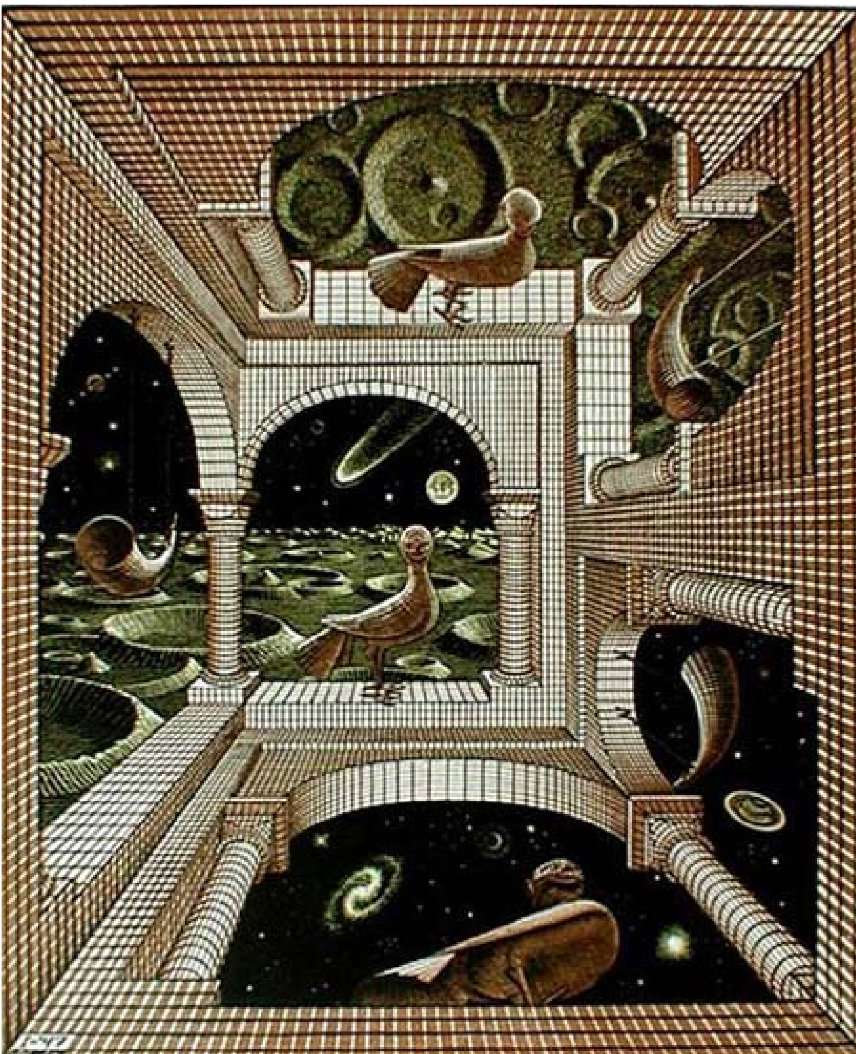
2009

Projet : Lignes de Fuites



MATHEMATIQUES

Adrien Benoist & Julien Issartel

11/12/2009

Sommaire

Introduction 3
I. Calcul de points de fuites : 3
A. Stockage 3
B. Affichage 3
C. SVD 4
II. Conique : 4
A. Construction 4
B. Vérifications 5
III. Cholesky : 5
A. Algorithme 5
B. Vérification 6
C. Améliorations 6
D. Complexité 6
IV. Homographie 7
A. Principe 7
B. Vérifications 7
C. Stockage 7
D. Algorithme 7
V. Dessin de la Teapot en Opengl 7
A. Principe 7
B. Construction 8
C. Implémentation 8
VI. Fonctions SAVE et LOAD 9
VII. Glui 9
VIII. Objectifs 10
IX. Améliorations 10
Conclusion 10

Introduction

Dans le cadre de ce projet nous devions créer une application permettant d'insérer un objet 3D dans une scène 2D, en tenant compte de la perspective. C'est une application concrète des mathématiques dans le monde de l'image. Pour cela il fallait implémenter différents algorithmes, tels que celui de Cholesky ou encore le calcul de l'image de la conique absolue, indispensables en réalité augmentée.

Ce projet, à réaliser par binôme, était composé d'Adrien Benoist et de Julien Issartel.

