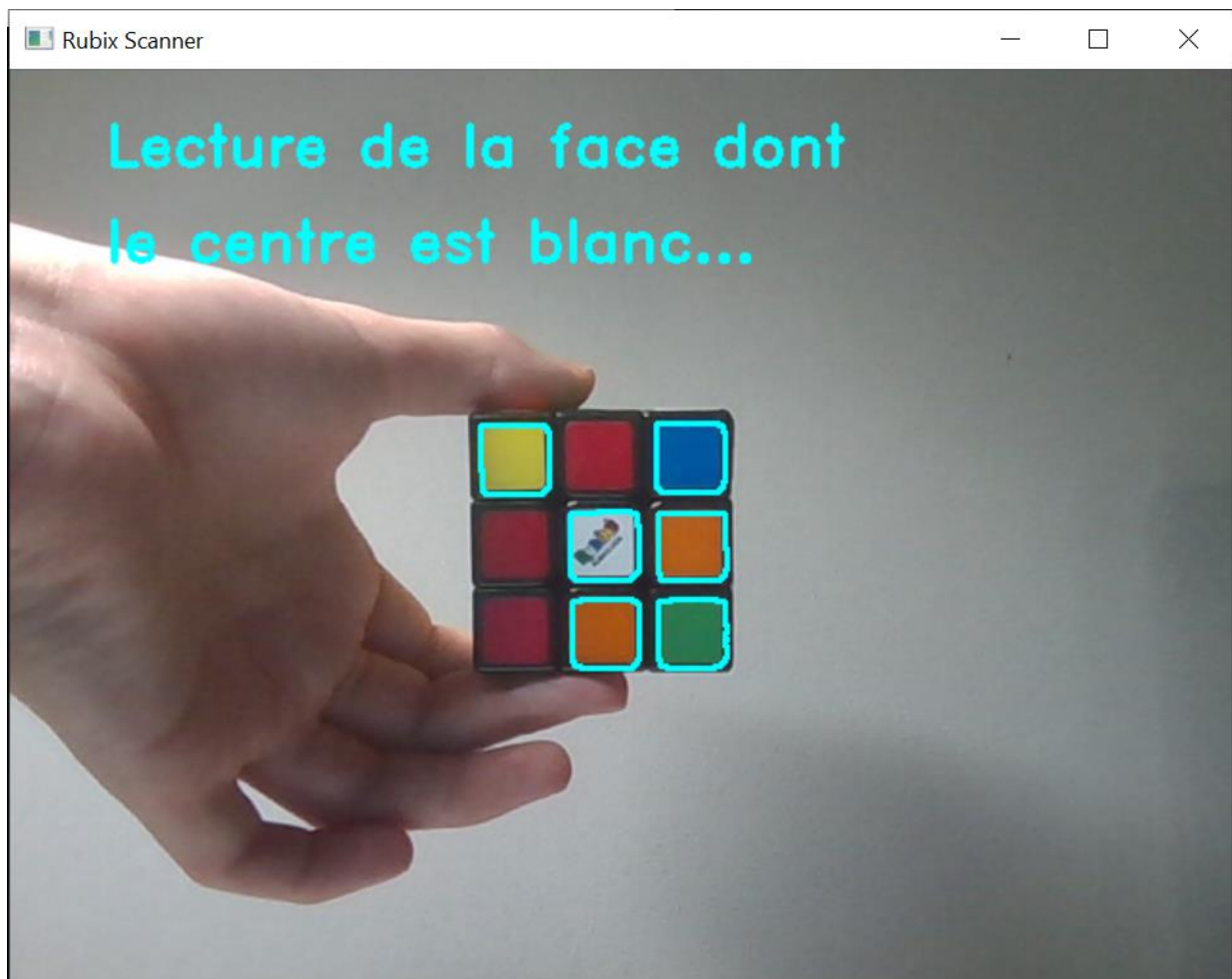
Nous vous conseillons de vous placer dans un environnement homogène, à lumière neutre. Votre Rubik's Cube doit comporter des bords noirs et les couleurs officielles : blanc, vert, rouge, bleu, orange, jaune.

Une première fenêtre vous indique comment positionner le Rubik's Cube initialement pour le scan.

La face blanche du cube doit se trouver face à la caméra. La face verte doit quant à elle se trouver en haut du cube (voir ci-contre)



Au bout de 8 secondes, le scan démarre. Les cases identifiées sont entourées de bleu turquoise.



Une fois la face entièrement détectée, une animation 3D vous montre comment tourner le cube vers la prochaine face.

