Stereovision

Controle 3D par un Rubik's Cube

Yoann Bourse

2010-2011 : Semestre 1

Résumé

Le but de ce projet est d'orienter un objet en 3D à l'écran en manipulant un Rubik's Cube devant une webcam. Je développerai ici les différentes méthodes envisagées, la réflexion sous-jacente, la solution implémentée ainsi que les améliorations possibles du programme. La version actuelle détecte sur chaque image les segments significatifs par LSD, puis calcule les intersections de tous les segments parallèles avant de les regroupper par direction de fuite grâce à un RANSAC. L'algorithme conserve ainsi les 3 points de fuite les plus significatifs dont il se sert pour calculer K^-1 et determiner les vanishing points qu'il utilise ensuite pour imprimer un cube en 3D.
Note : les images et le Rubik's Cube sont ceux de Fabrice Ben Hamouda, je n'en possède pas personnellement. De plus, les vidéos de la webcam ont été enregistrées en .avi à cause d'un problème de compatibilité avec openCV.

Table des matières

Le travail a bien entendu commencé sur des images statiques prises à la webcam, la première étape étant de parvenir à détecter les arêtes du cube pour en trouver les points de fuite. J'ai tout d'abord essayé des méthodes originales pour isoler le centre des faces avant de revenir à la detection de segments plus conventionnelle.

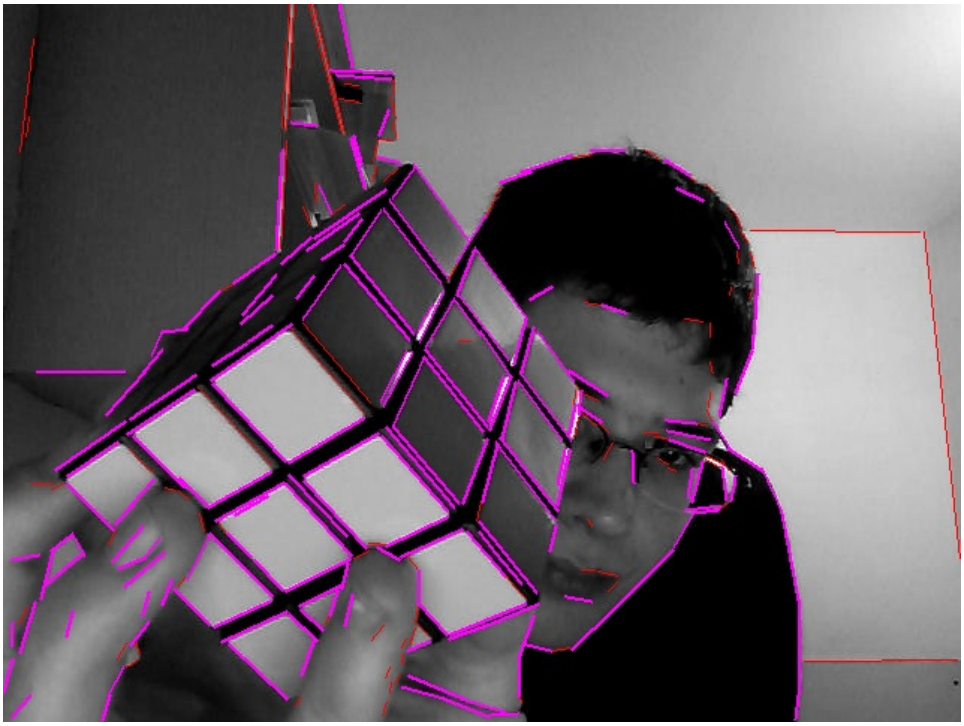
1 Detection des arêtes

1.1 Méthode de grossissement des contours

L'idée originelle pour retrouver les centres des faces était de grossir progressivement les contours afin de réduire les faces à quelques pixels plus aisément repérables qu'une face entière. L'uniformité du grossissement anisotrope permettrait de repérer de plus le centre de la face de manière assez précise. Après différentes méthodes d'expansion, j'ai abandonné cette idée dans la mesure où les parasites étaient beaucoup trop nombreux et peu discernables.