I. Calcul de points de fuites :

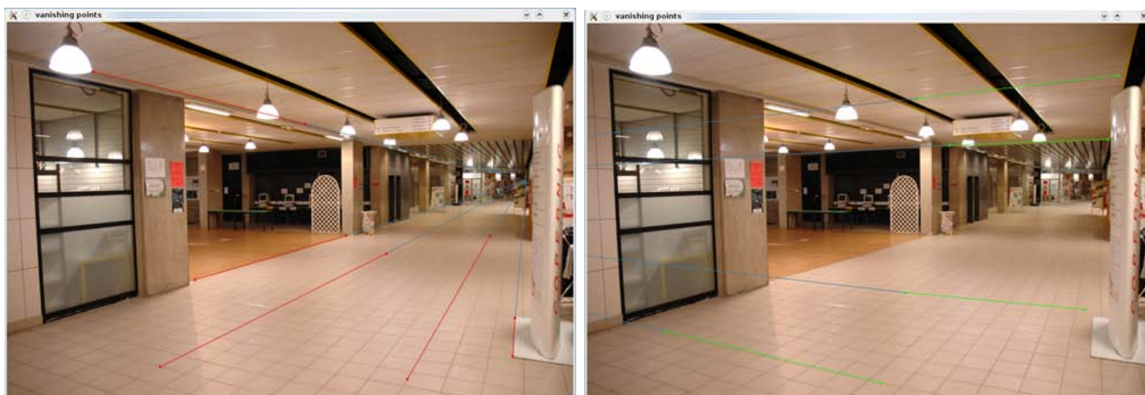
A. Stockage

Pour calculer un point de fuite, nous avons récupéré les coordonnées des points cliqués que nous avons placés respectivement dans la première et la deuxième ligne d'une matrice 3x3. La troisième ligne contient le produit vectoriel des deux lignes précédentes. Nous avons fait cela pour chaque couple de point. Tous les produits vectoriels ainsi crées ont ensuite été placés dans une liste de vecteurs. On crée une matrice de la taille de la liste à laquelle nous appliquons une SVD (fonction fournie dans OpenKraken). On obtient un vecteur qui est notre premier point de fuite. De façon à réaliser un point de fuite qui se met à jour au fur et à mesure que l'on clique, on ajoute dans la liste les nouveaux produits vectoriels issus des couples de points sélectionnés. Ainsi à chaque passe, la matrice nouvellement créée est actualisée (car elle est de la taille de la liste contenant les produits vectoriels) et la SVD qu'on lui applique donne un vecteur contenant les coordonnées du point de fuite, qui est également actualisé.

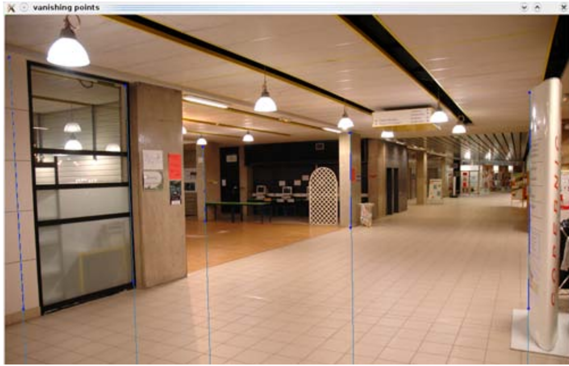
On applique cette démarche pour chaque liste et cela nous donne tout nos points de fuite.

B. Affichage

Une fonction de dessin prenant en paramètre les points de fuites et les segments cliqués trace un segment entre le deuxième point cliqué et le point de fuite, de façon à mieux visualiser où il se trouve (car les points de fuites sont souvent hors écran).



Visualisation des deux premiers points de fuite



Visualisation du troisième point de fuite

C. SVD

Il est important de mentionner l'utilisation de la SVD. Bien que nous n'en ayons pas implémenté l'algorithme, nous nous en sommes servis plusieurs fois.

Théoriquement, les lignes que l'on sélectionne mènent toutes exactement vers le point de fuite, toutes les équations de droites sont donc censées se rejoindre au même point. Sauf qu'avec les erreurs de précision de la caméra et surtout du « cliqueur », nous traçons des droites qui ne se rejoignent pas toutes parfaitement en un même point. La SVD permet d'obtenir, grâce à la méthode des moindres carrés, une solution qui convient au mieux aux équations qui lui sont proposées.

