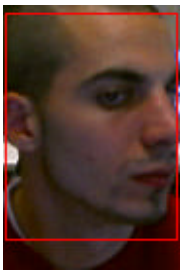
Marty Johann
SSC, 5 ème semestre

Assistant : Lengagne Richard
Hiver 2001-2002



Estimation de la Position d'un Visage dans une Séquence Vidéo et Représentation 3D

Sommaire

1.INTRODUCTION

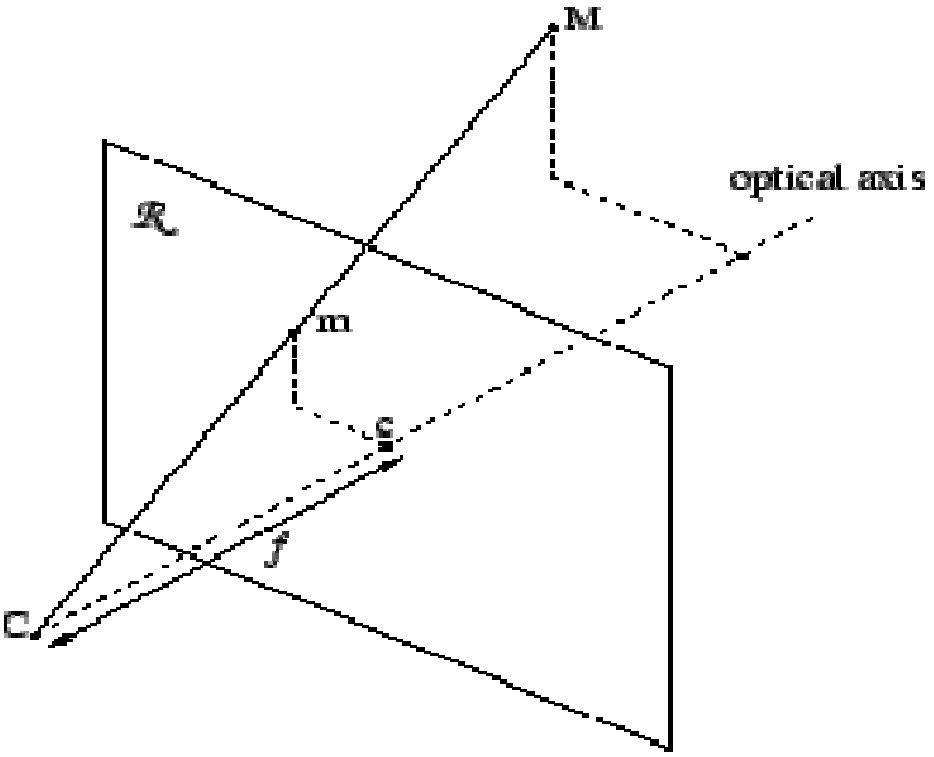
Le but de ce projet est d'estimer la position dans l'espace, l'orientation d'un visage dans une séquence vidéo. Nous pourrons ainsi visualiser le résultat dans une fenêtre openGL à l'aide d'un modèle 3D restituant la même position, oreintation que le visage extrait de la séquence.

Une fois le logiciel ***Borland C++ Builder [12]*** et les librairies adéquates installés (cf chapitres 5 et 6 pour le détail sur les librairies) nous avions besoin de deux outils majeurs à savoir ***openGL [11]*** pour le dessin 3D et l'algorithme ***Posit [16]*** disponible dans ***openCV [17]*** afin de définir la position du visage dans l'espace.

Mais pour pouvoir utiliser correctement l'algorithme ***Posit,*** il fallait calibrer la caméra afin d'obtenir les paramètres internes de celle-ci, indispensables pour cet algorithme.

2. MODELE ET CALIBRATION DE LA CAMERA

2.1) Modèle de la caméra



M : point de l'espace de coordonnées (X, Y, Z)
m : projection du point de l'espace sur le plan R (Retinal plane) de coordonnées (x, y)
c : projection du centre de la caméra sur la plan R.

Le mouvement des points de l'espace affecte les coordonnées comme suit :

$$M' = \begin{bmatrix} R & t \\ 0_3^T & 1 \end{bmatrix} M$$

M ayant pour coordonnées homogènes : (X, Y, Z, 1)
R la matrice de rotation et

$$t = [t_x t_y t_z]^T$$

le vecteur de translation.

2.2) Calibration

Outils nécessaires : Logitech quickcam 3000
logiciel de calibration
damier de calibration

La calibration consiste à présenter le damier de calibration devant la caméra pour que le logiciel puisse analyser un certain nombre de positions du damier et puisse ainsi déterminer les paramètres de la caméra à l'aide de calculs complexes (distance focale, et autres paramètres internes …).

Matrice de la caméra

La matrice est de la forme :

$$M = \begin{vmatrix} fx & 0 & cx \\ 0 & fy & cy \\ 0 & 0 & 0 \end{vmatrix}$$

Fx , fy représentent la distance focale de la caméra qe l'on utilise dans l'algorithme *Posit.*
Cx, cy représentent les coordonnées du centre de la caméra.

Matrice obtenue après la calibration de la caméra (quickcam) :

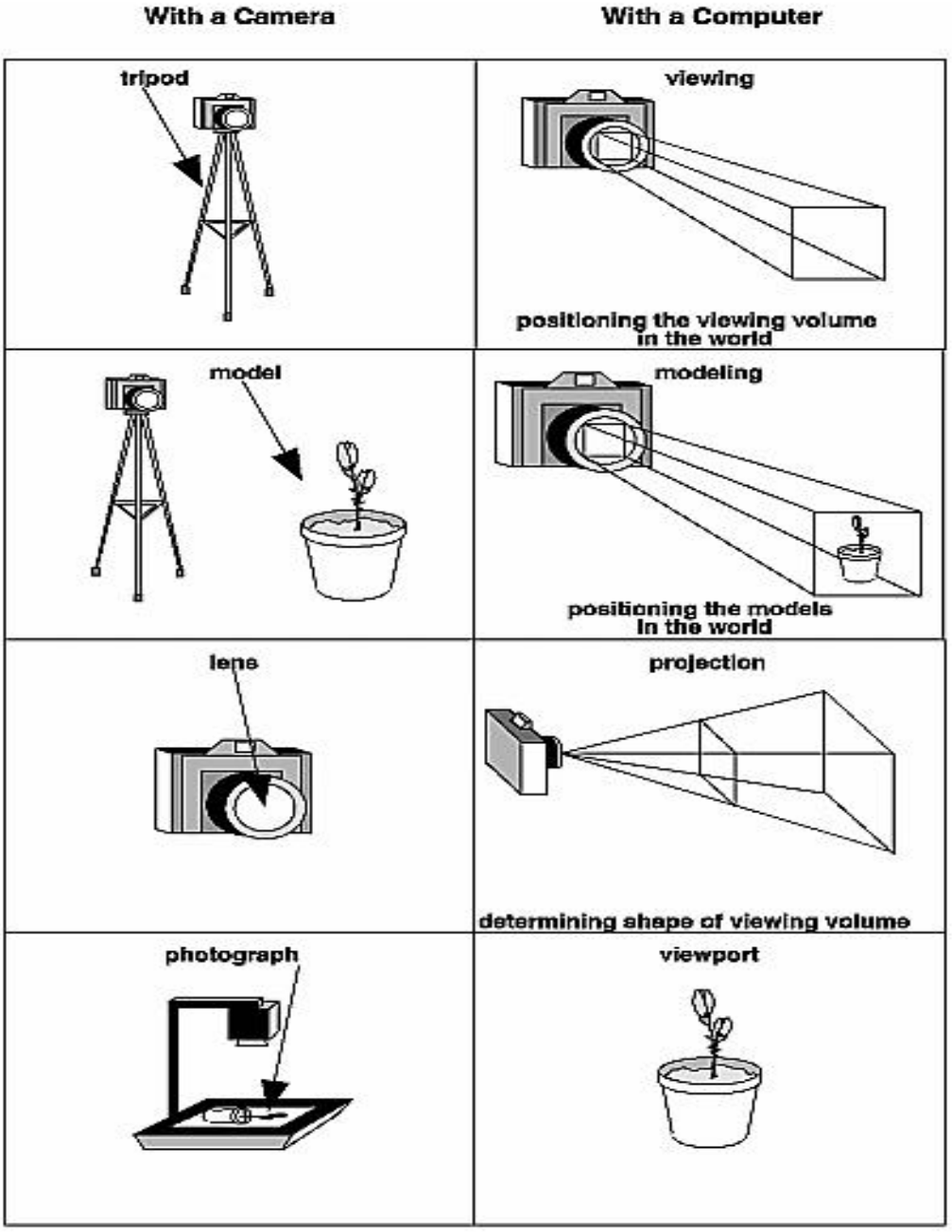
$$M = \begin{vmatrix} 340.790428988 & 0 & 146.358127166 \\ 0 & 341.674680222 & 104.845202422 \\ 0 & 0 & 0 \end{vmatrix}$$

3.OPENGL

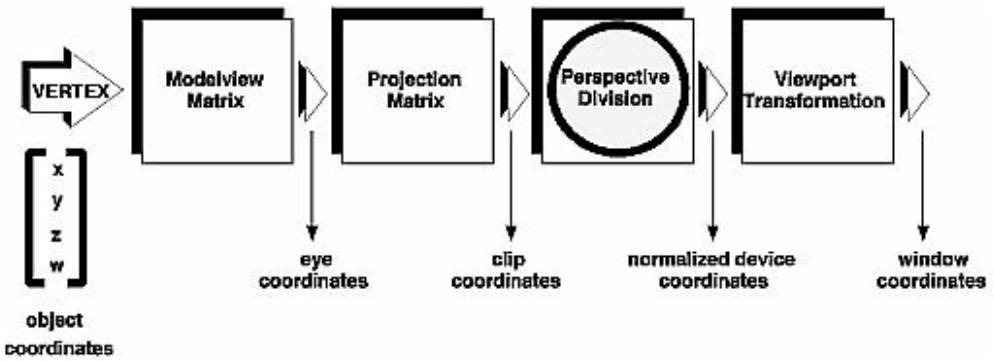
3.1) Introducion

Le premier outil nécessaire au développement est donc ***openGL [20].*** C'est un outil standard pour la visualisation 3D créé par Silicon Graphics.

L'analogie entre une caméra et un ordinateur est schématisée ci-après :



Le fonctionnement d'openGL est représenté ci-après :



3.2) Description de la création du modèle 3D avec openGL

Le modèle 3D est défini par un certain nombre de points (ici 910) et un certain nombre de facettes (ici 1766) . Chaque triplet de points définit un triangle, encore appelé facette. (cf Annexe A)

3.2.1) Méthodes

La méthode ***main()*** d'openGL est la méthode racine à partir de laquelle tous les appels aux autres méthodes se font. C'est aussi la méthode principale dans laquelle on initialise une fenêtre openGL avec un titre, une taille et une position.