1.2 Détection de lignes

Je suis donc revenu à des méthodes moins originales de détection de lignes. Face au grand nombre de parasites donnés par la transformée de Hough, je me suis tourné vers l'algorithme Line Segment Detector (LSD). Les parasites étant toujours très nombreux, j'ai fait usage de méthodes de seuillage rudimentaires (sur la longueur des segments : je garde ceux de taille entre 15 et 70) pour diminuer le nombre segments considérés. Ces grossières éliminations permettent de réduire de moitié le nombre de segments considérés, qui tombe pour la plupart des images entre 100 et 200 (contre 200 à 300 sans seuillage). Voici un exemple graphique des segments gardés (en magenta) contre les segments éliminés (en rouge) :



1.3 Améliorations

De nombreux segments parasites subsistent et pourraient être éliminés sans difficultés majeures en essayant diverses méthodes de détection. Beaucoup d'informations sont ignorées (la couleur par exemple). On pourrait donc garder en priorité les traits séparant deux zones unies (faible gradient).
Toutefois, les méthodes de détection de point de fuite appliquées par la suite ne nécessitent pas une telle précision,

2 Détermination des points de fuite

J'ai ensuite utilisé les segments detectés et seuillés, qui contiennent donc un nombre conséquent de parasite, afin de déterminer les points de fuite qui sont les intersections des droites parallèles s'appuyant sur les arêtes du cube.

2.1 Etude des intersections des segments

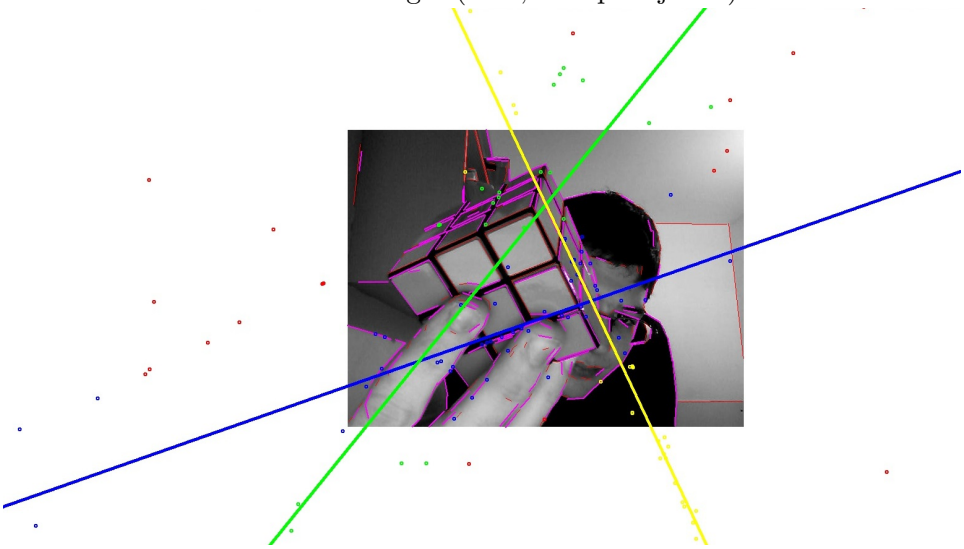
Pour cela, remarquant que ces arêtes sont constituées de nombreux petits fragments, j'ai considéré toutes les intersections de segments potentiellement parallèles : il s'agit d'effectuer une boucle quadratique sur le nombre de segments gardés en conservant les intersections avec d'autres segments qui lui sont parallèles et non confondus, ce que j'ai implémenté par un seuillage

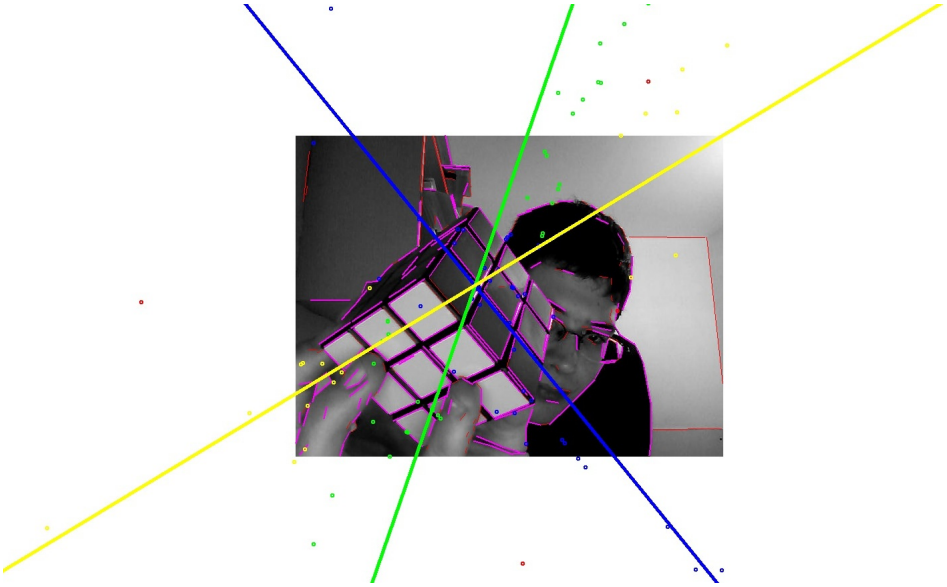
sur l'angle entre ces deux segments. On ne considère ainsi que les doites qui forment un angle suffisament proche de 0 en y restant quand même éloignées (entre 0.02 et 0.1 rad de différence).