La fonction utilisée décompose la matrice A surdéterminée en produit de matrices et on obtient les valeurs singulières de A sur la diagonale de la matrice « pseudo-diagonale » issue de cette décomposition. Les valeurs des éléments de la diagonale sont ensuite placées dans un vecteur de taille 3. Ce dernier représente les coordonnées du point de fuite.

II. Conique :

A. Construction

Construire la conique ne nécessite pas un réel algorithme. En prenant en paramètre les trois vecteurs de points de fuite, réaliser la conique consiste à réaliser un certain nombre de calculs (donnés dans l'énoncé, voir l'illustration ci-contre) avec leurs coordonnées pour remplir une matrice A de taille 3*4.

$$A = \begin{bmatrix} \vdots & \vdots & \vdots & \vdots \\ x_u x_v + y_u y_v & x_u w_v + w_u x_v & y_u w_v + w_u y_v & w_u w_v \\ \vdots & \vdots & \vdots & \vdots \end{bmatrix}$$

Une fois cette matrice remplie, nous l'avons passée dans la fonction donnée par OpenKraken qui applique une SVD à notre matrice A, pour renvoyer un vecteur solution de taille 4.

Des éléments de ce vecteur, nous avons rempli une matrice W (l'image de la conique absolue), avec l'organisation donnée par le TD (voir l'image ci-contre).

$$\omega = \begin{bmatrix} \omega_{11} & 0 & \omega_{13} \\ 0 & \omega_{11} & \omega_{23} \\ \omega_{13} & \omega_{23} & \omega_{33} \end{bmatrix}$$

B. Vérifications

Pour vérifier que l'on peut utiliser la conique, il faut que les points de fuites soient calculés. Par conséquent, un test simple a été fait avant de lancer la fonction de la conique. Il suffisait en effet de vérifier que chaque liste contenait au moins deux segments (un point de fuite valable étant l'intersection d'au moins deux segments).

Pour vérifier la validité du résultat de la conique, il faut appliquer deux méthodes :

$$\omega = \begin{bmatrix} \omega_{11} & 0 & \omega_{13} \\ 0 & \omega_{11} & \omega_{23} \\ \omega_{13} & \omega_{23} & \omega_{33} \end{bmatrix}$$

- Vérifier que la matrice obtenue est symétrique (ce qui est trivial et peu utile puisque la dernière manipulation pour créer la matrice le réalise, voir l'image ci-contre)
- multiplier un des points de fuite par la matrice résultat puis encore par un point de fuite :

*PointdeFuite*Matrice*Point de fuite*

Si le résultat est très proche de zéro, alors on peut considérer que la matrice obtenue est valide (ce qui est le cas avec notre matrice résultat ci-contre). Nous avons en effet obtenu comme résultat une valeur avoisinant les 10^{-15}.

```
Conical :
1.05974e-06 0 -0.00068748
0 1.05974e-06 -0.000331738
-0.00068748 -0.000331738 1
```

III. Cholesky :

A. Algorithme

On se sert de l'algorithme de Cholesky afin de trouver les paramètres intrinsèques de la caméra.

L'algorithme consiste à construire une matrice K à partir d'une image de la conique absolue W, que l'on décompose. De la définition mathématique donnée dans le sujet (ci-contre), nous en avons tiré ce pseudo-code :

$$L_{ii} = \sqrt{(A_{ii} - \sum_{k=0}^{k<i} L_{ik}^2)} \qquad \text{pour } i = j$$

$$L_{ij} = \frac{1}{L_{jj}} \left(A_{ij} - \sum_{k=0}^{k<j} L_{ik} L_{jk} \right) \qquad \text{pour } i > j$$

Pseudo code :

Pour i de 0 à largeur de K
    Pour j de 0 à hauteur de K
        Si j est égal à i
            K[i][i] = W[i][i]-Somme des élément de k à i de K[i][k]².
        Si i est supérieur à j
            K[i][j] = 1/K[j][j](W[i][j]-Somme des éléments de k à j de K[i][k]*K[j][k])*
Inverser K
Transposer K

Il ne faut pas oublier de renvoyer la transposée inverse de K pour obtenir la bonne matrice, car la décomposition de Cholesky ne donne pas le « bon K » directement.

B. Vérification

Une fois la matrice K obtenue, nous l'avons vérifiée.