La méthode ***init()*** qui sert a initialiser la couleur de fond de la fenêtre, la lumière (une source de lumière ou pas, et si oui laquelle), le modèle (***smooth*** = lissage des facettes, ou ***flat*** = pas de lissage des facettes donc on ne change pas le dessin des triangles) et éventuellement d'autres paramètres. (cf page 8)

La méthode ***display()*** sert quant à elle à afficher le modèle 3D. On peut aussi dans cette méthode changer un certain nombre de paramètres : *la source de lumière, la position de celle-ci, la couleur du modèle, le type de dessin*, GL_TRIANGLE_STRIP = dessine les triangles pleins , GL_LINE_LOOP = dessine seulement les arrêtes et les sommets de chaque triangle du modèle. (cf Annexe A)

La méthode ***reshape()*** permet de redimensionner la fenêtre openGL.

La méthode ***mouse()*** est quant à elle facultative. Elle permet de gérer les éventuels évènements de la souris.

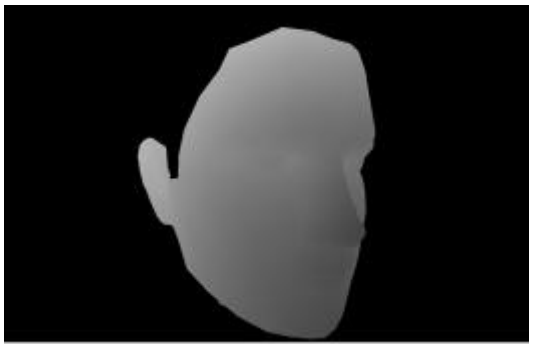
La méthode ***keyboard()*** est aussi facultative. Elle permet de gérer les éventuels évènements du clavier.

Pour chaque option dans openGL, on utilise la commande glEnable (option) pour activer l'option passée en paramètre.

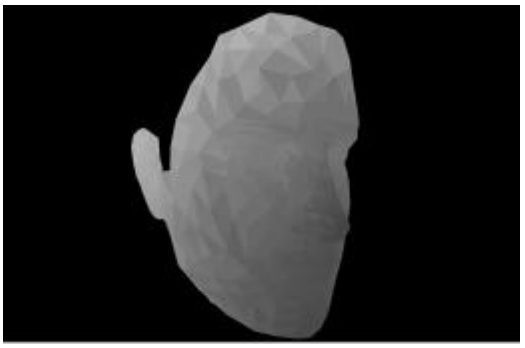
3.2.2) Exemples d'affichage du modèle

Avec l'option : ***GL_TRIANGLE_STRIP***

Ex : smooth(lissé)



Ex : flat(non lissé)



Avec l'option : ***GL_LINE_LOOP***, quelle que soit l'option smooth ou flat.

3.3) Superposition d'une image 2D avec le modèle 3D

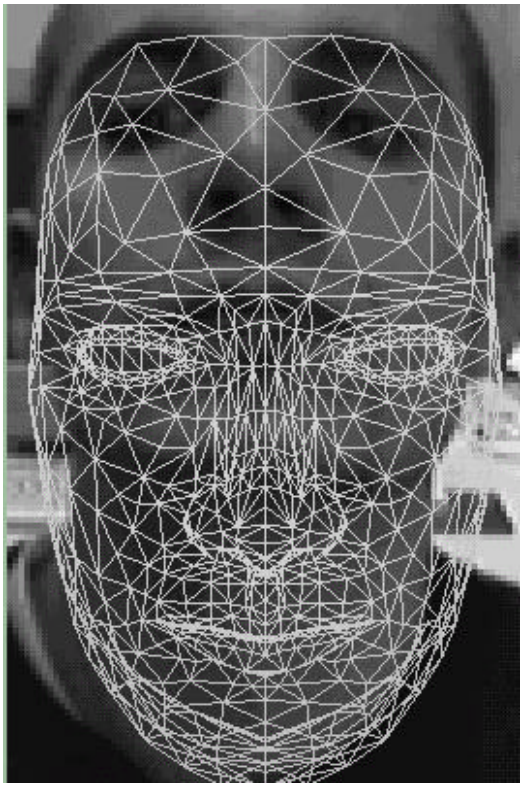
Le but de la superposition est d'obtenir l'image d'un visage et le masque 3D dans la même fenêtre openGL, ce qui permet un test direct de l'algorithme Posit.

Dans un premier temps on teste la superposition sans appliquer l'algorithme Posit.

La première chose à faire est de convertir l'image 2D couleur en image 2D en niveaux de gris (cf Annexe D) afin de pouvoir l'afficher dans une fenêtre openGL.

Une fois cette conversion réalisée, on appelle la méthode **screenDisplay(IplImag *image)** pour afficher l'image passée en paramètre dans la fenêtre openGL**.** Cette méthode permet de régler la taille de l'image par effet de zoom (cf Annexe D).

Ensuite nous devons utiliser la métode **glDisable(GL_DEPTH_TEST)** dans l'initialisation de la fenêtre openGL. Cette méthode permet de désactiver la profondeur de la fenêtre, faute de quoi l'image 2D se mettrait automatiquement devant le masque 3D et on ne verrait alors que l'image 2D.



4.ALGORITHME POSIT (OPENCV)

4.1) Introduction

L'algorithme ***Posit*** décrit dans openCV, a été créé par la société INTEL. Cet algorithme permet de déterminer les 6 degrés de liberté d'un objet 3D rigide. (3 degrés pour la matrice de rotation et 3 degrés pour le vecteur de translation)

POSIT (*pose with iterations*) utilise dans sa boucle d'itérations la position approchée trouvée par l'algorithme *POS* (*pose from orthography and scaling*).

POS calcule la matrice de rotation et le vecteur de translation par la résolution d'un système linéaire, et en approximant la perspective de l'objet par sa projection orthogonale.

POSIT converge vers une position précise en quelques itérations.
L'avantage principal de POSIT par rapport aux autres algorithmes existants est le faible nombre d'opérations et qu'il ne fait aucune inversion de matrices durant le calcul de la position.

4.2)Entrées / sorties de l'algorithme

4.2.1) Entrées

objectPoints N = nombre de points de l'objet 3D dans l'espace avec coordonnées 3D uniques et non coplanaires. (N>3). C'est une matrice N*3.

imagePoints ce sont les cordonnées des points 3D projetés sur le plan. C'est une matice N*2.

focalLength c'est la distane focale de la caméra (définie à l'aide de la calibration).

criteria critère pour le nombre d'itérations de l'algorithme.

4.2.2) Sorties

matrice de rotation 3*3

vecteur de translation 3*1

4.3) Fonctionnement de l'algorithme

Dans un premier temps l'image de l'objet est considérée comme une perspective de l'objet, à partir de laquelle on calcule une approximation en utilisant le model objet pseudoinverse. ("least-squares pose")
A l'aide de cette approximation on projette le modèle objet sur le plan de l'image afin d'obtenir une nouvelle perspective.

Et ainsi de suite jusqu'à convergence de la perspective, dont la position est le résultat de l'algorithme.

4.4)Restrictions

4.4.1) Distance objet-caméra

L'objet doit être suffisamment loin de la caméra afin d'éviter les distorsions. Si celui-ci est à bonne distance, environ une fois sa taille ou plus, alors l'algorithme converge en deux itérations au plus.

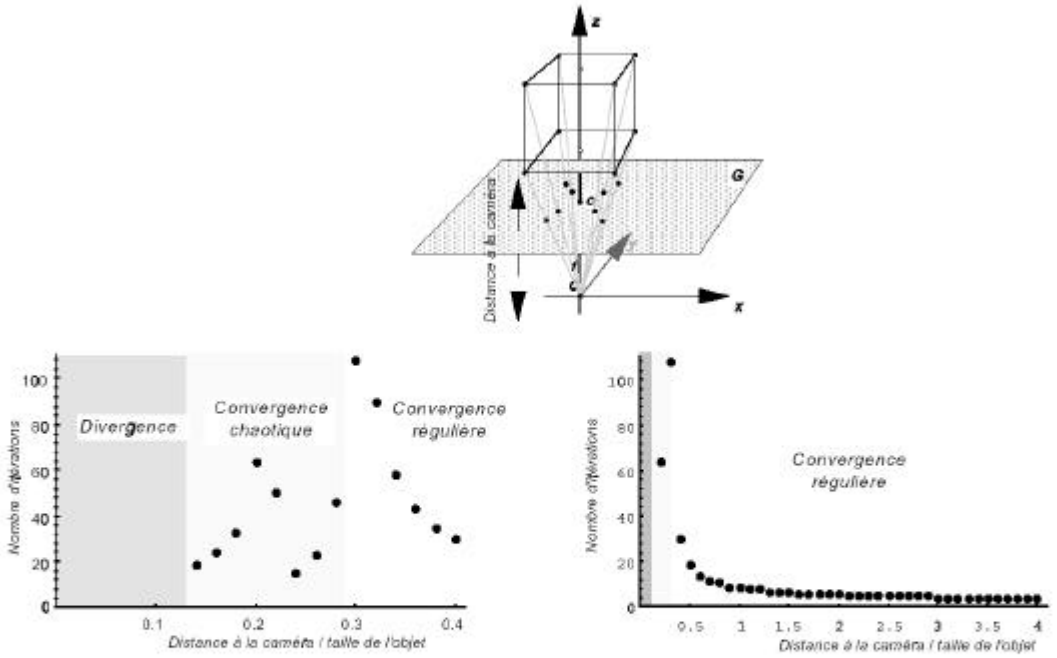


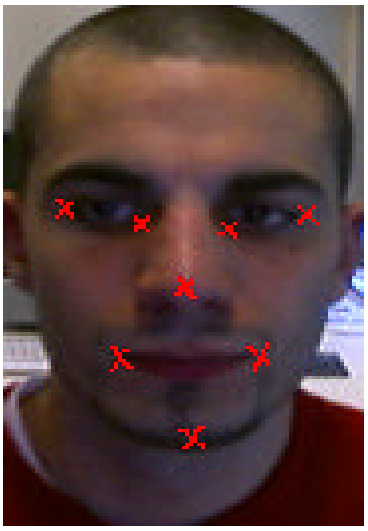
Figure 2.7: Nombre d'itérations pour POSIT en fonction de la distance de l'objet à la caméra. *En haut*: Définition de la distance de l'objet. *A gauche*: Analyse de la convergence aux courtes distances. La convergence apparaît si le cube de 10 cm est à plus de 1,2 cm de la caméra. *A droite*: Nombre d'itérations sur un domaine de distances plus étendu

---

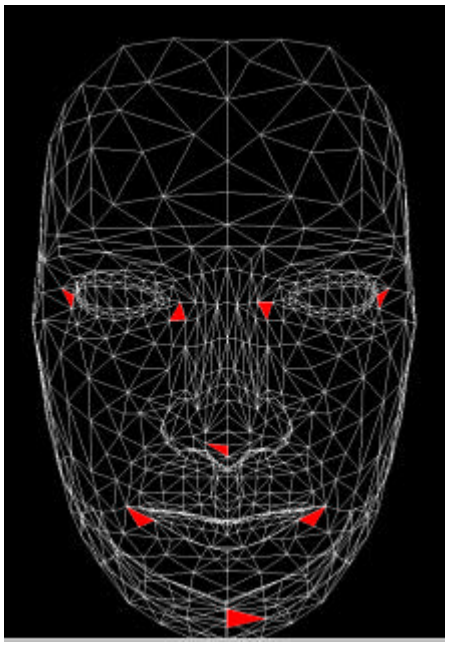
Figure tirée de la thèse de Daniel DEMENTHON autour de l'algorithme Posit qui montre bien (ici pour un cube) que suivant la distance de l'objet par rapport à la caméra, l'algorithme Posit converge plus ou moins bien.

