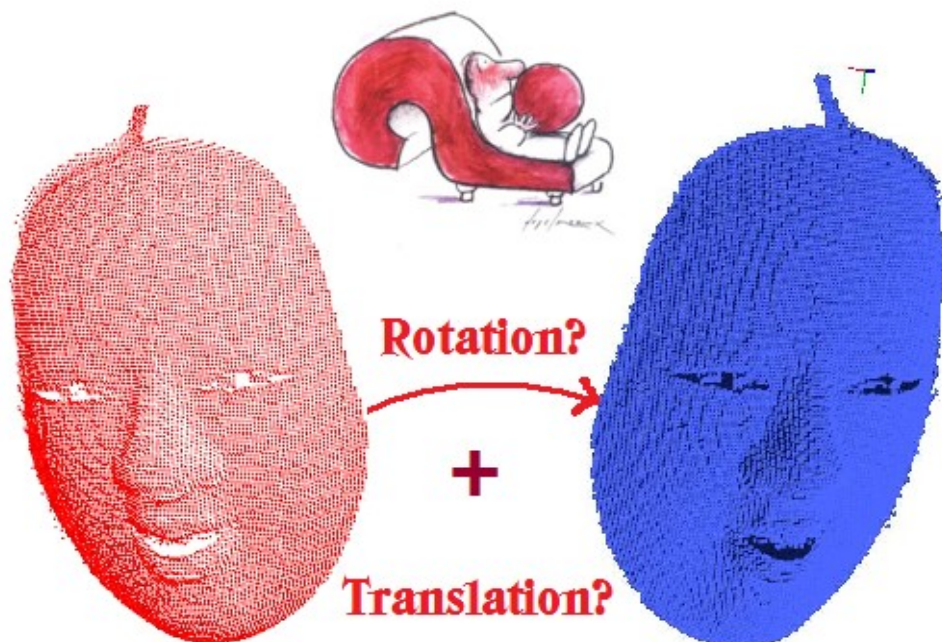
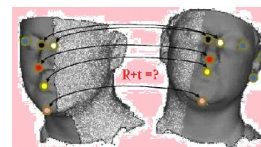


Projet I4: Estimation de deux positions différentes d'un visage 3D

- **Projet réalisé par:**
 - Hasna CHARFI
 - Islem REKIK
 - Guillaume BALAINE
 - Lilia SAYOUD
 - Mengmeng CAI
- **Suivi par: Mme Yukiko KENMOCHI**





SOMMAIRE

INTRODUCTION GENERALE DU PROJET.....	5
--------------------------------------	---

SECTION 1: THEORIE MATHEMATIQUE

INTRODUCTION: ETUDE THEORIQUE.....	7
------------------------------------	---

CHAPITRE 1: NOTIONS DE BASE	8
-----------------------------------	---

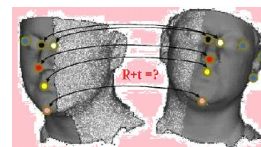
1. ETUDE PRELIMINAIRE.....	8
2. CALCUL DU VECTEUR NORMAL.....	10
3. CALCUL DES RAYONS DE COURBURES.....	11
4. CHOIX DU VOISINAGE.....	12

CHAPITRE 2: METHODES DE RECHERCHE DES PARTICULARITES DE L'IMAGE:	
--	--

1. ETUDE PRELIMINAIRE.....	14
2. EGI.....	14
3. POINTS PARTICULIERS.....	22
4. PLANS TANGENT STABLES.....	25
5. PLANS STABLES AVEC CONTRAINTES.....	27
6. DETERMINATION DES COURBURES.....	30

CHAPITRE 3: IDENTIFICATION DE LA TRANSFORMATION ENTRE DEUX POSITIONS DIFFERENTES.....	
--	--

1. ETUDE PRELIMINAIRE.....	31
2. LA TRANSLATION.....	32
2.1 Calcul.....	32
2.2 Application de la translation.....	32
3. LA ROTATION.....	33
3.1 Calcul.....	33
3.2 Application de la rotation.....	35

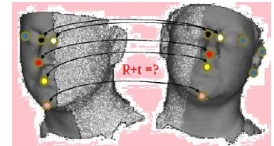


SECTION 2: ETUDE PRATIQUE (PROGRAMMATION JAVA)

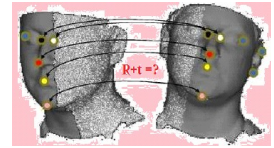
INTRODUCTION: IMPLEMENTATION EN JAVA.....	38
CHAPITRE 1: CLASSES DES NOTIONS DE BASE.....	40
1. La mise en place.....	40
2. L'implémentation des méthodes en Java.....	42
3. Vers une visualisation des patterns et de la généricité.....	43
CHAPITRE2: GENERATION DES FICHIERS DE FORMAT TXT ET WRL.....	45
CHAPITRE 3: RECHERCHE DE PARTICULARITES.....	47
1. EGI.....	47
2. Classes pour la visualisation des courbures.....	51
3. Classes pour les plans tangents et plans stables.....	52
4. Classes pour les plans tangents et stables avec contraintes.....	54
5. Classe pour la localisation des points particuliers.....	55
CHAPITRE 4: IDENTIFICATION DE LA TRANSFORMATION.....	56
1. Translation.....	56
2. Rotation.....	57
CHAPITRE 5: INTERFACE GRAPHIQUE	60
1. Utilisation de JavaView.....	61
2. Les boutons de l'interface graphique.....	62
3. Les classes créées en Java.....	68

SECTION 3: RESULTATS ET DISCUSSIONS

INTRODUCTION: VISUALISATION & ANALYSE DES RESULTATS.....	70
CHAPITRE 1: VISUALISATION DES PARTICULARITES	71
1. EGI.....	71
2. Vérification de la bonne détection du voisinage.....	72