Pour vérifier la validité de nos résultats implémentés par Cholesky, nous avons regardé la forme que prenait notre matrice, en l'affichant dans la console.

Les deux éléments de focale (les deux premiers éléments de la diagonale) doivent être égaux. Les deux premières valeurs de la dernière colonne doivent être égales à la moitié de la largeur et de la hauteur de l'image de base. La dernière case doit avoir une valeur approchant à 1. Enfin, les autres valeurs de la matrice doivent approcher de 0. Une fois toutes ces vérifications faites, et en relativisant les résultats à cause du manque de précision, nous pouvons considérer notre résultat de matrice comme valide.

C. Améliorations

Pour améliorer la précision de la matrice obtenue, nous avons divisé tous les éléments de la matrice par le dernier élément de la matrice, afin que ce dernier soit égal à un. De plus, pour ne pas accumuler d'erreurs, avons remplacé les valeurs quasiment nulles par 0) Après toutes ces transformations, la matrice ressemble à celle-ci-contre :

```
Cholesky :
651.761 0 648.722
0 651.761 313.036
0 0 1
```

Lorsque nous lancions deux fois la fonction Cholesky à la suite (par exemple en exécutant la fonction Cholesky puis l'affichage de la teapot qui relance Cholesky au cas où ça ne soit pas déjà fait), nous obtenions des résultats farfelus. Pour ce faire, nous avons réinitialisé la matrice au début de la fonction pour éviter ce genre de résultats.

Pour rendre l'utilisation du logiciel plus simple à l'utilisateur, nous avons lancé la fonction de la conique avant de lancer Cholesky. Ainsi, l'utilisateur peut directement lancer Cholesky sans avoir cliqué sur la conique. (Car la fonction Cholesky nécessite d'avoir la matrice renvoyée par la fonction conique).

D. Complexité

Cholesky implique deux boucles de déplacement dans la matrice, plus une qui se sert de l'élément précédent pour calculer i et j. Cholesky réalise aussi une somme pour chaque case explorée (une pour quand i est égal à j, ou une quand i est supérieur à j).

Ce qui nous mène à une complexité en : 0(n*n*2(n/2)), soit n^3.

IV. Homographie

A. Principe

L'homographie permet de projeter des points d'un plan à un autre (deux plans projectifs), tout en suivant certaines relations de transformations. C'est-à-dire que notre objet 3D (la teapot) appartenant au plan perpendiculaire à notre vision, va être projeté sur le plan défini par le sol de l'image.

B. Vérifications

Pour pouvoir réaliser l'homographie, il faut que Cholesky soit lancé. Ainsi, pour simplifier encore une fois le travail de l'utilisateur, lors du lancement de l'homographie, les fonctions de conique et de Cholesky sont lancées au préalable, avec vérification du point de fuite. Ces fonctions étant dans ce cas-là de moindre coût, cela ne semble pas très grave de les lancer une fois de plus (au cas où l'utilisateur l'ait déjà fait).

C. Stockage

Pour réaliser l'homographie, nous avions besoin de quatre points, que nous avons stockés dans une matrice. Afin de savoir combien de points ont été cliqués (pour arrêter de stocker les points et pour lancer l'homographie), nous avons créé un compteur en variable globale, qui s'incrémente à chaque clic. Ce compteur nous a été utile, puisqu'en plus de déterminer à combien de points cliqués on est, il nous a servi dans la fonction d'affichage des quatre points pour parcourir la matrice des points sélectionnés.

D. Algorithme

Pour réaliser l'homographie, nous avons réalisé une fonction qui créée le vecteur de pairs que l'on va donner comme paramètre à la fonction ComputeHomography() ;

Une fois celle-ci appelée dans la fonction de clics de souris, nous avons paramétré la caméra dans cette même fonction (la matrice de projection P et le modelview opengl).

V. Dessin de la Teapot en Opengl

A. Principe

Pour dessiner un objet 3D qui respecte la perspective, il faut pouvoir caractériser la caméra qui va observer la scène par sa position t (vecteur), son orientation R = [r1 r2 r3](matrice 3*3), et ses paramètres intrinsèques K (matrice déjà calculée avec Cholesky).