Successivement et dans cet ordre, les faces scannées sont : blanche, verte, rouge, bleue, orange et jaune.

A la fin du scan, le cube scanné est analysé pour vérifier sa validité. S'il est invalide, une fenêtre de correction s'ouvre – sur le modèle du « Rubix Editor » - où vous pourrez corriger les erreurs. Les centres des faces sont automatiquement coloriés dans la bonne couleur et ne sont pas modifiables. *Se reporter au II/C si correction.*

III) Spécifications techniques

A. Structure et généralités

Langage utilisé : Python

Version : 3.10.2

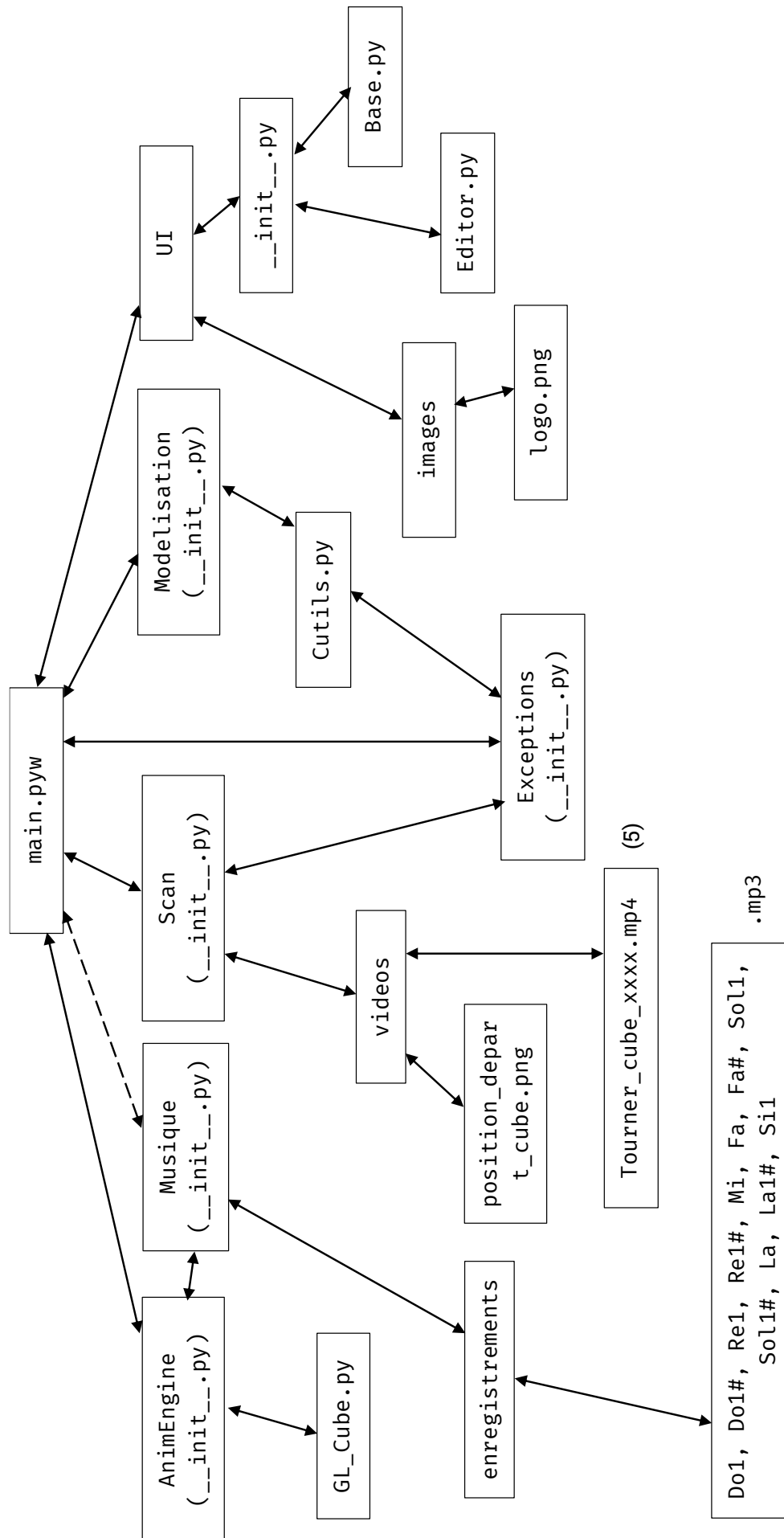
Bibliothèques utilisées :