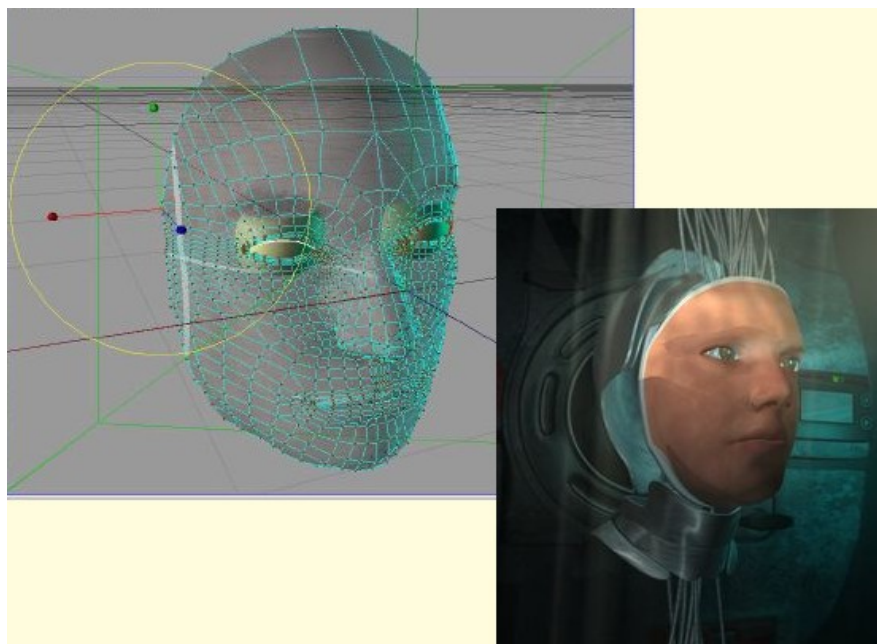
3. Visualisation des courbures.....	73
4. Visualisation des points particuliers.....	77
5. Visualisation des plans stables.....	78
 CHAPITRE 2: ANALYSE DE LA TRANSFORMATION GEOMETRIQUE.....	 81
 CHAPITRE 3: FAILLES ET SOLUTIONS POSSIBLES.....	 83
CONCLUSION.....	88
REMERCIEMENTS.....	90
BIBLIOGRAPHIE.....	91



INTRODUCTION GÉNÉRALE DU PROJET

L'acquisition et l'interprétation du mouvement du visage humain est un des sujets les plus importants dans le domaine de la vision par ordinateur, les applications concernant ce sujet sont nombreuses telles que les interfaces homme-machine, les systèmes de sécurité, les systèmes de télécommunication...

Dans le cadre de ce projet, nous envisageons le problème de l'estimation de la position d'un visage 3D à partir d'un nuage de points acquis par un scanner 3D.

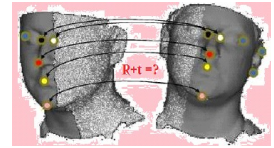


Plus précisément, on cherche à **caractériser la rotation et la translation entre deux positions différentes d'un visage humain**. Et pour faciliter le type de données utilisés, on utilisera un masque humain dont les expressions seront fixes en fonction du temps.

Pour cela, il suffit de **localiser trois points invariants** dans les deux positions et les transformations géométriques seront complètement définies.

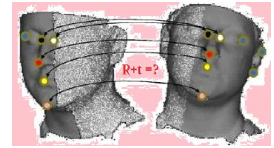
Dans ce rapport, on exposera d'une manière chronologique les différentes étapes de la réalisation de ce projet:

- *L'étude théorique du projet (mathématique et algorithmes)*
- *L'implémentation de la partie théorique en Java.*
- *La visualisation des résultats à l'aide du VRMLVIEW puis l'analyse et la synthèse.*

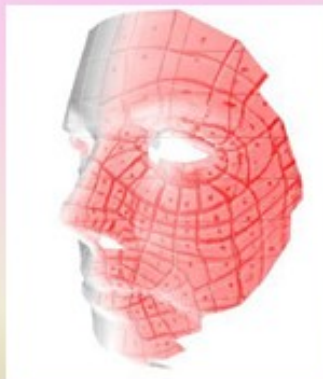


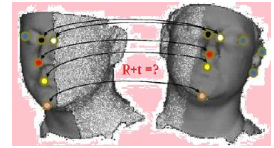
Et en ce qui concerne **le managment du projet**, on peut affirmer que suite à une étude préliminaire, une partie non négligeable du temps de travail a été consacrée à l'élaboration de techniques de management de projet pour rendre le partage du travail et le travail de groupe plus efficaces. Ainsi un site internet en PhP ainsi qu'un serveur de partage de fichier dynamique de type **cv**s et un **wiki** pour y analyser les documents ont été mis en place dès le départ pour faciliter la diffusion du travail.

Malgré tout lorsque le volume de travail sur le projet a atteint son pic il aurait alors fallu avoir un serveur de discussion de type Irc qui permette d'échanger les nouvelles plus rapidement qu'à travers un serveur cvs, étant donné que le client utilisé qui est un plug-in sous l'IDE d'Eclipse ne rafraîchi pas les données automatiquement.



► I- ETUDE THEORIQUE DU PROJET





I- INTRODUCTON: ETUDE THEORIQUE

La partie fondamentale de l'étude de ce projet est la partie théorique. En effet, l'étude des particularités d'une image 3D se base sur l'étude de ces caractéristiques topologiques et géométriques qui seront exploitées pour définir les algorithmes de recherche des zones (ensemble de points) particulières de l'image traitée.

L'étude théorique revient à recherche des algorithmes efficaces qui nous permettront de trouver les trois points essentiels qui nous permettront de définir la rotation et la translation.

Dans une première partie, on va définir théoriquement les outils de base (*définition du voisinage, définition du vecteur normal et la définition des rayons de courbures*) qu'on a utilisés et qui influenceront sur tous les résultats obtenus. En effet, en changeant les définitions utilisés pour ces outils on pourra améliorer les résultats et vis vers-ça.

Ensuite, on énoncera les algorithmes utilisés pour la recherche des zones particulières (points, plans, EGI...).

La dernière partie théorique sera consacrée pour présenter la théorie du calcul de la translation et la rotation qui sera basée sur les théories précédemment établies.

En effet, on peut schématiser notre projet dans la figure ci-dessous qui représente l'interconnexion entre toutes les théories établies et les définitions données et l'influence qui se propage le long du projet d'une partie vers une autre:

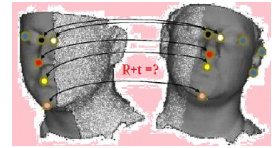
Schéma des trois parties essentielles du projet



* Chaque étape est la base de l'étape suivante.

* Les définitions établies pour une étape choisie influent sur la suivante.

NB: Dans ce rapport, la notation '*kappa 1*' est équivalente à $K1$ et celle de '*kappa 2*' est équivalente à $K2$ qui symbolisent les rayons de courbures.