B. Construction

Bien qu'avec OpenKraken la construction de P soit déjà implémentée (en lui donnant simplement K), nous avons essayé de construire P, à partir de K et H, en suivant la méthode donnée dans le sujet.

$$\mathbf{x} = \begin{pmatrix} x \\ y \\ w \end{pmatrix} = K \left[\begin{array}{ccc|c} r & r & r & t_x \\ r & r & r & t_y \\ r & r & r & t_z \end{array} \right] \begin{pmatrix} X \\ Y \\ 0 \\ W \end{pmatrix}$$

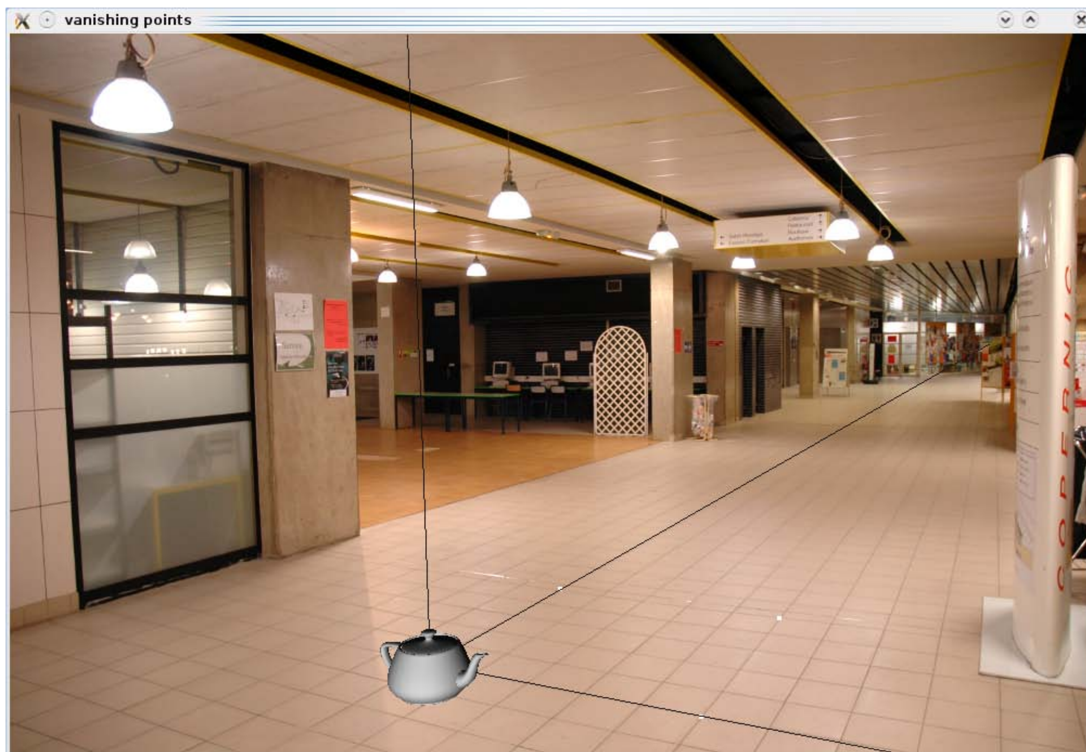
Pseudo code (Avec K la matrice des paramètres intrinsèques de la caméra, H l'homographie des points 3D et 2D de l'image) :

Inverse K
*Valeur temp=K*H*
Indentifier r1 et r2 et t à partir de temp
*r3=r1*r2*
On remplit la matrice [R|t]
P=K[R|t]*

C. Implémentation

Nous avons cependant préféré nous servir de la fonction directement fournie par OpenKraken pour calculer et utiliser les paramètres internes de la caméra. Ensuite, avec deux lignes de codes proposées par OpenGL, nous avons conditionné la matrice de Projection et la matrice Modelview qui seront utilisées lors de l'affichage.

Grâce à la caméra correctement paramétrée, nous avons complété la fonction d'affichage en 3D, en dessinant un repère en 3D et une teapot. A notre grande surprise, notre teapot s'est affichée couchée. Après réflexion, nous avons compris que ce n'était pas nos calculs qui étaient faux, mais que le positionnement des axes sous OpenGl était différent de ceux que nous avions calculé. Nous avons donc corrigé cela en faisant rotater notre teapot de 90 degrés sur l'axe des x. En voici le résultat :