Nom de la bibliothèque	Version utilisée	Installation avec PIP 22.0.4
numpy	1.22.2	<code>pip install numpy==1.22.2</code>
pygame	2.1.2	<code>pip install pygame==2.1.2</code>
tkinter	8.6	<i>Inclus avec Python 3.10.2</i>
PIL (Python Imaging Library)	9.1.0	<code>pip install Pillow==9.1.0</code>
PyOpenGL et PyOpenGL_accelerate	3.1.6	<code>pip install PyOpenGL==3.1.6 PyOpenGL_accelerate==3.1.6</code>
kociemba	1.2.1	<code>pip install kociemba==1.2.1</code>
pycuber	0.2.2	<code>pip install pycuber==0.2.2</code>
opencv-python	4.5.5.64	<code>pip install opencv-python==4.5.5.64</code>
scipy	1.8.0	<code>pip install scipy==1.8.0</code>
datetime		<i>Built-in Python 3.10.2</i>
random		<i>Built-in Python 3.10.2</i>
functools		<i>Built-in Python 3.10.2</i>
os		<i>Built-in Python 3.10.2</i>

Structure du projet :

Nous avons choisi de partitionner le code en plusieurs « modules » afin de le rendre plus lisible et modifiable par tous les membres du groupe. Il est ainsi plus adaptatif : modifier une fonction dans un module ne fait pas modifier tout le script.

Le fichier `main.pyw` met les modules en lien et contient le fil d'exécution principal. Il ne fait pas apparaître de console (extension `.pyw`).



AnimEngine : Regroupe les scripts nécessaires à l'affichage d'un Rubik's Cube en 3D.

- `__init__.py` : contient les fonctions nécessaires à l'intégration des mouvements, de l'affichage du cube et des entrées au clavier.
- `GL_Cube.py` : contient les éléments permettant le rendu d'un cube sur OpenGL, dont la couleur et le placement.

Musique : Regroupe les fichiers et scripts nécessaires à la diffusion de musique.

- `__init__.py` : contient les dictionnaires et fonctions permettant de jouer une note en fonction d'un mouvement du cube.
- `Répertoire enregistrements` : contient les notes de guitare enregistrées.

Scan (`__init__.py`) : Analyse l'entrée vidéo avec une reconnaissance d'objets et de couleurs pour scanner entièrement un cube.

Modélisation : Regroupe les scripts nécessaires à la modélisation d'un Rubik's Cube en programmation orientée objet.

- `__init__.py` : Classes `Cube` et `CubeByMatrix`, permettant de modéliser un Rubik's Cube.
- `Cutils.py` : Fonctions auxiliaires de traitement de matrices notamment.

UI : Regroupe les fichiers et scripts nécessaires à l'affichage d'une interface graphique.

- `__init__.py` : Importe les scripts `Base.py` et `Editor.py`.
- `Base.py` : Classe `App`, qui définit les propriétés, widgets de la fenêtre principale de commande.
- `Editor.py` : Classes `TopLevelApp` et `Case`, qui définissent la fenêtre d'édition du Rubik's Cube.
- `Répertoire images` : contient `logo.png`, le logo « Rubix » en grande taille.

Exceptions (`__init__.py`) : crée deux exceptions `CameraError` (erreur de lecture de vidéo) et `CombinaisonError` (Rubik's Cube impossible²), utilisées par les autres modules.

B. Notes musicales et mouvements

Les notes de musique sont jouées par le script `Musique/__init__.py`. Chacun des 12 mouvements simples énoncés dans la partie I/A est associé à une note, que nous avons préalablement enregistrée, à l'aide d'une guitare, d'un amplificateur et du logiciel Audacity.

Voici le tableau des correspondances :

² Voir plus de détails dans I/B.

Mouvement	Note associée
U	Do
D	Do#
L	Fa
R	Fa#
F	La
B	La#
U'	Mi
D'	Sol
L'	Re
R'	Re#
F'	Si
B'	Sol#

Toutes les notes sont dans la même gamme pour éviter les dissonances.

C. Documentation formelle des scripts, des classes et des fonctions

Fichier main.pyw

Modules importés: UI, Exceptions, AnimEngine, Modelisation, Scan, os, functools.

Constantes: NAME_INTERFACE, PATH_LOGO, DEFAULT_CUBE, G_path

Classes : Aucune.

Fonctions :

set_anim(anim_mode:str, anim_speed:float) -> None

Change le mode d'animation 3D du cube.