CHAPITRE 1: NOTIONS DE BASE

1. Etude préliminaire:

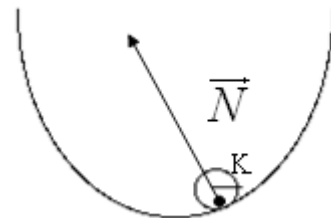
Avant de pouvoir commencer à étudier la structure du programme correctement adaptée au projet, il faut connaître les outils mathématiques dont on va se servir dans les algorithmes et les calculs des outils d'évaluation de position du visage.

Ces outils mathématiques doivent être certifiés, robustes et adaptés au calcul en informatique et à la 3D. Ils peuvent être aussi simple qu'un vecteur ou aussi compliqué qu'une donnée physique, l'intérêt est avant tout que leur calcul soit compatible avec la structure et le type de données utilisées.

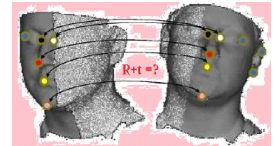
Etant donné qu'il s'agit dans ce projet avant tout de trouver l'orientation et la direction d'une face ou d'un visage nous avons été amenés à chercher des éléments qui permettent de les définir, avec à la base trois méthodes principales : trouver un plan de symétrie, calculer la sphère gaussienne étendue, trouver des points particuliers grâce au rayon de courbure.

De manière à explorer le plus de méthodes possibles il faut utiliser les outils les plus communément repris qui sont le vecteur normal à la surface $\vec{N}$ qui permet de donner la direction du plan du voisinage du point et la courbure K qui permet de déterminer la concavité ou convexité du voisinage du point.

Ces outils mathématiques conviennent bien au traitement informatique car ils consistent en quasi totalité de multiplications, divisions, additions et soustractions ce qui permettra de traiter des images de grande taille avec une architecture adaptée.

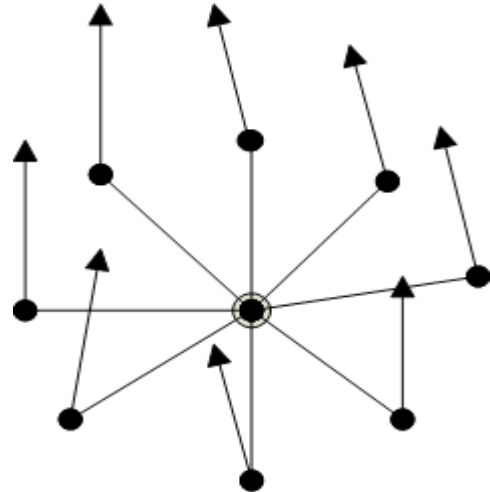


Malheureusement comme la plupart des outils mathématiques bruts ils sont à la base créés dans un espace continu, alors que nous travaillons sur des échantillons discrets d'image télémétriques. Deux possibilités s'offrent donc à nous, adapter les formules de calcul ou bien utiliser des outils de normalisation et de continuité mathématique pour adapter les données discrètes. C'est la première méthode qui va nous intéresser, on cherche à minimiser la complexité de calcul, car ces outils seront repris par les nombreuses méthodes implémentées et doivent être 'dénudés' au possible.

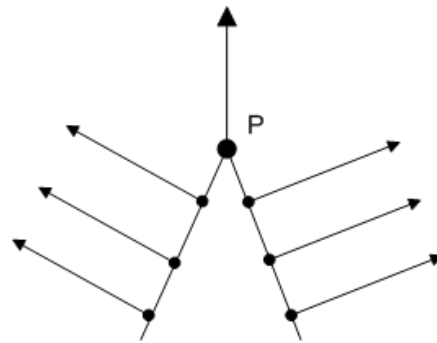


2. Calcul du vecteur normal:

Cet outil mathématique puissant va devoir permettre de déterminer l'orientation du plan sur lequel se trouve un point, sa robustesse dépend essentiellement de la taille du voisinage que l'on utilise pour le calcul.

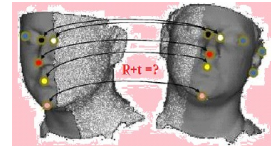


Au départ, l'idée était de faire une moyenne des produits vectoriels de couples de vecteurs entre le point d'origine du voisinage et les points du voisinage sur un Γ_8 mais cela ne donnait qu'une idée partielle de l'orientation du plan.



Le calcul a été alors porté sur un voisinage plus grand mais dans la réalité même si la donnée paraît plus robuste, cela uniformise les points et crée des ensembles de points qui ont des vecteurs normaux quasi identiques alors qu'ils n'ont pas forcément la même orientation. La solution adéquate est donc au final la moyenne de tous les produits vectoriels de couples de vecteurs non colinéaires formés avec trois points du voisinage.

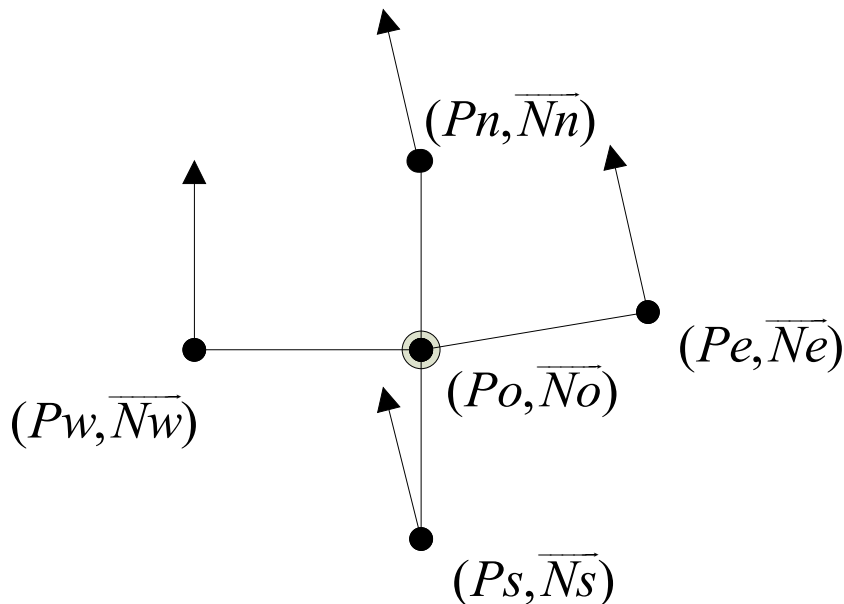
La combinaison de tous ces vecteurs donne une idée assez précise de l'orientation.