4.4.2) Nombre de points de référence par image

On doit connaître les coordonnées d'au moins quatre points par image. Dans le logiciel, on peut choisir entre 3 et 8 points. (voir chapitre 7)
Voici les 8 points de référence sur le visage

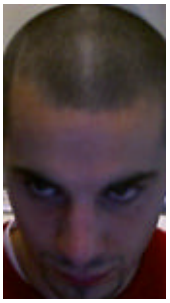


Et voici les 8 points de références sur le modèle 3D

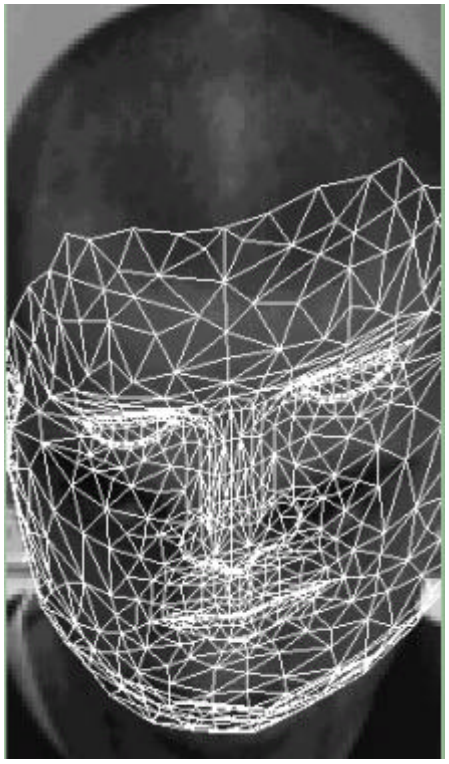


Nous avons choisi un neuvième point de référence identique pour chaque image. Ce point est le centre du repère du modèle 3D, qui est aussi le centre du modèle 3D, et son point de correspondance sur l'image 2D est le centre de cette image. Cette approximation est correcte pour les images que nous avons à traiter car le centre du visage 2D est toujours au centre de l'image.

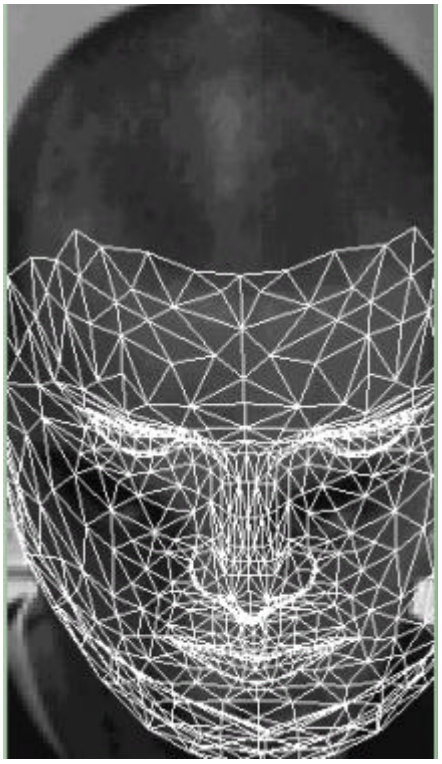
4.5)Résultats de l’algorithme



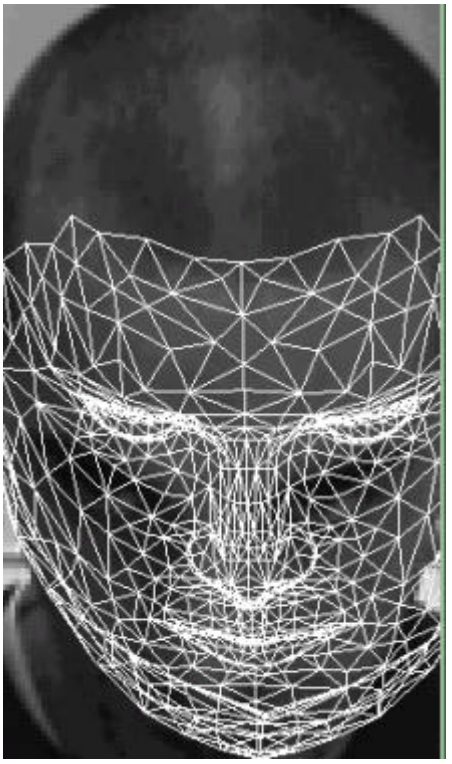
1. Visage de face



Résultat avec **4 points de référence sur l’image.**

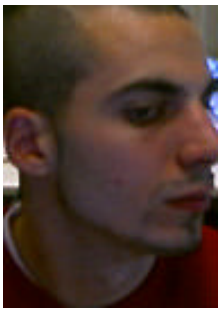


Résultat avec **6 points de référence sur l’image.**

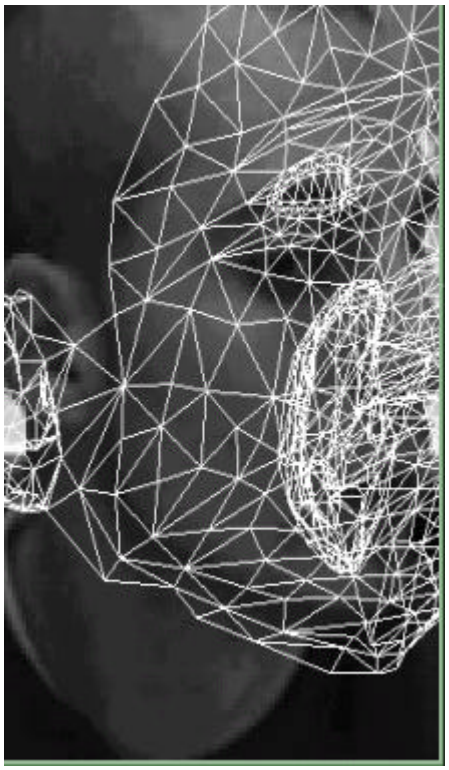


Résultat avec **8 points de référence sur l’image.**

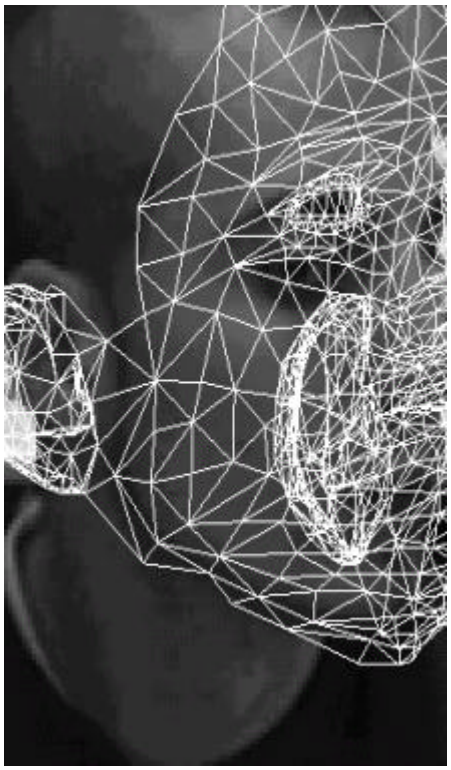
Lorsque l’on choisit seulement 4 points de référence sur l’image, la position n’est pas très précise et le modèle un peu déformé.
Avec un choix de 6 points de référence sur l’image, la position est plus précise, le modèle se recentre par rapport au visage.
Enfin avec 8 points de référence sur l’image, la position est encore plus précise et le modèle parfaitement centré.