Paramètres : anim_mode (str) = mode de rotation (rotation double, rotation verticale, rotation horizontale, pas de rotation)
 anim_speed (float) = vitesse d'animation (de 0.0 à 10.0)

Retourne : rien.

set_res_speed(speed:float) -> None

Actualise la vitesse des animations.

Paramètres : speed (float) = vitesse souhaitée (de 0.0 à 10.0)

Retourne : rien.

set_volume(volume:float) -> None

Actualise le volume de la musique.

Paramètres : volume (float) = volume souhaité (de 0.0 à 1.0)

Retourne : rien.

new_cube() -> None

Crée un nouveau cube aléatoire (en variable globale) et l'affiche.

Paramètres : aucun.

Renvoie : rien.

solve() -> None

Résout le Rubik's Cube en variable globale. Vérifie sa validité.

Affiche la liste des mouvements ; lance les animations 3D ; et la musique.

Paramètres : aucun.

Retourne : rien.

pause() -> None

Mets en pause les animations 3D.

Paramètres : aucun.

Retourne : rien.

resume() -> None

Permet (après avoir mis les animations en pause) de reprendre l'animation 3D.

Paramètres : aucun.

Retourne : rien.

launch_camera() -> None

Lance l'outil de scan et scanne le cube.

Récupère le cube scanné et vérifie sa validité.

Paramètres : aucun.

Retourne : rien.

launch_editor(mtx:list[list[str]]=DEFAULT_CUBE) -> None

Lance l'éditeur de Rubik's Cube, et ferme l'animation 3D.

Paramètres : mtx (list[list[str]]) : représentation d'un cube en liste de matrices. Facultatif (uniquement si cube déjà saisi, à corriger)

Retourne : rien.

back_editor(editor:Editor.ToplevelApp) -> None

Ferme l'éditeur et relance l'animation 3D.

Paramètres : editor (Editor.ToplevelApp) = objet qui représente la fenêtre d'édition.

Retourne : rien.

confirm_editor(editor:Editor.ToplevelApp) -> None

Récupère la saisie de l'utilisateur dans l'éditeur de Rubik's cube.

Crée un nouveau cube.

Vérifie la validité du cube.

Affiche le cube dans l'interface graphique et ferme l'éditeur.

Paramètres : editor (Editor.TopLevelApp) = objet qui représente la fenêtre d'édition.

Retourne : rien.

stop_execution() -> None

Arrête l'exécution du programme.

Paramètres : aucun.

Retourne : rien.

Fichier AnimEngine/__init__.py

Modules importés : pygame, OpenGL, AnimEngine.GL_Cube, Musique

Constantes : rot_cube_map, rot_keys

Classes :

EntireCube(object)

Classe qui modélise un Rubik's Cube en 3D.

__init__(self, scale:float) -> None

Instancie un objet EntireCube.

Paramètres : scale (float) = taille dans l'espace 3D.

Retourne : rien.

rotation(self, rotation_key:str) -> int

Effectue une rotation de la représentation 3D du Rubik's Cube.

Tableau des rotations :

up, down, left, right, ur, ul, dr, dl, stop

Paramètres : rotation_key (str) = nom de la rotation.

Retourne : (int) = 0 si l'action a été enregistrée ; -1 si le mouvement est invalide.

move_slice(self, mouvement:str, animate_speed:int=5, debug:bool=False) -> str|int

Fonction asynchrone.

Effectue un mouvement d'une tranche sur la représentation 3D du Rubik's Cube.

Tableau des mouvements :

L,L',D,D',M,M',E,E',R,R',U,U',B,B',S,S',F,F'

Paramètres : mouvement (str) = Nom du mouvement
animate_speed (int) = Vitesse du mouvement
debug (bool) = Affiche le mouvement réalisé
Retourne : (str|int) = mouvement (str), si l'action a été effectué. -1
(int) si une action est déjà en cours, ou le mouvement est
invalide

set_camera_mode(self, mode:str) -> None

Change le mode de contrôle de la caméra.

Modes disponibles : "manual" (touches clavier), "static" (aucun mouvement),
"animation" (mode animation)

Paramètres : mode (str) = Mode de la caméra.

Retourne : rien.

set_animation_mode(self, mode:int, speed:float=2.0) -> None

Change l'animation de caméra.

Paramètres : mode (int) = 0=static, 1=x, 2=y, 3=xy, 4=xyz, 5=z
speed (float) = Vitesse (entre 0.0 et 10.0)s

Retourne : rien.

**play_sequence(self, sequence:list[str], anim_speed:int=2,
awaited:bool=True, debug:bool=False, enable_sound:bool=False)
-> int**

Effectue tous les mouvements dans la liste "sequence" à la suite sur la
représentation 3D.