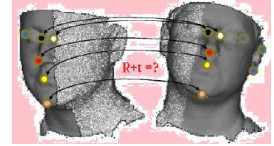


3. Calcul des rayons de courbures:

Le calcul des courbures est un domaine et même une discipline assez vaste, cet outil mathématique est encore à l'étude dans de nombreux domaines notamment le traitement d'objets 3D et il n'existe pas de définition particulière qui existe pour tout type de forme. Néanmoins il existe des outils puissants pour le voisinage discret proche d'un point, notamment Γ_4 (aussi nommé ensemble des points cardinaux ou 4-voisinage).

En ce qui concerne le visage, l'étude de la courbure d'un point unique n'apporte aucune information utilisable directement, en effet plus la précision augmente plus les imperfections du visage deviennent du bruit gênant et le calcul local de la courbure détecte certes alors ces imperfections, mais alors il devient impossible de dire avec certitude si tel ou tel point se trouve sur la zone la plus convexe ou concave à partir de lui seul. Il faudrait des outils qui soient capables de gérer le calcul de κ sur un voisinage étendu, c'est-à-dire au moins Γ_8 ou bien Γ_{24} . Une autre utilisation robuste de l'outil tel qu'il est pris dans ce projet est de comparer plusieurs zones de points selon la valeur de la courbure et ainsi regrouper des zones moyennement convexes ou concaves.





♦ Le point o et ses cardinaux en 3D:

$$\left(\begin{array}{l} dPu = Pe - Pw \\ dPv = Ps - Pn \\ Ru = dPu - No(No.dPu) / 2 \\ Rv = dPv - No(No.dPv) / 2 \\ Nu = Ne - Nw / 2 \\ Nv = Ns - Nn / 2 \end{array} \right) \quad \left(\begin{array}{l} E = Ru.Ru \\ F = Ru.Rv \\ G = Rv.Rv \\ L = -Ru.Nu \\ N = -Rv.Nv \\ M = -(Ru.Nv + Rv.Nu) / 2 \\ K = (E.G - F.F) / (L.N - M.M) \\ H = (E.N - 2.F.M + G.L) / 2(E.G - F.F) \\ K_1 = H + \sqrt{H^2 + K} \\ K_2 = H - \sqrt{H^2 + K} \end{array} \right)$$

♦ Les équations discrètes du calcul de K sur Γ_4

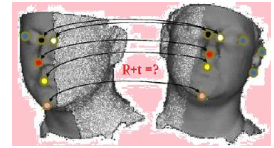
On fait la corrélation entre l'orientation des plans définis par le voisinage du point, pour ainsi obtenir les deux éléments de la courbure moyenne. Le problème majeur de cette méthode est qu'elle est très locale, et que suivant la précision des données on peut obtenir systématiquement des courbures infinies si les points sont sur le même plan.

Cela est logique puisque le rayon de courbure d'un plan est infini mais cela peut donner des résultats inutiles si l'échantillon de donnée contient trop de points sur le même plan que leur voisinage.

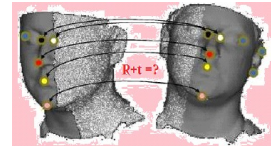
4. **Choix du voisinage:**

Le choix du voisinage se fait essentiellement au niveau des résultats observés et en fonction du type de données. En effet, selon l'architecture de la machine et du programme, l'exécution du calcul sera plus ou moins longue, il s'agit là essentiellement du calcul du vecteur normal moyen. Le calcul du vecteur normal est de complexité exponentielle car il doit prendre en compte la totalité des combinaisons de vecteurs du voisinage pour être assez robuste.

Cela reste raisonnable sur Γ_8 avec quelques centaines de vecteurs mais dès lors que l'on est sur Γ_{24} il y a plusieurs dizaines de milliers de combinaisons de vecteurs possibles et ce n'est plus du tout la même problématique. De plus, si les



données ne sont pas assez précises, il est plus efficace de développer des outils qui vont augmenter leur précision plutôt que de chercher à augmenter la taille du voisinage, car cela faussera systématiquement les calculs des rayons de courbure.



CHAPITRE 2: METHODES DE RECHERCHE DES PARTICULARITES DE L'IMAGE:

1. ETUDE PRELIMINAIRE:

Pour la détermination de la transformation géométrique entre deux positions différentes, on a étudié la topologie de l'image pour pouvoir déterminer des particularités qui fourniront une base pour la recherche du vecteur de translation 3D et de l'axe et l'angle de la rotation 3D.

Comme étude préliminaire, on a cherché de documents scientifiques qui puissent nous être utiles et nous donner une idée sur les algorithmes qu'on puisse implémenter pour la détection de ces particularités. Plusieurs algorithmes étaient intéressants comme celle de l'ICP (Iterative Closest Point) ou algorithme de Kuhn mais on n'a pas pu les implémenter car ils ne respectent pas la topologie et la structure des données dont on dispose.

En effet, tous les points de l'image sont représentés dans un tableau "data" unidimensionnel qui contient les valeurs de la profondeur "z" (coordonnée en espace 3D d'un point appartenant à l'image) et dont l'index varie selon les deux autres coordonnées x et y ($\text{index} = x + w * y$, avec w =largeur de l'image). En plus les trois coordonnées sont des entiers.

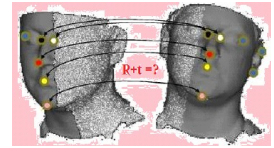
--> la structure de l'image étudiée est digitalisée et représentée dans un espace discret. Par conséquent, l'algorithme de l'ICP ne peut être utilisé car il est efficace pour une structure de donnée continue (notion de surface).

Par ailleurs, on a commencé à étudier la topologie et l'aspect géométrique différentiel de l'image pour pouvoir déterminer les clés de la transformation recherchée. Pour cela, on utilisera les outils de base qu'on a définis dans le premier chapitre.