On remarque comme prévu une accrétion dans la direction des arêtes du cube (points de fuite).

2.2 Determination de trois points fixes

Afin de garder les points les plus significatifs et de les résumer en une direction de fuite, j'ai effectué un RANSAC classique (en étudiant les droites passant par les points deux à deux et la distance d'un point à une droite). Les résultats sont extrêmement satisfaisants. Les points utilisés sont ensuite retirés de l'ensemble pour déterminer une deuxième direction de fuite par un deuxième RANSAC, puis un troisième, parfois plus hasardeux. Vous pouvez les voir dans l'ordre sur ces images (bleu, vert puis jaune) :





A partir de ces nuages de points associés à une direction de fuite, considérer comme point de fuite la moyenne des intersections s'est révélé inefficace : j'ai donc procédé à un second RANSAC, considérant cette fois-ci les points un par un et jugeant la distance entre deux points. Le résultat obtenu est très satisfaisant *(voir suite)*.
Il est à noter qu'une amélioration visant à tirer de nouveaux RANSAC dans les rares cas où les données parasite donnaient naissance à deux vanishing points dont les directions étaient parallèles a été implémentée (angle minimum entre les directions requis pour les conserver), mais n'a pas été conservée par manque de résultats significatifs.

3 Calcul de $K^{-}1$

Les trois points de fuite considérés nous permettent alors de récupérer la matrice $K^{-}1$. Approximant les pixels comme carrés, nous cherchons K sous la forme :

$$K = \begin{pmatrix} \alpha & 0 & u_0 \\ 0 & \alpha & v_0 \\ 0 & 0 & 1 \end{pmatrix}$$

L'image de la conique absolue IAC $\omega = (KK^t)^{-1}$ sera donc de la forme :

$$\omega = \begin{pmatrix} w_1 & 0 & w_3 \\ 0 & w_1 & w_5 \\ w_3 & w_5 & 1 \end{pmatrix}$$

Qui vérifie, avec les trois points M_i mesurés, $m_i^t \omega m_j = 0$. Pour résoudre ce

système, écrivons-le sous la forme $A\omega = B$: les équations donnent :

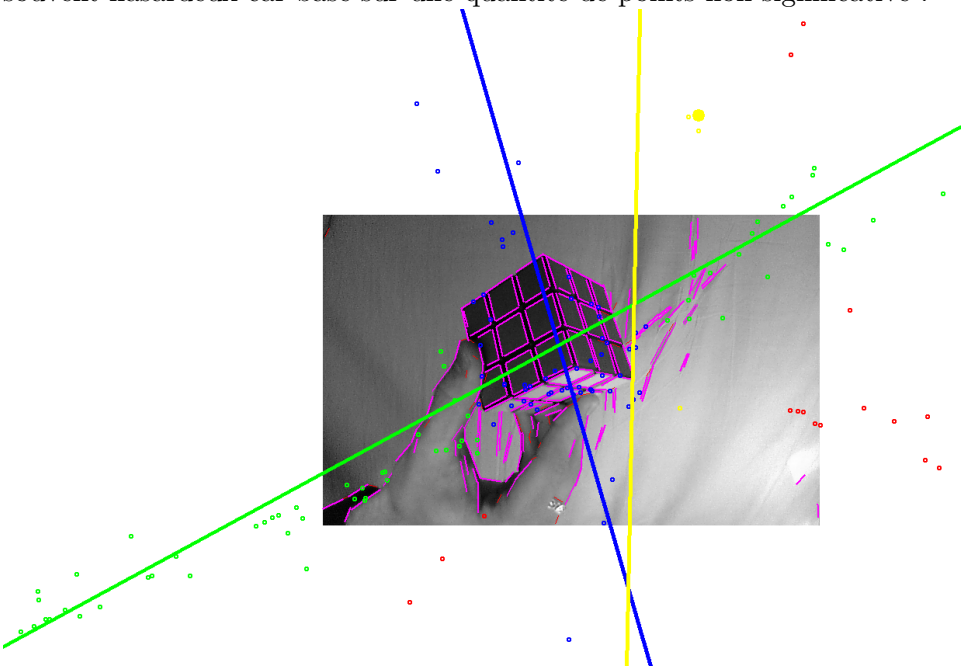
$$A\omega = \begin{pmatrix} m_1[1]m_2[1] + m_1[2]m_2[2] & m_1[1]m_2[3] + m_1[3]m_2[1] & m_1[2]m_2[3] + m_1[3]m_2[2] \\ m_3[1]m_2[1] + m_3[2]m_2[2] & m_3[1]m_2[3] + m_3[3]m_2[1] & m_3[2]m_2[3] + m_3[3]m_2[2] \\ m_1[1]m_3[1] + m_1[2]m_3[2] & m_1[1]m_3[3] + m_1[3]m_3[1] & m_1[2]m_3[3] + m_1[3]m_3[2] \end{pmatrix} \omega$$