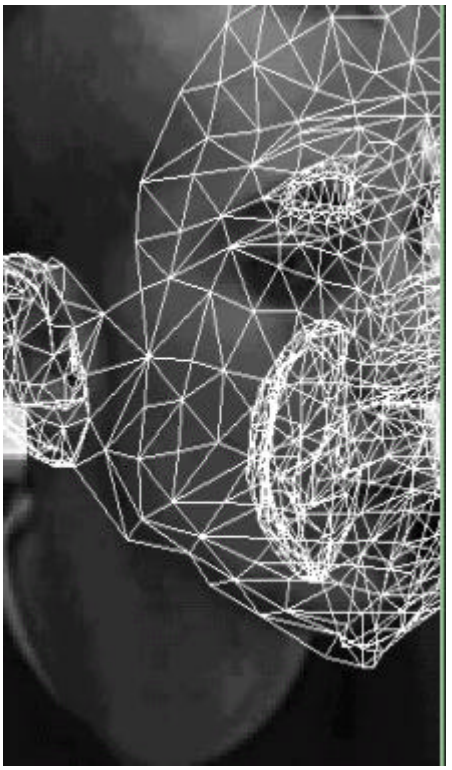
2.Visage de profil



Résultat avec **4 points de référence sur l'image.**



Résultat avec **5 points de référence sur l'image.**

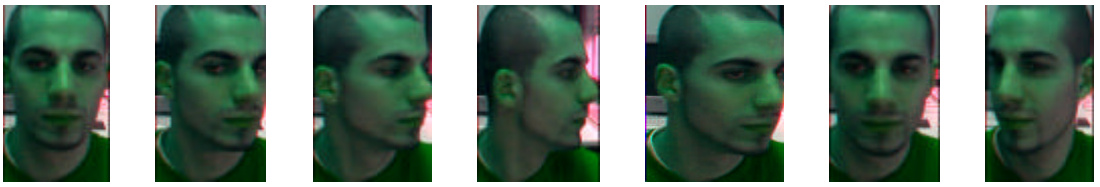


Résultat avec **6 de référence sur l'image.**

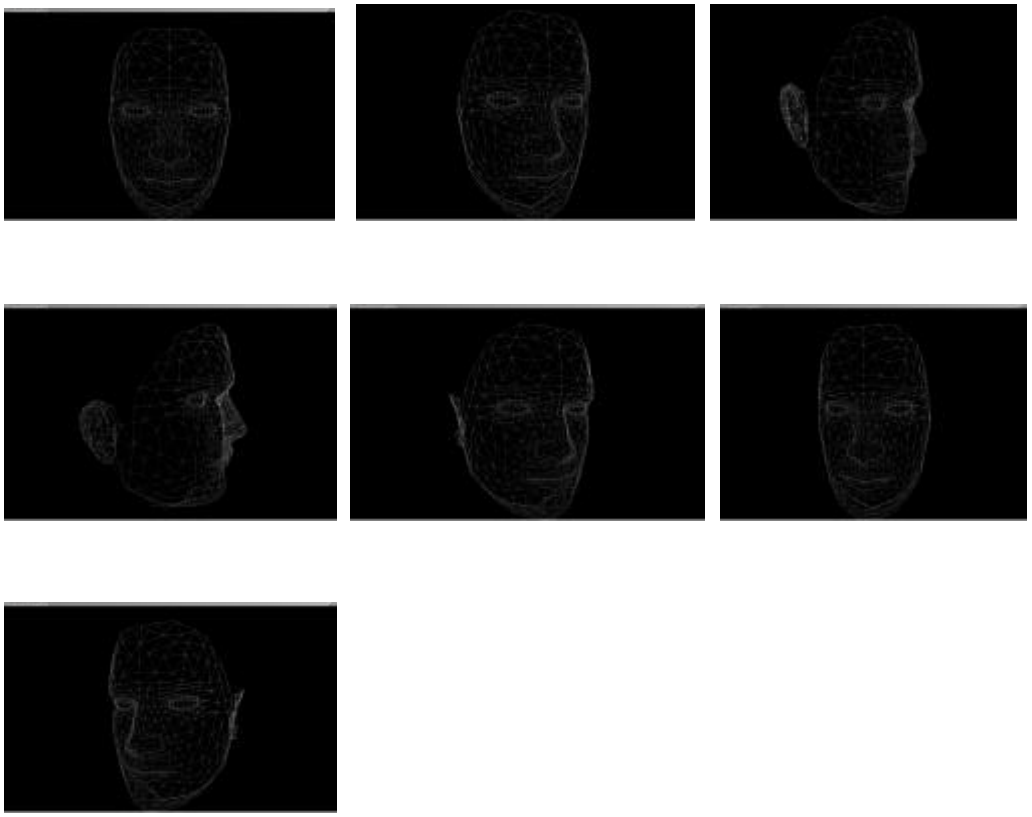
Lorsque l'on choisit seulement 4 points de référence sur l'image, on peut voir un décalage entre le visage et le modèle 3D au niveau de l'oreille droite. Avec 5 points de référence sur l'image, ce décalage a diminué, le modèle a tendance à se recentrer.
Enfin si on choisit 6 points de référence sur l'image, le modèle est exactement dans la même position que le visage.

4.6) Traitement d'une séquence

Une séquence vidéo est une série d'images enregistrées par la caméra.



La séquence openGL consiste ensuite à faire correspondre à chaque image 2D de la séquence vidéo un modèle 3D situé dans l'espace dans la même position que l'image 2D de cette séquence vidéo. Cette correspondance se fait manuellement.



Il resterait à automatiser la correspondance entre les points objets (points du modèle 3D) et les points images (points de l'image 2D) afin de pouvoir traiter sans intervention manuelle une séquence vidéo complète.

5.PROBLEMES RENCONTRES ET RESOLUTION

1 L'incompatibilité entre les librairies ***Microsoft Visual C++*** et celles de ***Borland C++ Builder 4*** nous a posé le premier problème. Pour résoudre celui-ci nous avons recréé les librairies nécessaires à l'aide de la fonction ***implib.exe***. Il a fallu auparavant télécharger les librairies dynamiques ***.dll*** depuis le site ***d'openGL [11]*** et depuis le site de ***openCV [17]***.

2. Le deuxième obstacle consista à adaper la méthodologie d'***openGL*** à ***Borland C++ Builder 4 [12]*** . Nous avons adapté les déclarations de classes et de méthodes.

3. Dans le traitement d'une séquence (cf paragraphe 3.4), le problème majeur consite à trouver les coordonnées des points de référence choisis sur l'image 2D correspondant à celles des points de référence sur le modèle 3D. En effet les coordonnées des points de référence du modèle 3D sont connues à priori mais on ne sait pas faire automatiquement cette correspondance. Nous pouvons choisir de 3 à 8 points de référence parmis les suivants : le ***bout du nez***, le ***coin extérieur de l'œil gauche***, le ***coin extérieur de l'œil droit***, le ***coin intérieur de l'œil gauche***, le ***coin intérieur de l'œil droit***, le ***coin extérieur gauche de la bouche***, le ***coin extérieur droit de la bouche et*** le ***milieu du menton ;*** sans oublier le point de référence commun à chaque image, ce qui représente un nombre de points de référence variant de 4 à 9.

4. La superposition du modèle 3D à l'image 2D se fait sans appliquer l'algorithme du calcul des facettes cachées.

Explication du principe des facettes cachées :

Lorsqu'on dessine le modèle 3D superposé à l'image 2D, on affiche les 1766 facettes.
Si le visage est de face, ceci ne pose pas vraiment de problèmes parce qu'il n'y a pas de superposition.
Mais si le visage est de profil alors on ne veut pas dessiner toutes les facettes, afin de ne pas voir en transparence celles qui ne devraient pas être visibles.

Le calcul est complexe mais voici la démarche :

1. on calcule la **normale à chaque facettes**
2. on calcule la **normale à la fenêtre openGL**
3. on calcule l**'angle entre ces deux normales**.

4. si cet angle est supérieur à 90 ° ou à –90° , alors on n'affiche pas la facette.

1. la normale à une facette est un vecteur orthogonal au plan contenant la facette, dirigé vers l'extérieur de la facette, c'est à dire vers l'extérieur du modèle 3D.

2. la normale à la fenêtre openGL est un aussi un vecteur dirigé vers l'extérieur de la fenêtre, c'est à dire vers nous si on se trouve en face de la fenêtre.

3. c'est un simple calcul d'angle.

4. c'est une vérification, si l'angle calculé est supérieur à 90 ou –90 degrés, on désactive l'affichage de la facette.

6.LIBRAIRIES, DLL, INCLUSIONS ET FICHIERS

1. Librairies:

Les librairies se trouvent dans le répertoire de la racine du disque ***S:\librairies***

Voici la liste des librairies indispensables :

winmm.lib, opengl32.lib, glu32.lib, BORLNDMM.lib, CP3245MT.lib, CppDll.lib, CV.lib, cvlgrfmts.lib, GLU32.lib, glut.lib, glut32.lib, ipl.lib, ipla6.lib, iplm5.lib, iplp6.lib, iplpx.lib, iplw7.lib .

2. DLL :

Les DLL se trouvent dans le répertoire de la racine du disque ***S:\DLL***

DLL utiles :

Winmm.dll, BORLNDMM.dll, glu32.dll, CP3245MT.dll, CppDll.dll, CV.dll, cvlgrfmts.dll, GLU32.DLL, glut.dll, glut32.dll, ipl.dll, ipla6.dll, iplm5.dll, iplp6.dll, iplpx.dll, iplw7.dll, OPENGL32.dll .

3. Inclusions :

Ce répertoire se trouve dans la racine du disque **S:\include**

Fichiers à inclure :

glut.h, process.h, vcl.h, stdlib.h, stdio.h, math.h, cstring.h, io.h, MyIplFunctions.h .

4. Répertoire du progamme :

Ce répertoire se trouve aussi dans la racine du disque **S:\projet**

Dans ce répertoire se trouve tous les fichiers codes sources du projet.

5. Répertoire des fichiers de points et de faces :

Ce répertoire se trouve dans la racine du disque **S:\points_et_faces** et contient les fichiers de coordonnées des points du visage 3D (face.pts) et de numéros des facettes du visage 3D (face.tri) .

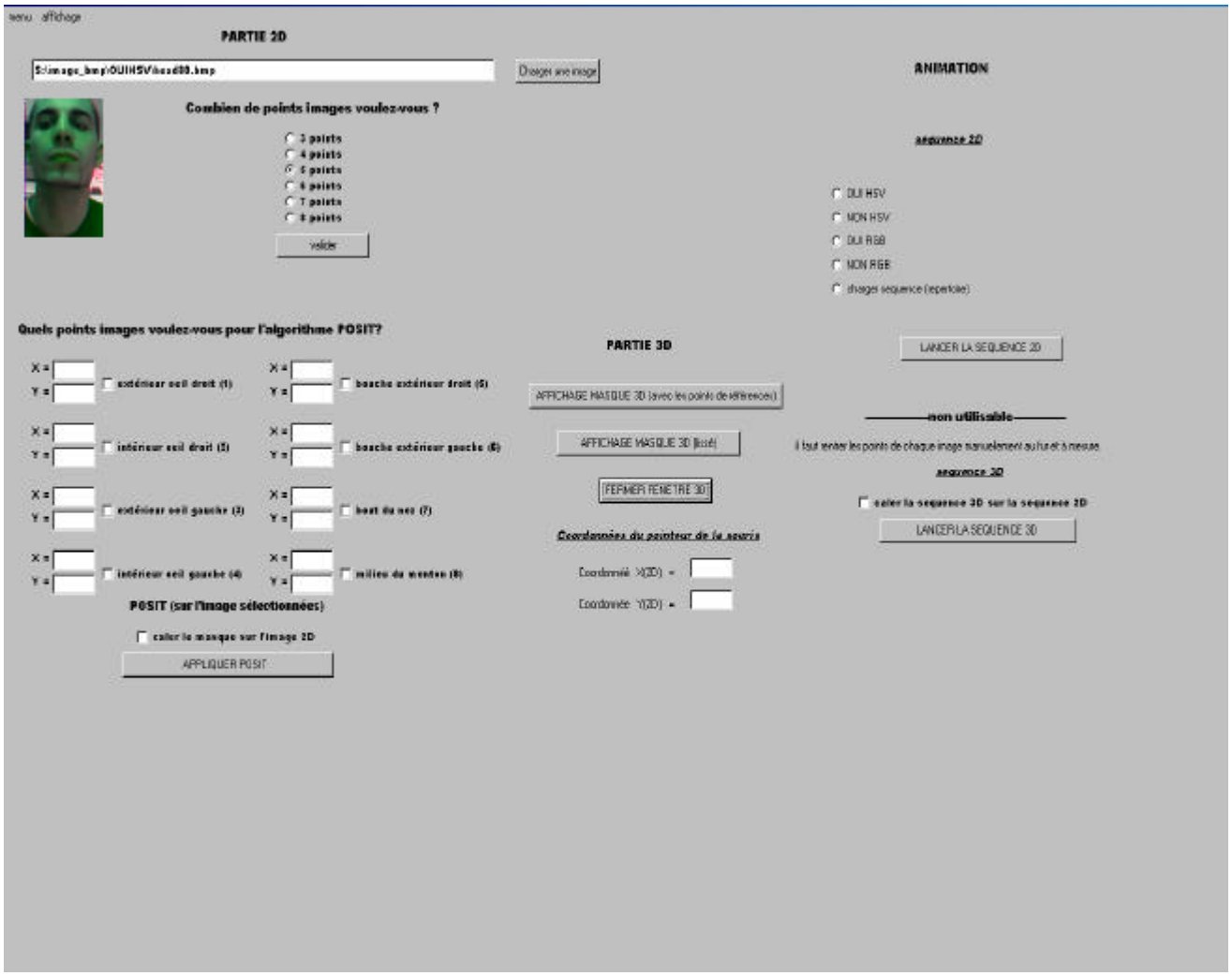
S:
DLL
image_bmp
inlcude
librairies
points_et_faces
projet