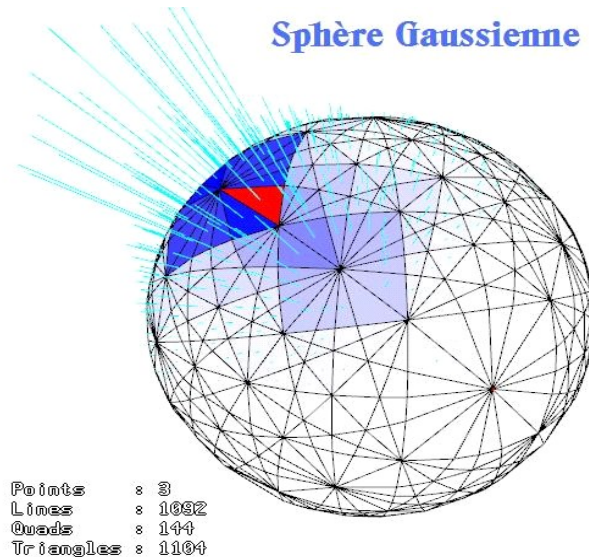
2. EGI (Extended Gaussian Image):

En partant de deux nuages de points pris pour deux positions différentes d'un visage, l'idée consiste à représenter les EGI de ces deux nuages en calculant les projections des vecteurs normaux des deux surfaces sur deux sphères unitaires qu'on appelle sphères gaussiennes.

Ensuite, on compare les deux représentations en se basant sur une technique de la corrélation de deux EGI calculées.

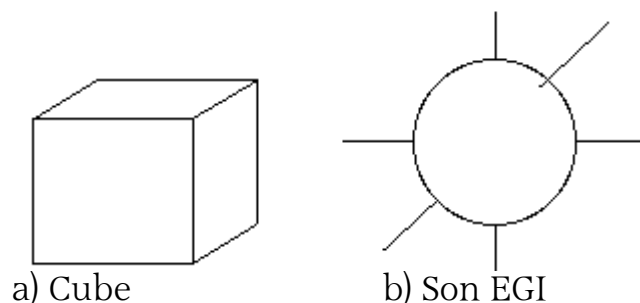


L'EGI ou Extended Gaussian Images est une distribution de tous les vecteurs normaux unitaires de la surface d'un visage projetés sur une sphère unitaire.

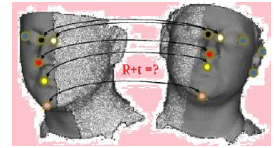


Cette méthode permet de connaître les particularités du visage parce qu'elle traite et diffère facilement entre les cas des objets convexes et cylindriques ainsi que les plans. Cette particularité est acquise par le fait qu'on a trois cas différents selon la forme ou les formes des parties d'un objet quelconque.

Dans le cas d'un plan, tous les points qui lui appartiennent ont des vecteurs normaux colinéaires donc ils ont le même vecteur normal unitaire, et par conséquence la représentation de l'EGI sera un point sur la surface de la sphère unitaire de telle sorte que ce point constitue avec le centre de la sphère le même vecteur que le vecteur normal unitaire des points du plan. (voir exemple)



Et dans le cas d'une distribution cylindrique des points sur une surface donnée, l'EGI est un ensemble des points qui sont plus au moins sur une courbe de la surface de la sphère gaussienne.



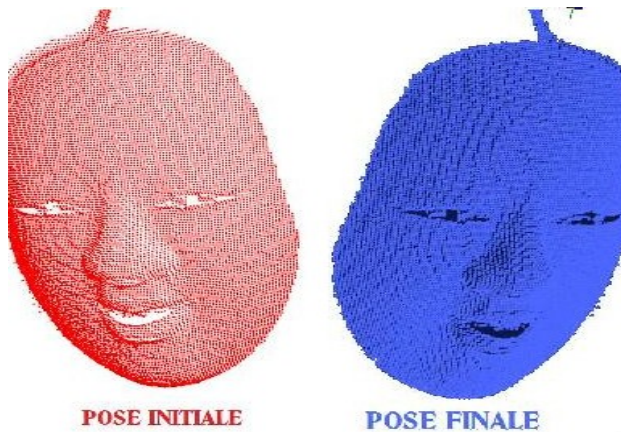
Le but de cette méthode est donc de représenter deux EGI de deux nuages différents d'un visage, pour pouvoir après comparer ces deux nuages par une technique basée sur la corrélation entre les deux EGI obtenues. La corrélation de sa part, se base sur un coefficient appelé coefficient de corrélation sphérique et mesure la similarité entre les directions des vecteurs normaux projetés sur la sphère gaussienne. Récemment, le coefficient proposé par Fisher est applicable pour la reconnaissance des objets de trois dimensions, soient deux vecteurs $\{X_1, X_2, \dots, X_m\}$ et $\{Y_1, Y_2, \dots, Y_m\}$ telle que chaque X_i correspond à Y_i , la matrice de corrélation est définie par la formule suivante:

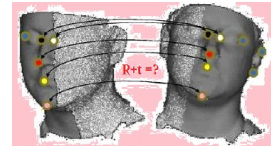
$$N = E(X \cdot Y^t)$$

Avec $E[\dots]$ est la somme des toutes les permutations de n vecteurs parmi m vecteurs.

Le coefficient de corrélation, donné par la formule ci-dessus, est la normalisation du déterminant de la matrice de corrélation N et il permet de mesurer la similarité entre deux images. Il est donné par la formule suivante:

$$\begin{aligned} \rho &= C \cdot \det\{E(X \cdot Y^t)\} \\ &= \frac{\det\{E(X \cdot Y^t)\}}{\sqrt{\det\{E(X \cdot X^t)\} \det\{E(Y \cdot Y^t)\}}} \end{aligned}$$





Une autre façon de procéder, mais en travaillant toujours sur la sphère gaussienne, est de distinguer entre les points de l'image en déterminant la nature de la partie à laquelle ils appartiennent, autrement dit, il faut voir si un point donné de l'image appartient à une partie plane de l'image, à une partie cylindrique ou à une partie concave.

Comme ça on peut travailler sur différentes parties, il y a des points qui se trouvent dans des endroits ayant plus ou moins la forme d'un plan comme par exemple le front et les joues, d'autres endroits qui sont un peu cylindrique comme le nez ou concaves comme les yeux.

Pour différencier entre les points de l'image d'entrée, il nous faut un algorithme qui précise le type de chaque point.

L'idée générale est de parcourir l'image et de calculer pour chaque point la distance maximale entre les vecteurs normaux de ses huit voisins projetés sur la sphère gaussienne. Ensuite, on compare cette distance à un seuil, alors si elle lui est inférieure, on associe ce point à l'ensemble des points qui se trouvent dans des plans de l'image, sinon on le classe parmi les points qui peuvent appartenir soit à des parties concaves, convexes ou cylindriques.