$$A\omega = \begin{pmatrix} -m_1[3]m_2[3] \\ -m_3[3]m_2[3] \\ -m_1[3]m_3[3] \end{pmatrix} = B$$

On obtient donc $\omega = A^{-1}B$ d'où l'on peut déduire $K^{t^{-1}}$ par l'algorithme de Cholesky : $K_{it} = cholesky(W) = cholesky(\begin{pmatrix} \omega_0 & 0 & \omega_1 \\ 0 & \omega_0 & \omega_2 \\ \omega_1 & \omega_2 & 1 \end{pmatrix}$
On peut alors utiliser les vrais vanishing points qui forment une base $B = Ki * M$.

3.1 Stockage de K^{-1}

Toutefois, les données récupérées par la détection de segments sur les images sont fragmentaires, et le troisième vanishing point est relativement souvent hasardeux car basé sur une quantité de points non significative :



J'ai donc divisé le programme en deux parties distinctes : une calcule le K^{-1} de la webcam une bonne fois pour toutes(#computeK == 1) et l'enregistre dans K.txt, l'autre ne se servant que de cette donnée pour calculer la position du cube.
Cela permet une plus grande précision dans la détermination de K^{-1} en

cumulant les informations de diverses images, mais surtout de limiter le besoin de vanishing point par image pour retrouver le cube à 2 (le troisième leur étant orthonormal). La cohérence des résultats s'en trouve grandement augmentée.
Ce stockage de K^{-1} permet de n'utiliser que les deux vanishing point réellement significatifs mis en avant par le RANSAC et donc de diminuer grandement le nombre d'images incohérentes. K^{-1} est alors determiné par l'accrétion de données significatives selectionnées manuellement par l'équation :

$$A = \begin{pmatrix} \dots & \dots & \dots \\ m_1^k[1]m_2^k[1] + m_1^k[2]m_2^k[2] & m_1^k[1]m_2^k[3] + m_1^k[3]m_2^k[1] & m_1^k[2]m_2^k[3] + m_1^k[3]m_2^k[2] \\ m_3^k[1]m_2^k[1] + m_3^k[2]m_2^k[2] & m_3^k[1]m_2^k[3] + m_3^k[3]m_2^k[1] & m_3^k[2]m_2^k[3] + m_3^k[3]m_2^k[2] \\ m_1^k[1]m_3^k[1] + m_1^k[2]m_3^k[2] & m_1^k[1]m_3^k[3] + m_1^k[3]m_3^k[1] & m_1^k[2]m_3^k[3] + m_1^k[3]m_3^k[2] \\ \dots & \dots & \dots \end{pmatrix}$$