Résultat de l'application

VI. Fonctions SAVE et LOAD

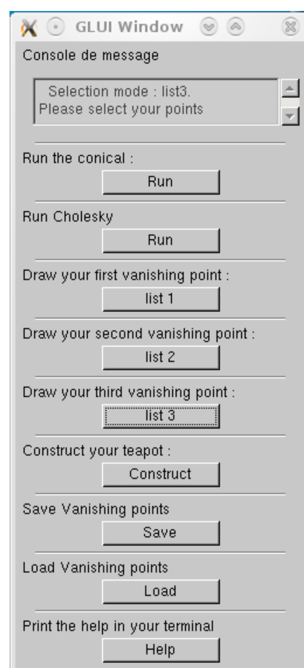
Lors des différentes phases de test, il était très rébarbatif de sélectionner tous nos points pour avoir nos segments. Nous avons donc décidé d'implémenter ces deux fonctions. La première (save) se contente d'enregistrer le numéro de la liste dans laquelle les matrices contenants les coordonnées des points sont stockées, ainsi que leurs coordonnées. Pour des raisons pratiques nous avons stocké les points 2 à 2. Les informations dans le fichier texte se présentaient donc de la manière suivante :

Numéro_liste X1 Y1 X2 Y2

La fonction load quand à elle, récupère les informations dans des vecteurs temporaires qu'on place ensuite dans des matrices pour pouvoir refaire la SVD et ainsi obtenir notre point de fuite. Une autre manière de faire aurait été de stocker également les points de fuite, mais nous nous sommes dit que les recalculer à la volée ne demanderait que très peu de ressource. Bien évidemment avant de charger les points dans les listes, on les vide. De cette manière si l'utilisateur commence à entrer des points puis en charge d'autre depuis un fichier, seuls ceux chargé seront présents.

VII. Glui

Pour faciliter l'expérience utilisateur, nous avons implémenté une petite interface GLUI. Celle-ci se contente juste de reprendre les raccourcis claviers sous forme de boutons de contrôle. Un petit plus est que les messages affichés dans la console sont retranscrits dans GLUI, ce qui évite d'avoir à regarder dans la console pour utiliser le programme. Malheureusement, GLUI étant limité en terme d'affichage (avec messageBox), nous n'avons pas affiché le résultat de matrices calculées (W et K), ni l'aide dans l'interface GLUI.



VIII. Objectifs

Atteint	Ne fonctionne pas	Ajouts
Sélection des lignes de fuite via un programme OpenGL fourni		Interface graphique utilisateur
Extraction des points de fuite		Fonctions de sauvegarde et chargement de points de fuite (fortement conseillé)
Calcul de l'image de la conique absolue W		
Décomposition de Cholesky		
Calcul de K		
Calcul de H via OpenKraken		
Calcul de P via OpenKraken		
Rendu en réalité augmentée		

Nous pensons avoir réalisé tout ce qui a été demandé dans l'énoncé, nous nous sommes même permis d'ajouter une interface GLUI pour faciliter l'expérience utilisateur.

IX. Améliorations

Il est tout à fait possible d'améliorer l'application. Pour ce faire, il faudrait pouvoir proposer une interface utilisateur plus complète et efficace.

De plus, maintenant que nous avons réalisé ceci sur une image fixe, il pourrait être intéressant de le réaliser en 3D, en temps réel. Avec un carré blanc donné comme repère à la caméra, on pourrait dessiner un objet 3D, et tout en bougeant le repère, l'objet suivrait l'orientation du carré. Il existe des librairies, qui comme OpenKraken facilitent un certain nombre de calculs en prenant en chargent des fonctionnalités prédéveloppées (comme ARToolKit).

Conclusion

La difficulté de ce projet ne résidait pas tant dans la programmation que dans les mathématiques. En effet nous sommes passés de mathématiques théoriques à des mathématiques appliquées, (la réalité augmentée est une manière concrète et visuelle d'utiliser les mathématiques). Il a donc fallu se poser les bonnes questions et comprendre les différents mécanismes qui lient les choses entre elles.

Le projet nous a permis de nous familiariser avec le fonctionnement de la réalité augmentée, et avec certains principes qui permettent de la mettre en place. Nous sommes globalement satisfaits d'avoir réussi à mettre en place ce qui nous a été demandé, et d'avoir obtenu un programme qui fonctionne correctement.

La prochaine étape, comme dit plus haut, serait de rendre cette application en temps réel, afin que celle-ci recalcule un repère sur un modèle mobile.