Dans ce cas, on applique un deuxième algorithme qui détermine le maximum de la distance entre les projections des projections précédentes sur le cercle perpendiculaire au plan défini par les projections des vecteurs normaux.

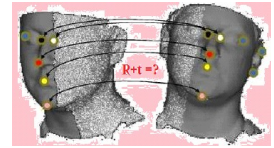
Un fois trouvée, on compare le maximum de la distance à un deuxième seuil, s'il est supérieur au seuil on dit que le point correspondant appartient à une partie concave ou convexe et sinon il est nécessairement sur un "cylindre" du visage.

Au niveau de la visualisation, on cherche à colorier chaque type de points par une couleur donnée, par exemple dans notre projet on a choisi la couleur rouge pour les point du plan, la couleur verte pour les points qui occupent les parties convexes et la couleur bleue pour les points qui appartiennent à des zones cylindriques du visage.

- Points appartenant à un plan:

Comme on l'a déjà expliqué dans le paragraphe précédent, il s'agit de parcourir les points de l'image sur laquelle on travaille et de déterminer les points dont la distance maximale calculée entre les projections des vecteurs normaux de ses huit voisins sur la sphère gaussienne est inférieure à un seuil bien précis.

L'algorithme de la détermination de la distance maximale est:



**** Idée sur l'algorithme de la détermination de la distance maximale****

Initialisation:

table = tableau contenant tous les vecteurs normaux des voisins du point traité.

double k=0; // initialisation de k

Algorithme:

```

pour tout i de 0 à Taille du tableau faire
{
    pour tout j de i+1 à la taille du tableau faire
    {
        si les coordonnées z de table[i] et de table[j] sont différents
de 0
        alors{
            double distance = distance entre table[i] et table[j]
            si distance >= k alors
            {
                k= distance;
            }
        }
    }
retourner k;
}

```

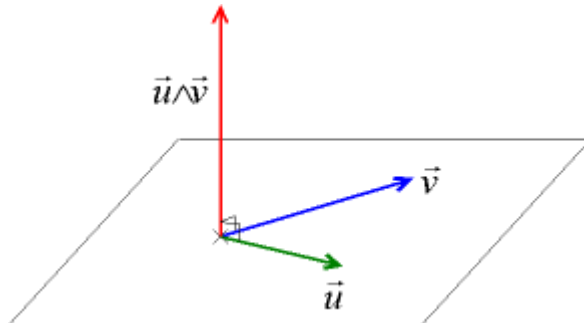
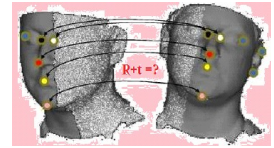
*** Complexité de l'algorithme:**

La complexité de cet algorithme est de **$O(n^2)$** avec **n**=taille du tableau

Un deuxième algorithme possible est de déterminer le maximum de l'angle que font deux vecteurs normaux de deux voisins du point traité: il s'agit dans ce cas de calculer le sinus de chaque angle qui est donné par la formule suivante:

Si on a deux vecteurs u et v alors: $\sin(\widehat{\vec{u}, \vec{v}}) = \|\vec{w}\| / \|\vec{u}\| \cdot \|\vec{v}\|$

où w est le produit vectoriel de u et v.



$$\|\vec{w}\| = \|\vec{u}\| \cdot \|\vec{v}\| \cdot \sin(\widehat{\vec{u}, \vec{v}})$$

** Idée sur l'algorithme de la détermination de l'angle maximal **

Initialisation:

Table = tableau contenant tous les vecteurs normaux des voisins du point traité.

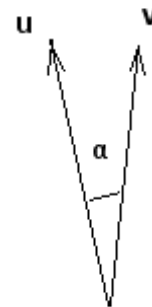
double sin_alpha=0; // initialisation de k

Algorithme:

```

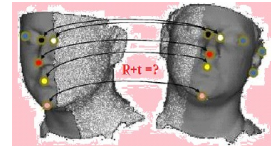
pour tout i de 0 à Taille du tableau faire
{
    pour tout j de i+1 à la taille du tableau faire
    {
        si les coordonnées z de tableau[i] et de tableau[j] sont
différents de 0 {
            v = produit vectoriel de table[i] et table[j].
            On définit
            double sin_gamma=
norme(v)/norme(table[i])*norme(table[j])
            si (sin_gamma>=sin_alpha) alors{
                sin_alpha = sin_gamma;
            }
        }
    }
}
retourner sin_alpha;

```



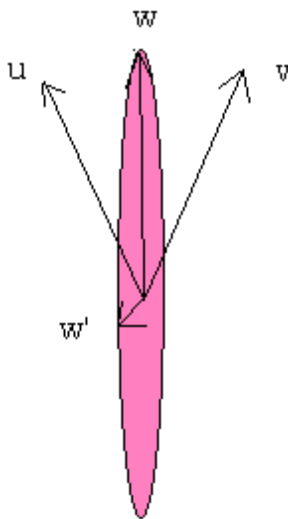
* **Complexité de l'algorithme:**

La complexité de cet algorithme est de $O(n^2)$ avec n =taille du tableau
(On parcourt le tableau deux fois successives)



- Points appartenant à des parties convexes, concaves ou cylindrique

Afin de différencier entre les points qui appartiennent à des zones concaves ou convexes du visage et ceux qui appartiennent à des zones cylindriques, il nous faut un deuxième algorithme qui calcule le maximum de la distance qui peut exister entre les projections des points qui sont à leur tour les projections des vecteurs normaux des voisins d'un point 3D donné, sur le plan du cercle perpendiculaire au plan défini par l'ensemble de ces points.



Une fois on a déterminé les deux vecteurs u et v qui forment l'angle maximal par rapport aux autres, la plan du cercle sera défini par le vecteur w qui est la somme de u et v et par le vecteur w' qui est le produit vectoriel de u et v .