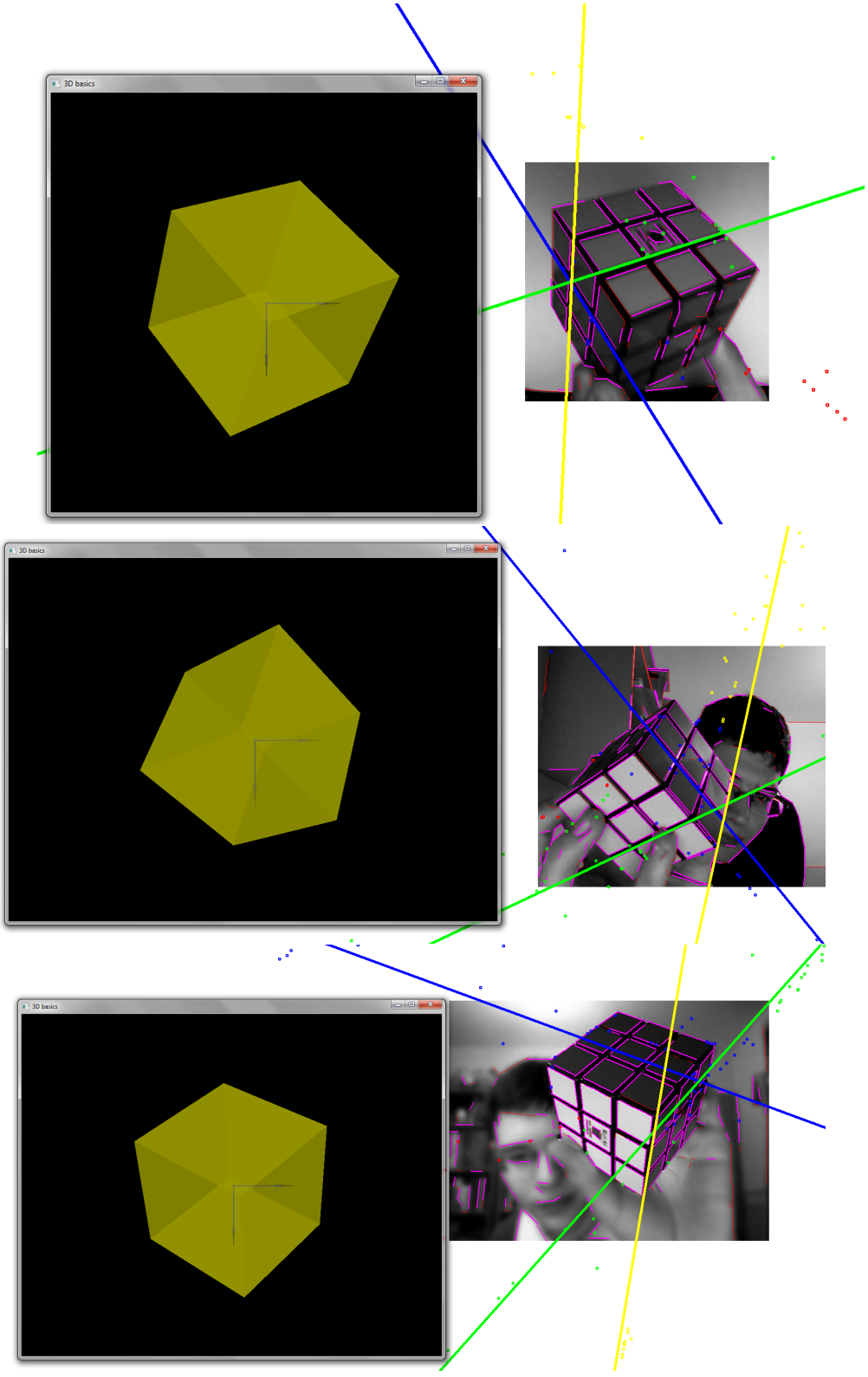
$$B = \begin{pmatrix} \dots \\ -m_1^k[3]m_2^k[3] \\ -m_3^k[3]m_2^k[3] \\ -m_1^k[3]m_3^k[3] \\ \dots \end{pmatrix}$$