Dans la racine, on crée un répertoire ***image_bmp*** dans lequel on sauve les images saisies par la caméra. Par exemple on a déjà quatre sous répertoires : OUIRGB, OUIHSV, NONRGB, NONHSV, qui sont utilisés pour les séquences.

On peut aussi pour faciliter les inlcusions et les utilisations de librairies tout inclure (DLL (*.dll) + librairies (*.lib) + inlcusion (*.h)) dans le même répertoire, par exemple **S:\projet.**

7.APERCU DE L'INTERFACE GRAPHIQUE DE L'APPLICATION

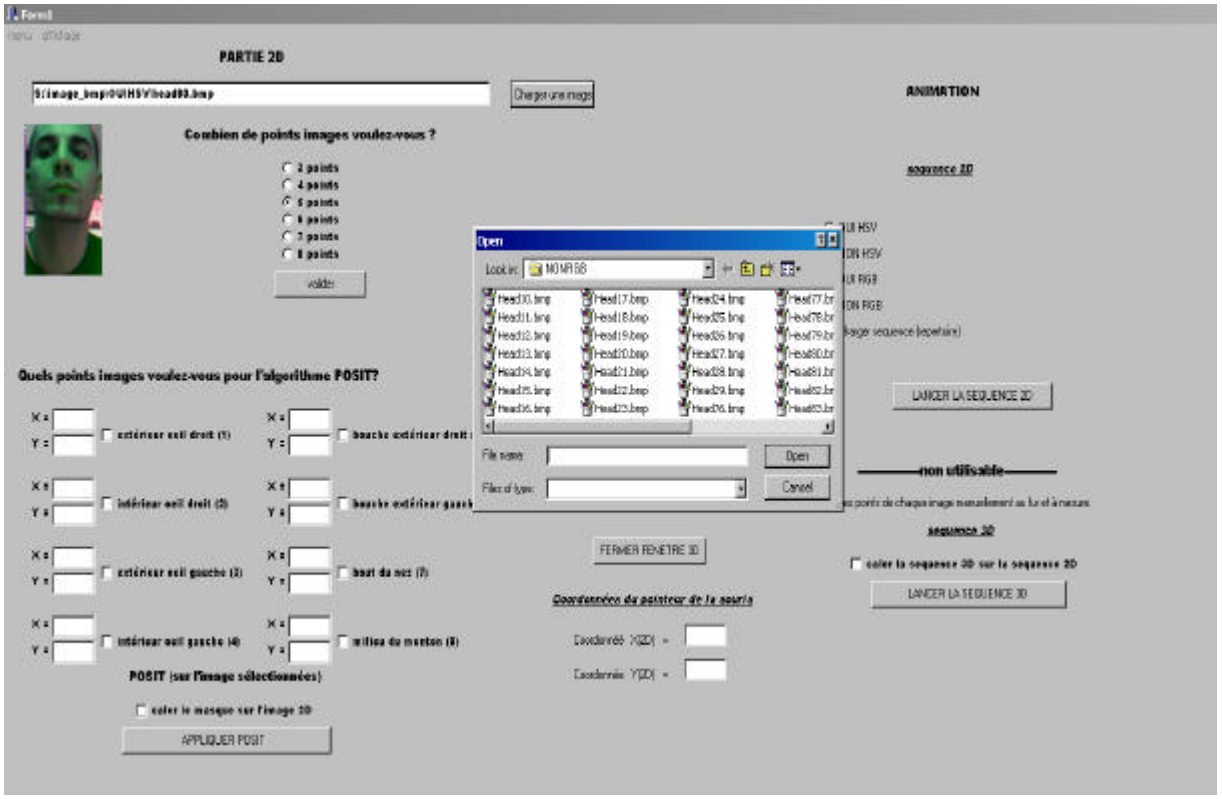
Cette application a été développée sous ***Borland C++ Buider 4.***



L’application est composée de quatre parties

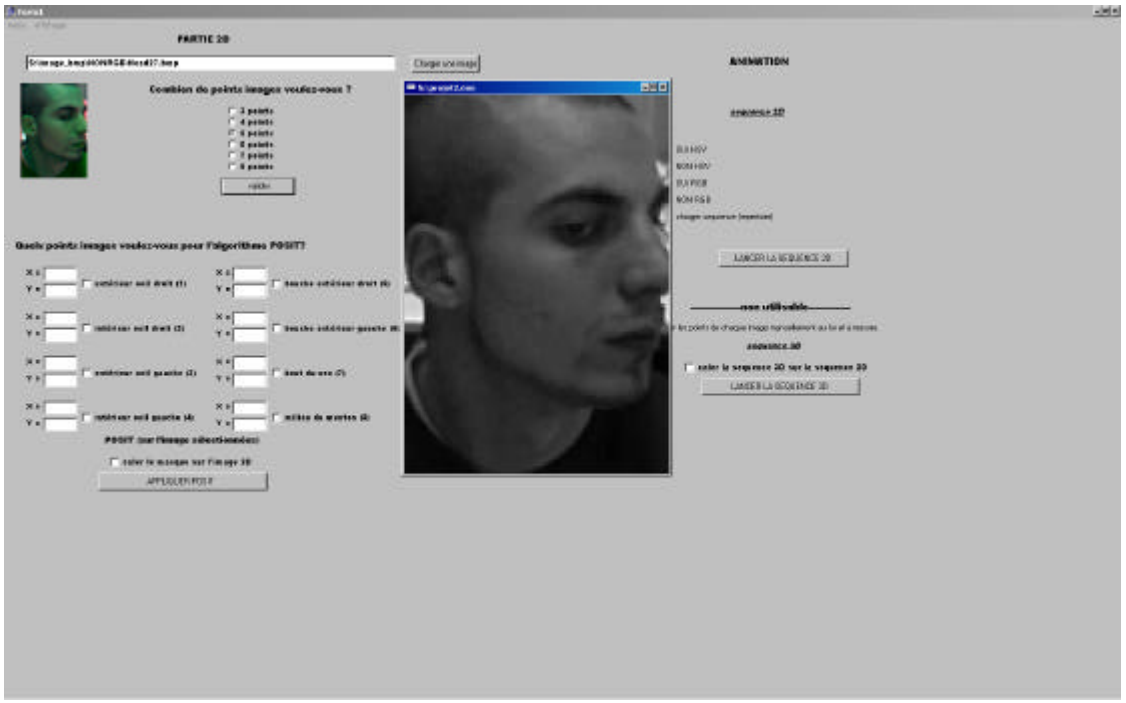
1. la **partie 2D**, définition des points images et de l’image sur laquelle on travaille.
2. la **partie 3D**, à partir de laquelle on visionne un modèle 3D.
3. la partie **POSIT**, à partir de laquelle on lance l’algorithme et on redessine automatiquement le modèle 3D dans la nouvelle position.
4. la **partie animation** fonctionne mais l’animation 3D n’est pas terminée, en effet pour l’instant on peut travailler sur une image puis sur une autre mais pas sur une séquence complète automatiquement.

On a la possibilité de charger une image à l’aide d’une fonction qui parcourt le disque dur.



On a la possibilité de zoomer (dans un rapport de 1 à 6) sur la taille du visage depuis le menu affichage.

Ex : zoom de 4



On a aussi la possiblité de choisir de 3 à 8 points de référence par image. De plus une fonction permet de charger les coordonnées des points sur lesquels on pointe la souris dans la fenêtre où se touve le visage en appuyant sur une des touches 1 à 8 du clavier numérique.

1.l'extérieur de l' œil droit : touche 1 du clavier numérique.
2.l'intérieur de l'œil droit : touvhe 2 du clavier numérique.
3.l'extérieur de l'œil gauche : touche 3 du clavier numérique.
4.l'intérieur de l'œil gauche : touche 4 du clavier numérique.
5.l'extérieur droit de la bouche : touche 5 du clavier numérique.
6.l'extérieur gauche de la bouche : touche 6 du clavier numérique.
7.le bout du nez : touche 7 du clavier numérique.
8.le milieu du menton : touche 8 du clavier numérique.

8.REFERENCES UTILES

1. Bresenham, J. E., Algorithm for Computer Control of a Digital Plotter, IBM Systems Journal 4(1), July 1965, pp 25-30.

2. Chen, S. E. and L. Williams, *"View Interpolation for Image Synthesis"*, SIGGRAPH '93, pp279-288.

3.Chen, S. E., *"Quicktime VR - An Image-Based Approach to Virtual Environment Navigation"*, SIGGRAPH '95, pp29-38.

4. Cohen-Or, D., E. Rich, U. Lerner, V. Shenkar, *"A Real-Time Photo-Realistic Visual Flythrough"*, IEEE Transactions on Visualization and Computer Graphics, Vol. 2, No. 3, September 1996, pages 255-265.

5. Cook, R. L., *"Shade Trees"*, SIGGRAPH '84, pp 223-231.

6. Cook, R. L., L. Carpenter, and E. Catmull, *"The Reyes Image Rendering Architecture"*, SIGGRAPH '87, pp 95-102.

7. Debevec, P. E., C. J. Taylor and J. Malik, *"Modeling and Rendering Architecture from Photographs"*, SIGGRAPH '96, August 1996, pp 11-20.