Paramètres : sequence (list[str]) = Chaîne de mouvements à
réaliser
anim_speed (int) = Vitesse d'animation
awaited (bool) = Détermine si l'exécution doit être bloqué
jusqu'à la terminaison de la fonction
debug (bool) = Affiche le mouvement réalisé
enable_sound (bool) = Joue le son correspondant au
mouvement

Retourne : (int) = 0 si l'action a été effectuée, -1 sinon.

update_camera_anim(self) -> None

Mise à jour du mode de caméra (interne)

Paramètres : aucun.

Retourne : rien.

update_async_slices(self) -> None

Mise à jour des mouvements asynchrones

Paramètres : aucun.

Retourne : rien.

get_cube_by_location(self, location:tuple[int]) -> Cube
Retourne un cube selon la position.

Paramètres : location (tuple[int]) = couple position dans le Rubik's
Cube

Retourne : (GL_Cube.Cube) = objet 3D correspondant au Cube.

set_entire_color(self, color:tuple[float|int]) -> None
Colore entièrement le cube.

Paramètres : color (tuple[float|int]) = 0 <= RGB <= 1

Retourne : rien.

get_side(self, side:str) -> tuple[list|int]

Renvoie la liste des cubes et un identifiant selon la face choisi.

Paramètres : side (str) = Face à renvoyer (L-R-F-B-U-D)

Retourne : (list, int) = liste des cubes, identifiant

color_entire_face(self, side:str, color:tuple[float] =
(0.0,1.0,0.0)) -> None

Colore une face du cube avec une seule couleur.

Paramètres : side (str) = U <=> Dessus, L <=> Gauche, R <=> Droite, B <=>
Arrière, F <=> Avant, D <=> Dessous
color (tuple[float]) = 0 <= RGB <= 1

Retourne : rien.

color_face(self, side:str, color:list[tuple[float|int]] =
[(0,1,0),(0,1,0),(0,1,0),(0,1,0),(0,1,0),(0,1,0),(0,1,0),(0,1,0),(0,1,0)]) -> None

Colore une face selon une matrice (liste de tuples ici).

Paramètres : side (str) = U <=> Dessus, L <=> Gauche, R <=> Droite, B <=>
Arrière, F <=> Avant, D <=> Dessous
color (list[tuple[float|int]]) = 0 <= RGB <= 1

Retourne : rien.

load_color_matrix(self, matrix:list[list[str]]) -> None
Lecture d'une matrice représentant un cube.

Paramètres : matrix (list[list[str]]) = Matrice à lire

Retourne : rien.

set_camera_rotation(self, x_rot:float, y_rot:float,
z_rot:float = 0) -> None
Mets la rotation de la caméra.

Paramètres : x_rot (float) = Rotation sur l'axe x
 y_rot (float) = Rotation sur l'axe y
 z_rot (float) = Rotation sur l'axe z (optionnel)
Retourne : rien.

tick(self) -> None
Rafraichissement des fonctions (animations, mouvements...) du cube.
Paramètres : aucun.
Retourne : rien.

Fonctions :

init(Debug_:bool=False, fps_:int=60) -> EntireCube
Initialise la représentation 3D.

Paramètres : Debug (bool) = Option de débog (inactif par défaut)
 fps_ (int) = Limite de fps (60 par défaut)
Retourne : Rubix (EntireCube) = Objet de la représentation 3D initialisé,
 instance de EntireCube

update(Rubix:EntireCube) -> None
Mise à jour du rendu pour chaque image (actualise Rubix, pygame, et touches de debug si actif)

Paramètres : Rubix (EntireCube) = Objet EntireCube.
Retourne : rien.

stop() -> None
Arrête l'exécution de pygame.

Paramètres : aucun.
Retourne : rien.

initInterface(path_logo:str, name:str) -> None
Initialisation des données de l'interface de pygame.

Paramètres : path_logo (str) = Chemin du logo.
 name (str) = nom de la fenêtre.
Retourne : rien.

main() -> None
Initialisation de pygame et OpenGL.

Paramètres : aucun.
Retourne : rien.

Fichier AnimEngine/GL_Cube.py

Modules importés : OpenGL

Constantes : edges, surfaces, vertices, colors

Classes :

Cube(object)

Classe modélisant un cube unique en 3D.

__init__(self, id:int, scale:float) -> None

Instancie un objet Cube.

Paramètres : id (int) = identifiant du cube.
 scale (float) = taille dans l'espace 3D.