**** Idée sur l'algorithme de la détermination de la projection d'un vecteur normal sur le plan du cercle ****

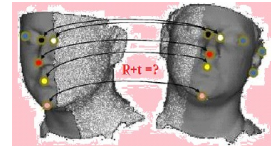
Entrée: PointComplete3D p , Vecteur Normal v

Initialisation:

On définit **proj** comme un vecteur

Algorithme:

```
si le vecteur normal  $u$  de  $p$  n'est pas nul alors{
  on définit
  double  $t = u \cdot v / \text{norm}(u)$  ;
```



```
double x' = t*(u(x)+ v(x));  où u(x) est l'abscisse de u ...
double y' = t*(u(y)+ v(y));  où u(y) est l'ordonnée de u ...
double z' = t*(u(z)+ v(z));

proj(x)= x';
proj(y)= y';
proj(z)= z';
}
sinon {
    proj(x)= 0;
    proj(y)= 0;
    proj(z)= 0;
}
retourner proj;
}
```

*** Complexité de l'algorithme:**

la complexité de cet algorithme est en temps constant

Maintenant il nous faut un autre algorithme qui calcule les distances entre les différentes projections sur le plan du cercle.

**** Idée sur l'algorithme de la détermination du maximum de la distance entre les différentes projections ****

Entrée: PointComplete3D **p**

Initialisation:

on définit

double k=0;

projs = ensemble des projections des vecteurs normaux des voisins d'un point

Algorithme:

```
pour tout i de 0 à la taille de projs faire{
    pour tout j de i+1 à la taille de projs faire{
        double distance = distance entre projs[i] et projs[j];
        si distance >= k alors{

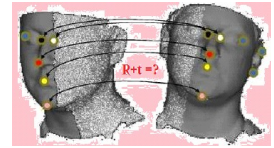
            k=distance;

        }
    }
}retourner k;}
```

*** Complexité de l'algorithme:**

La complexité de cet algorithme est de **$O(n^2)$** avec **n**=taille du tableau

(On parcourt le tableau deux fois successives)



Après ces deux algorithmes, on utilise un troisième algorithme se basant sur ces deux derniers et qui détermine le type de chaque point du visage.

**** Idée sur l'algorithme qui décide le type de chaque point du visage****

Entrée: Point Complete3D

Algorithme:

*si la distance calculée par l'algorithme précédent est $<$ seuil
--> le point appartient à une zone du cylindrique visage
sinon --> ce point appartient à une partie concave ou convexe
du visage.*

*** Complexité de l'algorithme:**

La complexité de cet algorithme est en temps constant

3. POINTS PARTICULIERS:

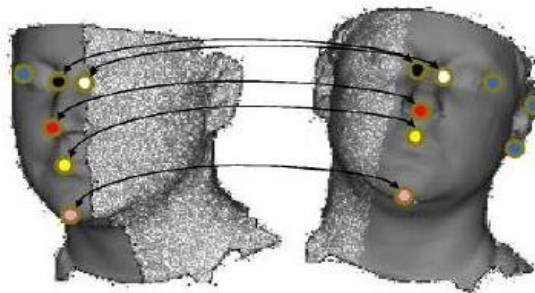
La détermination des points particuliers nous aidera à établir la correspondance ou “matching” entre les deux positions différentes.

La détection des points particuliers était basée sur l'étude des convexités et des concavités des images. Par exemple, on essaiera de déterminer le bout du nez (l'extremum de la convexité) et les yeux.

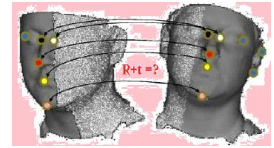
La résolution de cette *problématique* qui est la détermination de l'axe et l'angle de la rotation et du vecteur de translation nécessite l'identification de **trois points particuliers**... Par conséquent, la correspondance sera complètement définie et la transformation adéquatement calculée.

Plus l'erreur de la localisation est minimisée, plus la correspondance converge vers de estimations meilleures.

Détection de points particuliers



→ Correspondance établie entre les
deux poses différentes.



◆ Détection du nez:

En exécutant l'algorithme de la détection du nez, on parcourt tous les points de l'image et on extrait celui qui a des valeurs de kappa1 et kappa2 positives avec la somme de leurs carrés maximale en y ajoutant la contrainte que sa 3ème coordonnée z soit maximale.

**** Idée sur l'algorithme de la détection du nez ****

Initialisation:

entier temp =0, temp_z=0;

point p_temp;

Algorithme:

Pour tout point p appartenant à l'image faire

```
{
    si (kappa1 et Kappa2 sont positives) alors
    {
        calculer la somme S des carrés des kappa1 et kappa2 du point p
        si( S>temp & p.getZ()>temp_z) alors{
            temp=s; temp_z=p.getZ();
            p_temp=p;}
    } //fin si
} //fin for
```

retourner p;

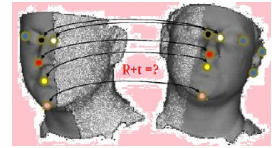
NB: *p.getZ()* --> renvoie la valeur de coordonnée z du point p.

*** Complexité de l'algorithme:**

Cet algorithme est en temps linéaire: **O(n)** avec **n**=taille de l'image.

◆ Détection des yeux:

Pour détecter les yeux, on a utilisé une méthode figurant dans un document scientifique (voir bibliographie, [doc1](#)) et qui explique le déroulage de l'algorithme d'ICP et la détection de trois zones particulières du visage (les yeux et le nez).



Pour cela, on calcule en utilisant les deux valeurs qui définissent l'aspect géométrique de l'image κ_1 et κ_2 (deux rayons de courbures convexes et concaves) le “**shape index**” S_p de chaque point.

La formule mathématique utilisée est la suivante:

$$S_p = 0.5 - \text{atang}((\kappa_1 + \kappa_2) / (\kappa_1 - \kappa_2))$$

$$0 < S_p < 1$$

Théoriquement, les yeux ont les valeurs du S_p qui sont les plus proches de zéro. Pour cette raison, on a défini un seuil s pour la sélection des points qui ont des valeurs du $S_p < s$. --> Le choix des points est limité par la contrainte du seuil.

** Idée sur l'algorithme de la détection des yeux**

Initialisation:

entier seuil = s ;

Liste list = vide;

Algorithme:

Pour tout point p appartenant à l'image faire

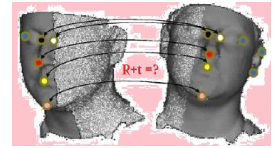
```
{
    calculer  $S_p$  du point  $p$ 
    si  $S_p < s$  alors{
        list.ajouter( $p$ )
    }
}
```

} //fin for

retourner list;

* Complexité de l'algorithme:

Cet algorithme est en temps linéaire: $O(n)$ avec n =taille de l'image.