8. Grossman J.P., andW. J. Dally, *"Point Sample Rendering"*, Proceedings of the 9th Eurographics Workshop on Rendering, June 29 - July 1, 1998, Vienna, Austria, pp 181-192.

9. Logie J. R., and J.W. Patterson, *"Inverse Displacement Mapping in the General Case"*, Computer Graphics Forum, 14 5, December 1995, pp 261-273.

10. Marcato, R.W., *"Optimizing an Inverse Warper"*, Master's Thesis, Massachusetts Institute of Technology, May 1998, pp 35-37.

11. McMillan, L. and G. Bishop, *"Plenoptic Modeling: An Image-Based Rendering System"*, SIGGRAPH '95, pp 39-46.

12. McMillan, L., *"An Image-Based Approach to Three-Dimensional Computer Graphics"*, Ph.D. Dissertation, University of North Carolina, April 1997.

13. Miller, R. and Miller, R., *"The Making of Myst"* , Video accompanying the adventure game *"Myst"* by Cyan, Inc.

14. Patterson J. W., S. G. Hoggar and J. R. Logie, *"Inverse Displacement Mapping"*, Computer Graphics Forum, 10 2, June 1991, pp 129-139.

9.REMERCIEMENTS

Je tiens à remercier très sincèrement les personnes qui m'ont aidé dans la réalisation de ce projet :

- Dr. Richard Lengagne, Senior researcher et mon assitant pour ce projet. (LIG)
- Dr.Vincent Lepetit, Senior researcher. (LIG)
- Ali Shahrokni, Researcher. (LIG)
- Luca Vacchetti, Researcher. (LIG)
- Slobodan Ilic, Researcher. (LIG)
- Gilles Froidevaux, Assistant. (DMT/ISR)

Leurs conseils judicieux et leur soutien m'ont été d'une aide précieuse. Ce projet a été très enrichissant à tous les points de vue et m'a donné beaucoup de plaisir durant sa réalisation.

10.ANNEXES

ANNEXE A

DESCRIPTION DE LA CREATION DU MODELE 3D

```
//    déclaration    de    la    matrice    des    faces    (    des
//     entiers )
GLint face[1767][3];


// déclaration de la matrice des points (des doubles )
GLdouble point[910][3];
```

On rempli ces matrices avec les valeurs contenues dans les fichiers face.pts et face.tri.

```
//-----------------------------------------------------------------
//                            METHODE MAIN
//-----------------------------------------------------------------

void TVisage::Main()   {   //TVisage est le nom de la classe

// création de l'objet visage, c'est à dire remplissage des
//matrices de points et de faces
Visage=new TVisage();

// initialisation de glut
glutInit(numéro de la fenêtre, nom de la fenêtre ); //nom = argv

glutInitDisplayMode (GLUT_DOUBLE | GLUT_RGBA | GLUT_DEPTH);
glutInitWindowSize(taille selon x, taille selon y);


glutInitWindowPosition (position selon x, position selon y);

//création de la fenêtre avec les dimensions précédemment
//définies
glutCreateWindow (&argv[0]);

// exécution de la méthode init()
init ();

// exécution de la méthode display (affichage)
glutDisplayFunc(display);

// exécution de la méthode reshape  (redimensionnement)
glutReshapeFunc(reshape);

// exécution de la méthode mouse (gère les clicks sur les boutons
de la souris)
```

```
glutMouseFunc(mouse);
// exécution de la méthode keyboard (gère les évènements du clavier)
glutKeyboardFunc(keyboard);

// exécution de la boucle principal de glut
glutMainLoop();

return;
}

//-----------------------------------------------------------------
//      METHODE D'INITIALISATION, COULEURS, FENETRE, CAMERA ...
//-----------------------------------------------------------------

void TVisage::init(void)
{
// initialisation de la couleur
glClearColor (0.0, 0.0, 0.0, 0.0);

glShadeModel (GL_SMOOTH);  // ou FLAT

// on valide la lumière
glEnable(GL_LIGHTING);       //par défault glDisable(GL_LIGHTING)
                            //qu'on peut toujours appeler

//utilisation de la première source de lumière
glEnable(GL_LIGHT0);

glDisable(GL_DEPTH_TEST);

}

//-----------------------------------------------------------------
//               METHODE D'AFFICHAGE DU VISAGE ...
//-----------------------------------------------------------------

void display(void) {

//efface l'écran et les profondeurs
glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT);

glPushMatrix ();

glPushMatrix ();

glPopMatrix ();
if(Form1->CheckBox9->Checked){         // appel pour superposer
Visage->ScreenDisplay(Form1->img);     //l'image au modèle 3D
}                                     // si la case est cochée

// boucle pour tracer le visage
```

```
for (int i=0;i<1766;i++)
{
glBegin(GL_LINE_LOOP);
glVertex3d(point[face[i][0]][0],point[face[i][0]][1],point[face[i][0]][2]);
glVertex3d(point[face[i][1]][0],point[face[i][1]][1],point[face[i][1]][2]);
glVertex3d(point[face[i][2]][0],point[face[i][2]][1],point[face[i][2]][2]);
glEnd();
}

glRotated((GLdouble) -spinView, 0.0, 1.0, 0.0);

glPopMatrix ();

glFlush ();

//échange des deux buffers
glutSwapBuffers();

}
//-----------------------------------------------------------------
//              METHODE D'AFFICHAGE DU VISAGE APRES POSIT...
//-----------------------------------------------------------------

void reDisplay(void) {

//efface l'écran et les profondeurs
glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT);

glPushMatrix ();

glRotated((GLdouble) spin, 0.0, 1.0, 0.0);

glPushMatrix ();

glRotated((GLdouble) spinView, 0.0, 1.0, 0.0);


if(Form1->CheckBox9->Checked){
Visage->ScreenDisplay(Form1->img);      //appel pour superposer
     }                                  //l'image au modèle 3D si
                                        //la case checkbox9 est
                                        //cochée

glPopMatrix ();




//affichage du modèle avec calcul des coordonnées après Posit
for (int i=0;i<1766;i++)
{
glBegin(GL_LINE_LOOP);
```

```
glVertex3d((point[face[i][0]][0]*rotative[0]+point[face[i][0]][1]*rotative[1]+point[face[
i][0]][2]*rotative[2]
+translative[0]),(point[face[i][0]][0]*rotative[3]+point[face[i][0]][1]*rotative[4]+point
[face[i][0]][2]*rotative[5]+translative[1]),(point[face[i][0]][0]*rotative[6]+point[face[
i][0]][1]*rotative[7]+point[face[i][0]][2]*rotative[8]+translative[2]));

glVertex3d((point[face[i][1]][0]*rotative[0]+point[face[i][1]][1]*rotative[1]+point[face[
i][1]][2]*rotative[2]+translative[0]),(point[face[i][1]][0]*rotative[3]+point[face[i][1]]
[1]*rotative[4]+point[face[i][1]][2]*rotative[5]+translative[1]),(point[face[i][1]][0]*ro
tative[6]+point[face[i][1]][1]*rotative[7]+point[face[i][1]][2]*rotative[8]+translative[2
]));

glVertex3d((point[face[i][2]][0]*rotative[0]+point[face[i][2]][1]*rotative[1]+point[face[
i][2]][2]*rotative[2]+translative[0]),(point[face[i][2]][0]*rotative[3]+point[face[i][2]]
[1]*rotative[4]+point[face[i][2]][2]*rotative[5]+translative[1]),(point[face[i][2]][0]*ro
tative[6]+point[face[i][2]][1]*rotative[7]+point[face[i][2]][2]*rotative[8]+translative[2
]));

glEnd();
}

glRotated((GLdouble) -spinView, 0.0, 1.0, 0.0);

glPopMatrix ();

glFlush ();

//échange des deux buffers
glutSwapBuffers();

}
//-------------------------------------------------------------------
//                METHODE DE REDIMENSIONNEMENT DE LA FENETRE
//-------------------------------------------------------------------

void reshape (int w, int h){ //avec mouvement de caméra

//définit l'origine et la taille
glViewport (0, 0, (GLsizei) w, (GLsizei) h);


//sélectionne le mode qui va être modifié - la projection
glMatrixMode (GL_PROJECTION);
glLoadIdentity();
float * temp1=Visage->glMat2glVect();
glLoadMatrixf(temp1);  // charge la matrice de déplacement de la
//caméra, qui va suivre le modèle
//sélectionne le mode qui va être modifié - le modèle
glMatrixMode(GL_MODELVIEW);
glLoadIdentity();
}
//-------------------------------------------------------------------
//                METHODE DE REDIMENSIONNEMENT DE LA FENETRE
//-------------------------------------------------------------------
void reshape_1 (int w, int h){ //sans le mouvement de caméra
//définit l'origine et la taille
```

```
glViewport (0, 0, (GLsizei) w, (GLsizei) h);
//sélectionne le mode qui va être modifié - la projection
glMatrixMode (GL_PROJECTION);
glLoadIdentity();
gluPerspective(60.0, (GLfloat) w/(GLfloat) h, 1.0, 50.0);
//sélectionne le mode qui va être modifié - le modèle
glMatrixMode(GL_MODELVIEW);
glLoadIdentity();
}
//-------------------------------------------------------------------
//                   METHODE POUR GERER LA ROTATION DU VISAGE
//-------------------------------------------------------------------
void spinDisplay(){
//modifie le paramètre désiré
spinView = spinView + 0.6;
if(spinView > 360)
spinView = spinView-360;
spin = spinView;
//rappelle display
glutPostRedisplay();
}
//-------------------------------------------------------------------
//            METHODE POUR GERER LES EVENEMENTS DE LA SOURIS
//                  ici, pour la rotation du visage
//-------------------------------------------------------------------
void mouse(int button, int state, int x, int y){
// C'est la méthode qui gère la souris. Bouton de gauche pour
//lancer la rotation
// et bouton de droite pour arrêter la rotation
switch (button) {
case GLUT_LEFT_BUTTON:
if (state == GLUT_DOWN)
glutIdleFunc(spinDisplay);
break;
case GLUT_RIGHT_BUTTON:
if (state == GLUT_DOWN)
glutIdleFunc(NULL);
break;
default:
break;
}
}
//-------------------------------------------------------------------
//            METHODE POUR GERER LES EVENEMENTS DU CLAVIER
//-------------------------------------------------------------------
void keyboard(unsigned char key, int x, int y){
}    //remplir selon l'utilisation en vue.
```

ANNEXE B

DESCRIPTION DE L'UTILISATION DE POSIT

- Déclaration des matrices des points images et objets :

```
CvPoint2D32f imagepoints[nombre de points images];
CvPoint3D32f objectpoints1[nombre de points objets];
(nombre de points images = nombre de points objets)
```