Retourne : rien.

isAffected(self, axis:int, slice:int) -> None

Renvoie True si le cube est affecté à un axe

Paramètres : axis (int) = axe.
 slice (int) = tranche en animation.

Retourne : (bool) = True si le cube est affecté à un axe, False sinon.

get_position(self) -> list

Renvoie l'index position du cube.

Paramètres : aucun.

Retourne : (list) = index position du cube.

update(self, axis:int, slice:int, dir:int) -> None

Mise à jour du cube et rotation.

Paramètres : axis (int) = axe de rotation
 slice (int) = tranche en animation
 dir (int) = sens de rotation (1 ou -1)

Retourne : rien.

transformMat(self) -> list

Rotation du cube.

Paramètres : aucun.

Retourne : (list) = matrice des transformations

set_color(self, _colors:list[tuple[float|int]]) -> None

Change la couleur d'un cube.

Paramètres : _colors (list[tuple[float|int]]) = liste des tuples
 des nouvelles couleurs. Exemple : ((1, 0, 0), (0, 1,

0), (1, 0.5, 0), (1, 1, 0), (1, 1, 1), (0, 0, 1))

Retourne : rien.

get_color(self) -> tuple[tuple]

Renvoie le tuple couleur du cube

Paramètres : aucun.

Retourne : (tuple[tuple]) = couleurs du cube.

draw(self, surf:tuple[tuple], animate:bool, angle:float, axis:int, slice:int, dir:int) -> None

Dessine le cube avec OpenGL.

Paramètres : surf (tuple[tuple]) = surfaces du cube
animate (bool) = rotation du cube
angle (float) = angle de rotation d'une tranche
axis (int) = axe de rotation
slice (int) = tranche en animation
dir (int) = sens de rotation (1 ou -1)

Retourne : rien.

Fonctions : aucune.

Fichier Musique/__init__.py

Modules importés : pygame, os

Constantes : G_path, correspondances

Classes : aucune.

Fonctions :

play_mvt(mvt_rubix:str, volume:float=1.0) -> None

Joue un son correspondant à un mouvement.

Paramètres : mvt_rubix (str) = caractères représentant un mouvement de Rubik's Cube.
volume (float) = volume du son (entre 0.0 et 1.0)

Retourne : rien.

Fichier Scan/__init__.py

Modules importés : Exceptions, cv2, datetime, numpy, scipy.spatial, os

Constantes: G_path, SECONDS_WAITING, MARGE_PIXEL, WINDOW_NAME, PATH_LOGO

Classes : aucune.

Fonctions :

read_camera(video:cv2.VideoCapture) -> numpy.ndarray

Lit la vidéo renvoyée par la caméra.

Paramètres: video (cv2.VideoCapture) = objet représentant l'entrée video.

Retourne: (numpy.ndarray) = représentation matricielle de l'image obtenue.

conditions_ok(contour:numpy.ndarray) -> bool

Vérifie, pour un contour, s'il peut correspondre à une case de Rubik's Cube.

Paramètres: contour (numpy.ndarray) = contour openCV

Retourne: (bool) = True si le contour correspond, False sinon.

to_int_list(obj:list|tuple|dict|set) -> list[int]

Convertit un itérable en liste d'entiers, si possible (lève une ValueError sinon).

Paramètres: obj (iterable) = itérable à convertir.

Retourne: (list[int]) = une liste d'entiers.

calcul_couleur(image:numpy.ndarray, x:float, y:float, w:float, h:float) -> list

Calcule la couleur moyenne d'une case.

Paramètres: obj (iterable) = itérable à convertir.

x (float) = l'abscisse de la case

y (float) = l'ordonnée de la case

w (float) = la longueur de la case

h (float) = la hauteur de la case

Retourne: (list) = liste contenant :

- (int) = valeur moyenne HUE

- (int) = valeur moyenne SATURATION

- (int) = valeur moyenne VALUE

- (float) = une valeur indicatrice de l'emplacement de la case

to_matrix(l:list) -> list[list]

Convertit une liste de 9 éléments en matrice de 3x3.

Paramètres: l (list) = liste de 9 éléments

Retourne: (list) = matrice de 3x3

association_couleurs(r:int, g:int, b:int) -> str

Associe une couleur à une case.

Paramètres : r (int) = Red (entre 0 et 255)
 g (int) = Green (entre 0 et 255)
 b (int) = Blue (entre 0 et 255)

Retourne : (str) = caractère représentant la couleur de la case : r, b, g, y, o ou w.

analyze_face(video:cv2.VideoCapture, text1:str=" ", text2:str=" ")
 -> list[list]

Détecte, analyse et modélise une face du cube.