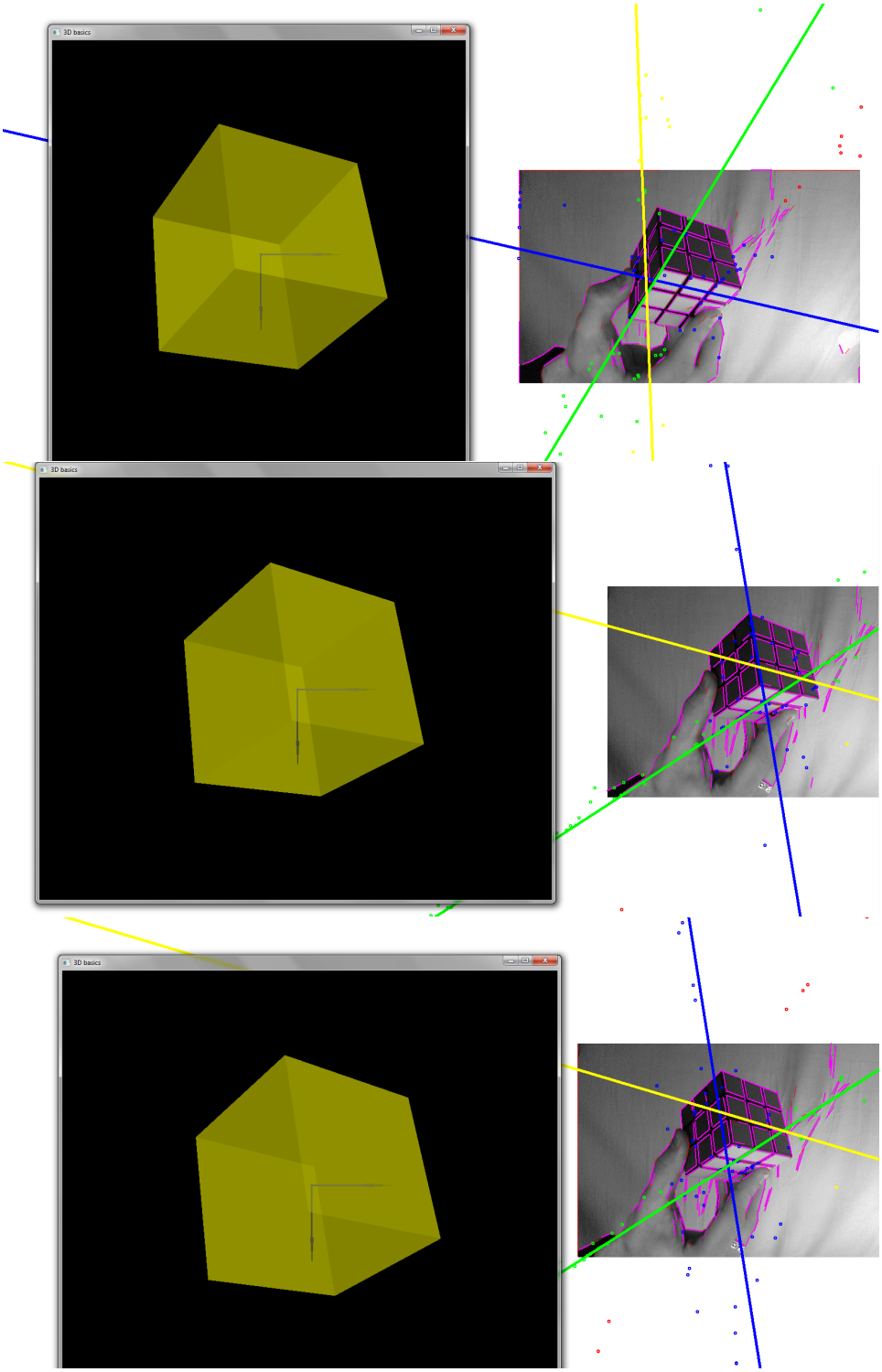
On obtient donc $\omega = pseudoinverse(A)B$ d'où l'on peut de nouveau déduire $K^{t^{-1}}$ par l'algorithme de Cholesky (voir section précédente). Pour ma webcam,

$$K^{-1} = \begin{pmatrix} \dots & & \\ 0.00183588 & 0 & -0.725645 \\ 0 & 0.00183588 & -0.369035 \\ 0 & 0 & 1 \dots \end{pmatrix}$$

4 Output

Après avoir essayé une représentation peu expressive sous forme d'angles d'Euler, le programme actuel présente en sortie un cube obtenu à partir des trois vecteurs de direction des vanishing points (en jaune) sur la base canonique de R3 (en bleu) :