- Déclaration des constantes de l'algorithme Posit résultant de la calibration (distance focale ...) :

```
float cx=146.358127166 , cy=104.845202422;
float fx=340.790428988 , fy=341.674680222;
```

- Déclaration des matrices de translation et de rotation :

```
CvPoint3D32f translation[3];
CvMatrix3 rotation[9];
```

- Déclaration des critères pour la durée de l'algorithme, le nombre d'itérations :

```
CvTermCriteria criteria;
 criteria.type = CV_TERMCRIT_EPS|CV_TERMCRIT_ITER;
criteria.epsilon = 0.000010;
criteria.maxIter = 10000;
```

Déclaration des points objets et images :

Le premier point objet doit toujours être le point (0,0,0)

```
objectpoints1[0].x = 0;
objectpoints1[0].y = 0;
objectpoints1[0].z = 0;

objectpoints1[1].x =…;
objectpoints1[1].y =…;
objectpoints1[1].z =…;

      …..
```

Le premier point image n'est pas forcément (0,0), il peut être n'importe quel point mais ce sera de toute facon la projection du premier point objet. Donc le premier point image définit la position du repère que l'on va utiliser. Dans notre cas nous utilisons le centre de l'image 2D comme projection du point (0,0,0) car le format des images le permet.

```
imagepoints[0].x=(image->width)/2;
imagepoints[0].y=(image->height)/2;

imagepoints[1].x=…;
imagepoints[1].y=…;     point correspondant au point objet 1

      …..
```

- Création de l'objet Posit et exécution de l'algorithme pour cet objet :

```
CvPOSITObject *posit =  cvCreatePOSITObject(objectpoints1, nombre_de_points_objet);

cvPOSIT(imagepoints,posit, fx,criteria,rotation,translation);
```

rotation et translation sont la matrice de rotation et le vecteur de translation en retour de l'algorithme.

Utilisation de l'algorithme pour obtenir les nouvelles coordonnées

En retour de l'algorithme, on a donc une matrice de rotation et un vecteur de translation.

J'enregistre les valeurs de la matrice de rotation dans la matrice rotation[9] et celles du vecteur de translation dans la matrice translation[3].

Ensuite je calcule les coordonnées comme suit :

X' = Rotation[0]*X + Rotation[1]*Y + Rotation[2]*Z + Translation[0];

Y' = Rotation[3]*X + Rotation[4]*Y + Rotation[5]*Z + Translation[1];

Z' = Rotation[6]*X + Rotation[7]*Y + Rotation[8]*Z + Translation[2];

(cf lien vers la thèse de DEMENTHON pour plus de détails sur les calculs géométriques …)

Pour les points images, une chose très importante est de bien choisir le repère sur l'image.
Il faut que l'origine (0,0) soit en bas à gauche de l'image 2D!!
Il est donc évident que dans le code avant d'utiliser Posit je change la coordonnée y comme suit :

Imagepoints[i].y= (Image-->height) – imagepoints[i].y;

ANNEXE C

ALTERNATIVE A POSIT : *IMAGE WARPING*

Définition de la technique

C'est une technique utilisée pour la reconnaissance d'expressions faciales. Mais si on enregistre chaque type d'expression faciale d'une personne comme un ensemble d'images modèles, alors chaque ensemble d'images peut être déformé en une nouvelle image (*view synthesis*) qui a le même point de vue (angle) que l'image en entrée. (voir schéma ci-dessous)

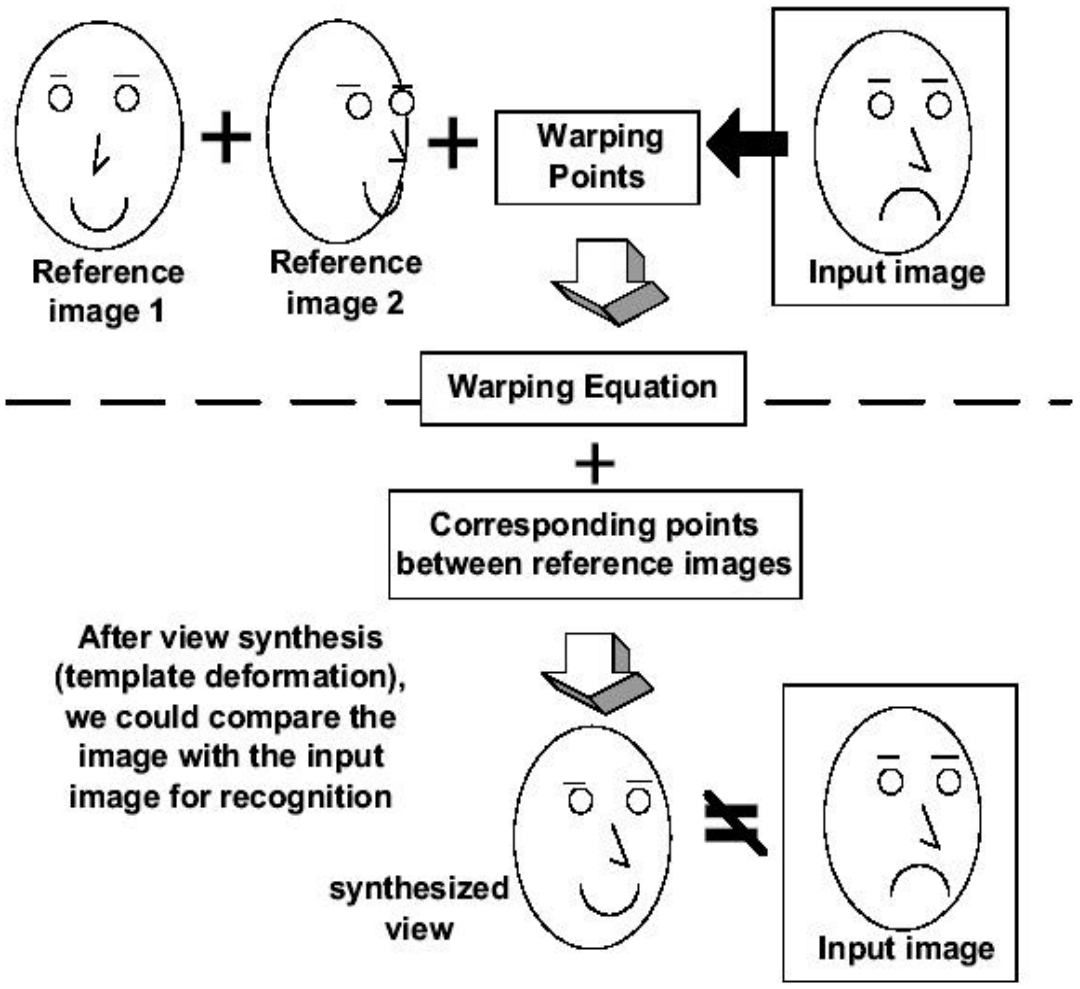


Figure 1 : (View synthesis) Déformation du modèle par synthèse de la vue.

Pour pouvoir appliquer cette technique il faut un certain nombre de points (**corresponding points**), au moins quatre points identiques sur trois images différentes (dans notre cas ce sont les points d'un visage (*figure2*) mais dans l'exemple suivant ces points sont les sommets d'un cube) afin de résoudre les **"warping" equations**. Si on a plus que quatre points, la résolution des équations sera plus rapide.

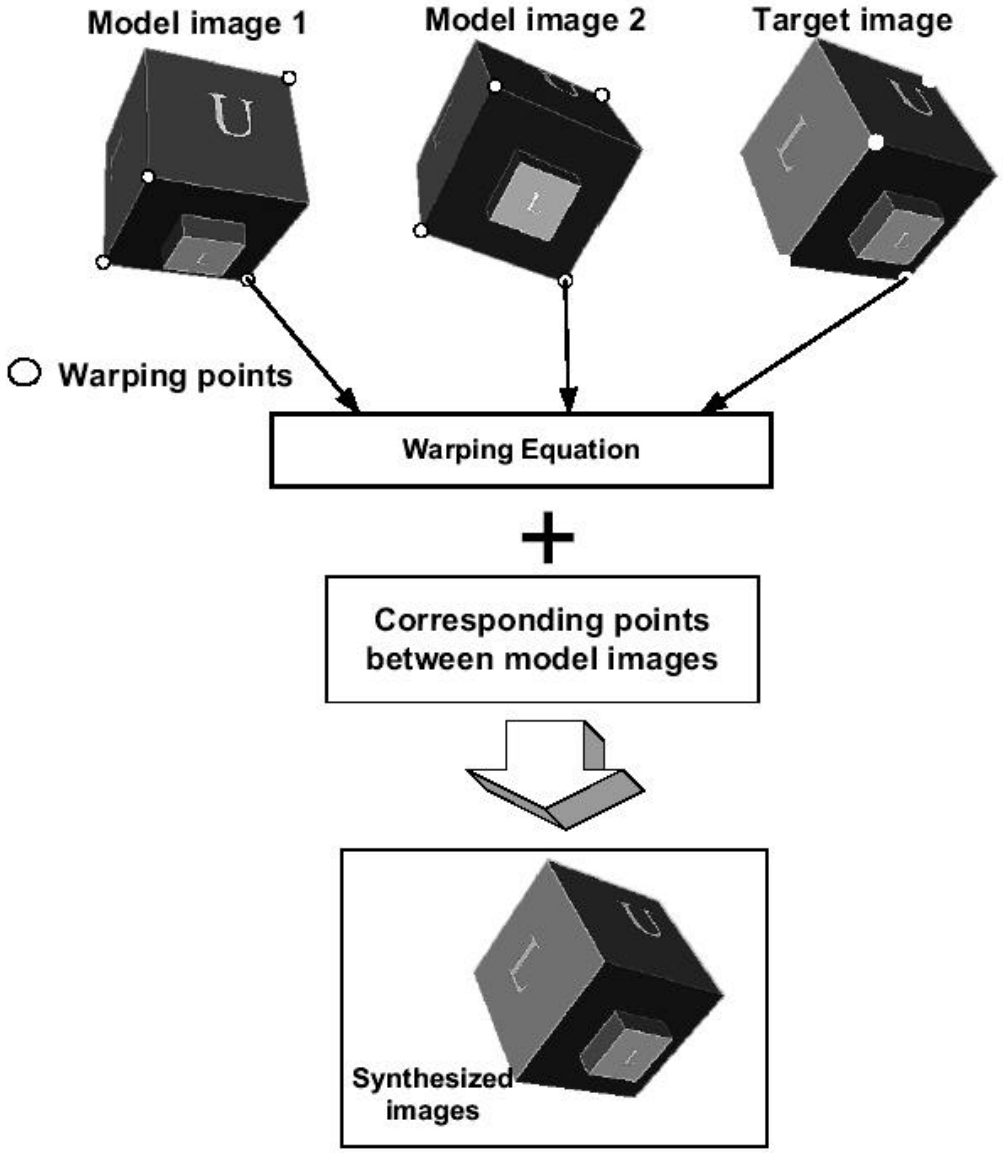


Figure 2 : Vue globale du processus.