Paramètres : video (cv2.VideoCapture) = objet représentant l'entrée video.
 text1 (str) = premier texte à afficher.
 text2 (str) = second texte à afficher.

Retourne : (list) = matrice 3x3 représentant une face.

wait_with_video(video:cv2.VideoCapture, vid:str, tstamp:int=8,
text1:str=" ", text2:str=" ") -> None

Fait patienter le scan en affichant un texte, en gardant la caméra visible.

Paramètres : video (cv2.VideoCapture) = objet représentant l'entrée de la
 caméra
 vid (str) = chemin d'accès à la vidéo.
 tstamp (int) = nombre de secondes d'attente.
 text1 (str) = premier texte à afficher.
 text2 (str) = second texte à afficher.

Retourne : rien.

wait_with_image(video:cv2.VideoCapture, img:str, tstamp:int=8,
text1:str=" ", text2:str=" ") -> None

Fait patienter le scan en affichant un texte et une image.

Paramètres : video (cv2.VideoCapture) = objet représentant l'entrée de la
 caméra
 img (str) = chemin d'accès à l'image.
 tstamp (int) = nombre de secondes d'attente.
 text1 (str) = premier texte à afficher.
 text2 (str) = second texte à afficher.

Retourne : rien.

rot90_anticlockwise(m:list[list]) -> list[list]

Effectue une rotation à 90 degrés (dans le sens inverse des aiguilles d'une montre)
d'une matrice 3x3.

Paramètres : m (list[list]) = matrice 3x3.

Retourne : (list[list]) = matrice 3x3, rotation effectuées.

main() -> None

Exécution du processus de scan.

Paramètres : aucun.

Retourne : rien.

initInterface(path_logo:str, name:str) -> None

Initialise les constantes de l'interface.

Paramètres : path_logo (str) = chemin d'accès au logo de Rubix.
 name (str) = nom de la fenêtre.

Retourne : rien.

Fichier Modelisation/__init__.py

Modules importés : Modelisation.Cutils, pycuber

Constantes : aucune.

Classes :

Cube(pycuber.Cube)

Classe modélisant un cube, crée avec la librairie pycuber.

Hérite de pycuber.Cube.

__init__(self) -> None

Instancie un objet Cube.

Paramètres : aucun.

Retourne : rien.

scramble(self) -> None

Mélange le cube

Paramètres : aucun.

Retourne : rien.

apply(self, formula:str|pc.Formula) -> None

Applique une formule au cube ; permet + de cohérence dans notation.

Paramètres : formula (str) = formule aux conventions internationales.

Retourne : rien.

get_matrix(self) -> list

Obtient la matrice représentant le cube.

Paramètres : aucun.

Retourne : (list) représentation en liste de matrices du cube.

get_formula_to_solve(self) -> str

Obtient la formule permettant de résoudre le cube.

Paramètres : aucun.

Retourne : (str) formule permettant de résoudre le cube.

CubeByMatrix(object)

Classe modélisant un cube, crée avec une représentation en liste de matrices.

__init__(self, matrice) -> None

Instancie un objet CubeByMatrix.

Paramètres : matrice (list) = représentation en liste de matrices du cube.

Retourne : rien.

print(self) -> None

Affiche le cube.

Paramètres : aucun.

Retourne : rien.

get_matrix(self) -> list

Obtient la matrice représentant le cube.

Paramètres : aucun.

Retourne : (list) représentation en liste de matrices du cube.

get_formula_to_solve(self) -> str

Obtient la formule permettant de résoudre le cube.

Paramètres : aucun.

Retourne : (str) formule permettant de résoudre le cube.

Fonctions : aucune.

Fichier Modelisation/Cutils.py

Modules importés : Exceptions, pycuber, random, kociemba

Constantes : ACTION_LIST

Classes : aucune.

Fonctions :

gen_formula(nb_step:int=24) -> pc.Formula

Génère une formule de mélange du cube aléatoire.

Paramètres : nb_step (int) = nombre d'étapes de mélange

Retourne : (pc.Formula) = formule de mélange

convert_to_matrix(cube:pycuber.Cube) -> list

Convertit un cube en matrice.

Paramètres : cube (pycuber.Cube) = objet cube (comprend celui du fichier __init__.py par héritage)

Retourne : (list) = représentation en liste de matrices d'un cube.

copy_3by3(mat:list) -> list

Copie profonde d'une liste de matrices carrées d'ordre 3.