3D basics
3D basics
3D basics

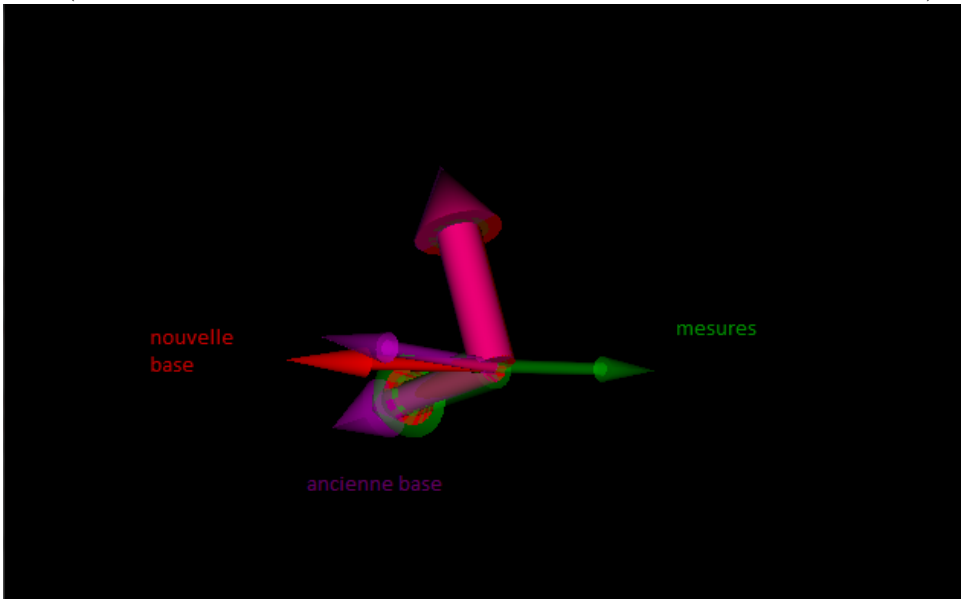

3D basics
3D basics
3D basics

5 Acquisition en temps réel

L'intégration de l'acquisition en temps réel pose le problème de la continuité de la représentation du cube, en particulier dans le cas d'informations non significatives ou manquantes. Il s'agit donc de maintenir une base de vecteurs et d'y autoriser des modifications suffisament petites en ignorant les états non décidés.

5.1 Ordonnancement et orientation

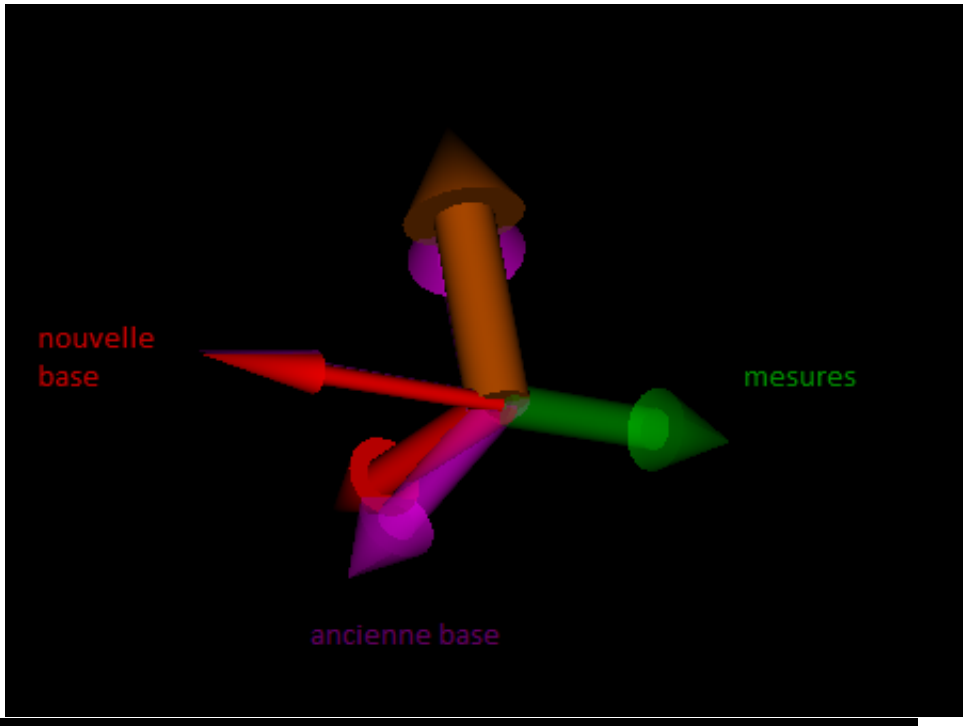
Il n'y a aucune raison que les detections de segments nous donne toujours les mêmes vecteurs de la base, ni conservent leur ordre. Il s'agit donc d'aparier les nouveaux vecteurs detectés avec les anciens. On remplacera donc chaque vecteur de l'ancienne base par le nouveau vecteur qui donne avec lui le plus grand produit scalaire (plus faible angle, puisqu'ils sont normalisés). Cependant, il faut aussi considérer l'opposé du nouveau vecteur : l'orientation n'a aucune raison d'être identique. Une fois les paires formées, il s'agit donc de rétablir l'orientation de la base précédente, en inversant les vecteurs de la nouvelle base qui donnent un produit scalaire négatif avec le vecteur qu'ils remplacent. Le résultat peut être représenté comme suit (l'épaisseur des vecteurs correspondant à leur ordre dans la base) :