Résultats de cette technique

Nous n'avons pas eu le temps nécessaire pour pouvoir tester nous même cette technique, mais voici quelques résultats de reconnaissance d'expressions faciales issus de cette technique.

Image set 1: Wide range of viewpoint

Type of Expression	Test Cases	Accuracy
Happiness	25 images	84%
Sadness	25 images	96%
Surprise	25 images	96%
Anger	25 images	88%
Normal	25 images	88%

Image set 2:Different viewpoint, constant facial expressions

Type of Expression	Test Cases	Accuracy
Happiness	92 images	88%
Sadness	67 images	97%
Surprise	69 images	94%
Anger	71images	90%

Image set 3: Different viewpoint, different facial expressions

Type of Expression	Test Cases	Accuracy
Happiness, Sadness, Surprise, Anger, Normal	50 images	92%

Figure 3 : Résultats de reconnaissance d'expressions faciales.

Ces résultats sont tout à fait corrects et permettent d'espérer pouvoir utiliser cette technique afin dans un premier temps de trouver la position du visage dans l'espace et dans un deuxième temps les coordonnées des points images. En effet, pour reconnaître une expression faciale, il faut auparavant détecter entre autre la bouche dans le visage.

ANNEXE D

CHARGER, CONVERTIR ET AFFICHER UNE IMAGE IPL

img1= **loadIplImage** (adresse , 24);
img= **cvCreateImage**(cvSize(img1->width,img1->height), IPL_DEPTH_8U, 1);
iplColorToGray(img1, img);
adresse : localisation de l'image au format Bitmap que l'on veut convertir en noir et blanc.
Img est l'image convertie en noir et blanc par la fonction **iplColorToGray.**

```
//--------------------------------------------------------------------------
//              METHODE DE CHARGEMENT D'UNE IMAGE IPL
//--------------------------------------------------------------------------
 IplImage * TVisage::loadIplImage(const char * filename, int desiredColor)
{
IplImage * result = NULL;
int filter = 0;
int width, height, color;
int ok = 0, error;
printf("(LdIm");
fflush(stdout);
if( gr_fmt_find_filter == 0 )
{
fprintf(stderr, "> Error while reading image file %s.\n",
filename);
return NULL;
}
if( !(filter = gr_fmt_find_filter( filename )))
{
fprintf(stderr, "> no filter found\n");
goto exit;
}
if( !gr_fmt_read_header( filter, &width, &height, &color ))
{
fprintf(stderr, "> Can't read header\n");
goto exit;
}
color = desiredColor >= 0 ? desiredColor : color > 0;
if (desiredColor == 24)
result = cvCreateImage(cvSize(width, height),
IPL_DEPTH_8U, 3);
else if (desiredColor == -1)
result = cvCreateImage(cvSize(width, height),
IPL_DEPTH_8U, 1);
else if (desiredColor == 1)
result = cvCreateImage(cvSize(width, height),
IPL_DEPTH_1U, 1);
error = gr_fmt_read_data( filter, result->imageData,
result->widthStep, color );
```

```
ok = error != 0;
exit:
gr_fmt_close_filter( filter );
if (!ok)
fprintf(stderr, "> Error while reading image file %s.\n",
filename);
printf(")"); fflush(stdout);
return result;
}
//--------------------------------------------------------------------------
//      METHODE POUR AFFICHER UNE IMAGE IPL DANS
//                  UNE FENETRE OPENGL
//--------------------------------------------------------------------------
void TVisage::ScreenDisplay(IplImage *Img)
{
double proj[16];
glViewport(0, 0,(zoom)*((GLsizei)Img->width),(zoom)*((GLsizei)Img->height));
glMatrixMode(GL_MODELVIEW);
glPushMatrix();
glLoadIdentity();
glMatrixMode(GL_PROJECTION);
glPushMatrix();
glLoadIdentity();
gluOrtho2D(0.0,(GLdouble) Img->width, 0.0,(GLdouble) Img->height);
glGetDoublev(GL_MODELVIEW_MATRIX, proj);
if(Img->nChannels ==1){
glPixelZoom((zoom),-(zoom));
glRasterPos2i(0,Img->height);
}else{
glPixelZoom(1.0,1.0);
glRasterPos2i(0,0);
}
if(Img->nChannels == 1)
glDrawPixels(Img->width, Img->height,GL_LUMINANCE ,GL_UNSIGNED_BYTE,
Img->imageData);
else
glDrawPixels(Img->width, Img->height, GL_RGB,GL_UNSIGNED_BYTE,
Img->imageData);
glPopMatrix();
glMatrixMode(GL_MODELVIEW);
glPopMatrix();
}
```

ANNEXE E

CHARGER LA MATRICE DU MOUVEMENT DE LA CAMERA

```
int xdim =(zoom)*(img->width);
int ydim =(zoom)*(img->height);
plane={0,0,1,0,0,100);
T[0][0]=fx1; //340.790428988;      T[1][0]=0;                          T[2][0]=0;
T[0][1]=0;                         T[1][1]=fy; //341.674680222;        T[2][1]=0;
T[0][2]=(img->width)/4;            T[1][2]=(img->height)/4;            T[2][2]=1;
T[0][3]=0;                         T[1][3]=0;                          T[2][3]=0;
```

On appelle cette fonction avant d'initialiser glut.
FillProjectionMatrix(Mat, T, xdim, ydim, plane);
Mat est la matrice en retour.

Ensuite dans la méthode reshape, on charge la matrice du mouvement de la caméra.
glLoadMatrixf(glMat2glVect();

```
//-------------------------------------------------------------------------
//      METHODE DE REMPLISSAGE DE LA MATRICE
//           DE PROJECTION POUR LA CAMERA
//-------------------------------------------------------------------------
```

void TVisage::FillProjectionMatrix(double Mat[3][4], double Transf[3][4], int xdim, int ydim, double Plane[6]){

```
int i,j;
double zmin   = Plane[4], zmax = Plane[5];
double xratio = 2.0/(double)xdim, yratio = 2.0/(double)ydim, zratio =
2.0/(zmax-zmin);

 double ShiftedTransf[3][4];
 for(i=0;i<3;i++) for(j=0;j<4;j++) ShiftedTransf[i][j]=Transf[i][j];
 ShiftTransform(*ShiftedTransf,1.0,1.0,xratio,yratio,NULL);
  /* transpose and copy 1st,2nd and 4th lines */
 for(j=0;j<4;j++){
     Mat[j][0]=ShiftedTransf[0][j];
     Mat[j][1]=ShiftedTransf[1][j];
     Mat[j][3]=ShiftedTransf[2][j];
 }
/* Plane distance computation in 3rd column */
   for(j=0;j<4;j++)
     Mat[j][2]=Plane[j]*zratio;
     Mat[3][2]=  -(1.0+Mat[3][2]+zmin*zratio);
}
```

```
//-------------------------------------------------------------------------
```

```
void TVisage::ShiftTransform (double *Transform,double dx,double dy,double
sx,double sy,int Normalize){
   double ScaleTranslation[3][3], *Mat=(double *)ScaleTranslation;
int    i;
for(i=0;i<9;i++) *Mat++ = .0;
ScaleTranslation[0][0]=sx;
ScaleTranslation[0][2]=(-dx);
ScaleTranslation[1][1]=sy;
ScaleTranslation[1][2]=(-dy);
ScaleTranslation[2][2]=1.0;
MulMat(*ScaleTranslation,Transform,Transform,3,3,4);
if(Normalize){
double s=fabs(Transform[11]);
if(s>EPS){
for(i=0;i<12;i++,Transform++) *Transform=(*Transform/s);
}
else
fprintf(stderr,"ShiftTransform: Cannot normalize.\n");
}
}
//---------------------------------------------------------------------------
//      METHODE DE MULTIPLICATION DE MATRICES
//---------------------------------------------------------------------------
void TVisage::MulMat(const double *m1,const double *m2,double *m3,int dim1,int
dim2,int dim3){

if (m3==m1 || m3==m2) {
double *maux=TmpMat(dim1,dim3);
MulMat(m1,m2,maux,dim1,dim2,dim3);
FillMatrix(maux,m3,dim1,dim3);
}
else
MulMat1(m1,m2,m3,dim1,dim2,dim3);
}
//---------------------------------------------------------------------------
//      METHODE DE MULTIPLICATION DE MATRICES
//---------------------------------------------------------------------------
void TVisage:: MulMat1(const double *m1,const double *m2,double *m3,int
dim1,int dim2,int dim3){
register int i,j,k;
const double *p1,*p2;
for (i=0,p1=m1;i<dim1;i++,p1+=dim2)
for (j=0;j<dim3;j++) {
*m3=0;
for (k=0,p2=m2;k<dim2;k++,p2+=dim3){
*m3+=p1[k]*p2[j];
}
m3++;
}
}
//---------------------------------------------------------------------------
```

```
//  METHODE POUR METTRE LE MODELE A L'ENDROIT
// SI IL EST A L'ENVERS
//  ici, on ne l'utilise pas car le modèle est déja dans le bon sens.
//---------------------------------------------------------------------------
void TVisage::FlipglMat(void){

                        //glMat.SetValue(0, 1, -glMat.GetValue(0, 1));
                        //glMat.SetValue(1, 1, -glMat.GetValue(1, 1));
                        //glMat.SetValue(2, 1, -glMat.GetValue(2, 1));
                        //glMat.SetValue(3, 1, -glMat.GetValue(3, 1));
}
//---------------------------------------------------------------------------
// REMPLIR UN VECTEUR AVEC LES VALEURS DE LA MATRICE
//---------------------------------------------------------------------------
float * TVisage::glMat2glVect(void){

for(int i = 0; i < 4; i++)
for(int j = 0; j < 4; j++)
GLvector[j+4*i] = Mat[i][j];
return(GLvector);
}
//---------------------------------------------------------------------------
//      REMPLIR UN VECTEUR AVEC LES VALEURS PASSEES EN
//      PARAMETRE
//---------------------------------------------------------------------------
void TVisage::FillVect(const double *v1,double *v2,int n){

int i;
if(v1)
for(i=0;i<n;i++) *v2++=*v1++;
else
memset(v2,0,n*sizeof(double));
}
```