Paramètres : mat (list) = liste de matrices carrées d'ordre 3.

Retourne : (list) = copie de mat.

get_value(square:pycuber.Square) -> str

Retourne la couleur d'une case modélisée par pycuber.

Paramètres : square (pycuber.Square) = case pycuber.

Retourne : (str) = caractère représentant la couleur de la case.

get_k_string(m:list) -> str

Transforme la représentation en matrice d'un cube en chaîne de caractères compatible avec la librairie kociemba.

Paramètres : m (list) = représentation en liste de matrices d'un cube

Retourne : (str) = chaîne de représentation d'un cube avec kociemba.

get_formula_to_solve(mtx:list) -> str

Obtient la formule de résolution d'un cube.

Paramètres : mtx (list) = représentation en liste de matrices d'un cube

Retourne : (str) = formule de résolution

Fichier UI/Base.py

Modules importés : tkinter, PIL

Constantes : SIZE_LOGO

Classes :

App(tkinter.Tk)

Classe modélisant l'UI de Rubix.

Hérite de tkinter.Tk (UI faite avec tkinter).

```
__init__(self, path_logo:str, win_title:str,  
win_resizable:bool=False, camera_anim_opt:list=["Rotation  
double", "Rotation triple", "Rotation horizontale", "Rotation  
verticale", "Rotation pivot", "Pas de rotation"]) -> None
```

Instancie un objet App.

Paramètres : path_logo (str) = chemin d'accès au logo de Rubix.
 win_title (str) = nom de la fenêtre.
 win_resizable (bool) = True si fenêtre
 redimensionnable, False sinon.
 camera_anim_opt (list) = Liste des modes d'animations
 possibles.

Retourne : rien.

```
wait(self) -> None
```

Affiche un message d'attente pendant le scan par caméra.

Paramètres : aucun.

Retourne : rien.

```
go(self) -> None
```

Affiche l'UI et enlève le message d'attente.

Paramètres : aucun.

Retourne : rien.

```
show_error(self, text:str) -> None
```

Affiche un message d'erreur.

Paramètres : text (str) = texte du message

Retourne : rien.

```
show_warning(self, text:str) -> None
```

Affiche un message d'avertissement.

Paramètres : text (str) = texte du message

Retourne : rien.

```
show_formule_resolution(self, string:str) -> None
```

Modifie le texte "Trophées NSI 2022 - Projet Rubix" avec la formule de résolution.

Paramètres : string (str) = formule de résolution

Retourne : rien.

```
disable_resolve(self) -> None:
```

Désactive le bouton de résolution.

Paramètres : aucun.

Retourne : rien.

enable_resolve(self) -> None

Active le bouton de résolution.

Paramètres : aucun.

Retourne : rien.

Fonctions : aucune.

Fichier UI/Editor.py

Modules importés:tkinter, functools

Constantes:decal_row, decal_colonne, to_color, L

Classes :

ToplevelApp(tkinter.Toplevel)

Classe modélisant l'éditeur de Rubik's cube.

Hérite de tkinter.Toplevel (fenêtre additionnelle à tkinter.Tk).

__init__(self, master:tkinter.Tk, name:str, mode:str) -> None

Instancie un objet ToplevelApp.

Paramètres : master (tkinter.Tk) = application/fenêtre tkinter de
référence

name (str) = nom de la fenêtre

mode (str) = mode de l'éditeur ("Correction" ou "Saisie")

Retourne : rien.

show(self, mtx:list) -> None

Affiche le contenu de la fenêtre.

Paramètres : mtx (list) = représentation d'un cube en liste de matrices.

Retourne : rien.

Case(tkinter.Button)

Classe modélisant une case de Rubik's Cube comme un bouton.

Hérite de tkinter.Button.

__init__(self, master:Frame, color:str) -> None

Instancie un objet Case.

Paramètres : master (tkinter.Frame) = frame tkinter où on place le
bouton.

color (str) = couleur du bouton (red, blue, yellow, ...)

Retourne : rien.

change_color(self, c) -> None

Change la couleur du bouton.

Paramètres : c (str) = couleur actuelle du bouton.

Retourne : rien.

disable(self) -> None

Désactive la case : couleur non-changeable.

Paramètres : aucun.

Retourne : rien.

Functions : aucune.

Fichier Exceptions/__init__.py

Modules importés : aucun.

Constantes : aucun.

Classes :

CameraError(Exception)

CombinaisonError(Exception)

Fonctions : aucune.