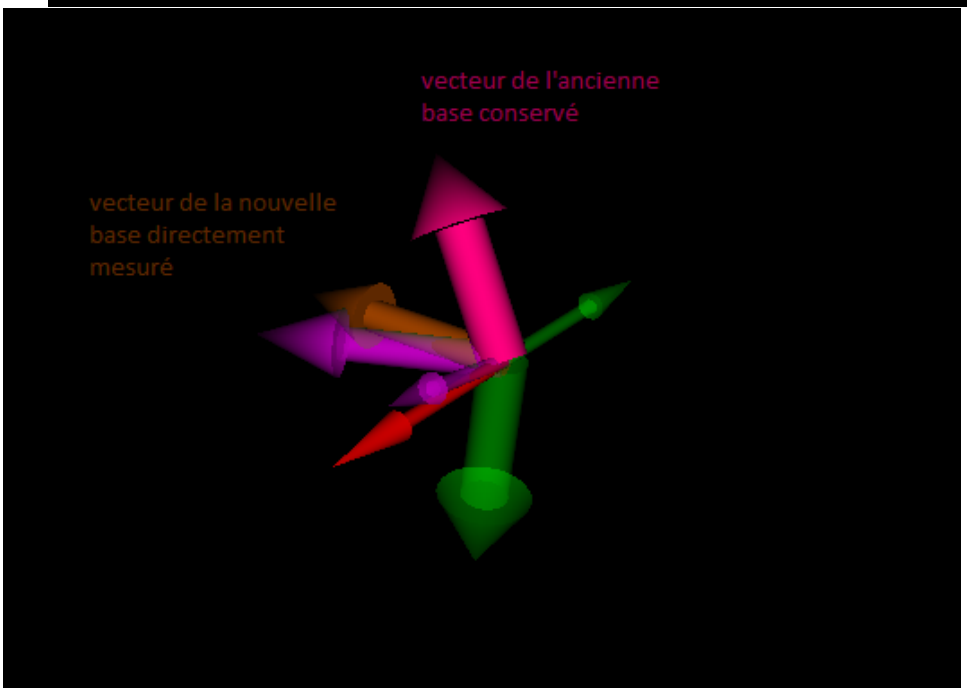


5.2 Continuité et cohérence

Nous déterminons ainsi la base la plus proche possible de la précédente compte tenu des données récoltées, mais cela n'apporte rien quant à la cohérence des données extraites de l'image, qui peut être relativement chaotique (flou..). En particulier, deux vecteurs de l'ancienne base pourraient

être remplacé par le même vecteur relevé dans l'image. Il faut donc s'assurer de la continuité des bases à travers le temps (les différentes images).
Un simple seuillage sur la différence des normes est insuffisant : J'ai décidé de selectionner les vecteurs individuellement : on ne remplace les vecteurs de l'ancienne base que si leur produit scalaire avec leur remplacant (proximité) est suffisament élevé. Ceci fait, si nécessaire, on ne garde que les deux remplaçants les plus cohérents et on recalcule le cube par un nouveau produit vectoriel (pour un maintien de l'orthonormalité). Le résultat est très satisfaisant et garantit un mouvement continu et cohérent.


nouvelle
base
mesures
ancienne base


vecteur de l'ancienne
base conservé
vecteur de la nouvelle
base directement
mesuré

6 Captures d'écran

Le programme permet d'afficher le résultat sous formes d'arêtes ou de cube (modifier la propriété #drawArrows (1 pour afficher les lignes, 2 pour les lignes sans le cube). Il propose aussi d'afficher les points d'intersection et la direction des points de fuites considérés sur l'input (modifier la propriété #verbose entre 0,1 (lignes) et 2 (lignes et points)). Voici des captures d'écran réalisées à partir de diverses vidéos :