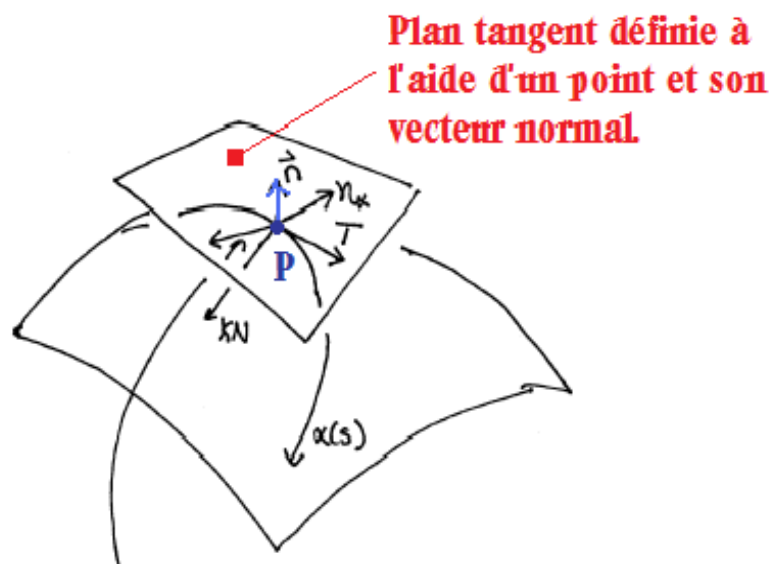


4. PLANS TANGENTS STABLES:

Dans cette étape, on va utiliser une autre méthode pour la détermination des particularités des images. En s'inspirant de l'idée des plans stables figurant d'un document scientifique doc2. (voir *bibliographie, doc 2*), on a eu l'idée de définir les plans tangents stables comme étant les plans qui contiennent le maximum de points appartenant à l'image.

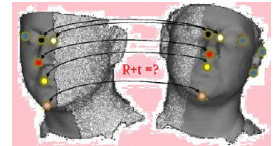
Pour cela, on va déterminer tous les plans tangents puis on choisit les trois premiers plans qui ont successivement les valeurs les plus grandes des points appartenant à l'image.

L'équation du plan tangent est complètement définie par un point p et son vecteur normal qu'on a déjà calculé (un outil de base). (voir figure suivante)



- Coordonnées du vecteur normal (a, b, c)
- Equation du plan tangent: $aX + bY + cZ + D = 0$
- D est déterminée à l'aide du coordonnées du point P .

Les trois vecteurs normaux de ces plans stables serviront pour le calcul de la translation et de la rotation entre les deux positions du visage.



**** Idée sur l'algorithme de la détection des plans tangents stables***

Initialisation:

entier seuil =s;

Lste list = vide;

Algorithme:

Pour tout **p** plan tangent de l'image faire $O(n)$

{

calculer **n** le nombre de points appartenant à ce plan $O(n)$

list.ajouter(**n**)

}

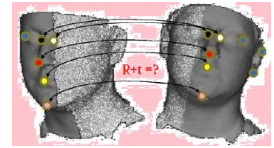
chercher les trois plus grande valeurs v_1, v_2 et v_3 de la Liste "list".

Retourner les plans associés à ces valeurs.

*** Complexité de l'algorithme:**

Cet algorithme est en temps polynomial: $O(n^2)$ avec n =taille de l'image.

--> temps de calcul (presque deux minutes)



5. PLANS STABLES AVEC CONTRAINTES:

La 3ème méthode utilisée est la détection des plans stables mais en fixant certaines contraintes parmi lesquelles on peut citer:

- **La contrainte du voisinage**: le plan stable doit contenir le point avec son voisinage (8 voisins) sinon le point n'est pas ajouté à la liste associée au plan tangent.

** Idée sur l'algorithme de la détection des plans tangents stables** avec contrainte du voisinage

Initialisation:

entier seuil =s;

Liste list = vide;

Liste counter=vide;

Algorithme:

Pour tout **p** plan tangent de l'image faire **O(n)**

{

Pour tout point **v** de l'image **faire** **O(n)**

Si les points du voisinage du point **v** appartiennent au plan **p** **alors**

{

list.ajouter(v)

}

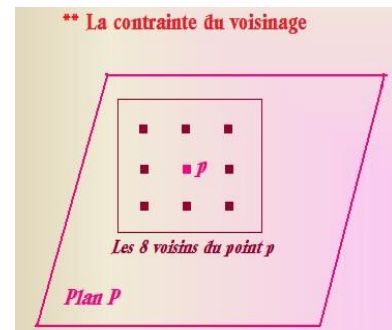
counter.ajouter(list.size());

}

chercher les trois plus grandes valeurs v1, v2 et v3 de la Liste "counter".

Retourner les plans associés à ces valeurs.

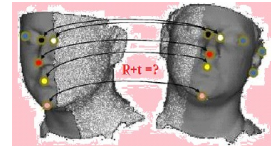
NB: list.size() --> renvoie le nombre des points appartenant au plan p avec la contrainte du voisinage.



* Complexité de l'algorithme:

Cet algorithme est en temps polynomial: **O(n^2)** avec **n**=taille de l'image.

- **La contrainte des points particuliers**: on combinera deux méthodes (détection des points particuliers et des plans stables). En fait, on ne détecte que les plans stables qui contiennent des points particuliers.



**** Idée sur l'algorithme de la détection des plans tangents stables**** avec contrainte des points particuliers

Initialisation:

entier seuil =s;

Liste plans = vide;

Liste listePoints = vide;

Liste counter=vide;

Algorithme:

```
Pour tout p plan tangent de l'image faire    O(n)
{
    Pour tout point v point particulier de l'image faire    O(n)
    Si v est un point particulier appartient au plan p alors
    {
        plan.ajouter(p) // liste des plans qui contiennent des points particulier
    }
}

Pour tout plan p' appartenant à la liste plans faire    O( plan.size() )
{
    Pour tout point v de l'image faire    O(n)
    Si v appartient au plan p' alors
        listePoints.ajouter(v)

    counter.ajouter( listePoints.size() ).
}
```

chercher les trois plus grandes valeurs v1, v2 et v3 de la Liste "counter".

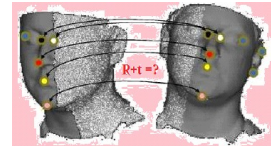
Retourner les plans associés à ces valeurs.

NB: `listePoints.size()` --> renvoie le nombre des points appartenant au plan p' avec la contrainte des points particuliers.

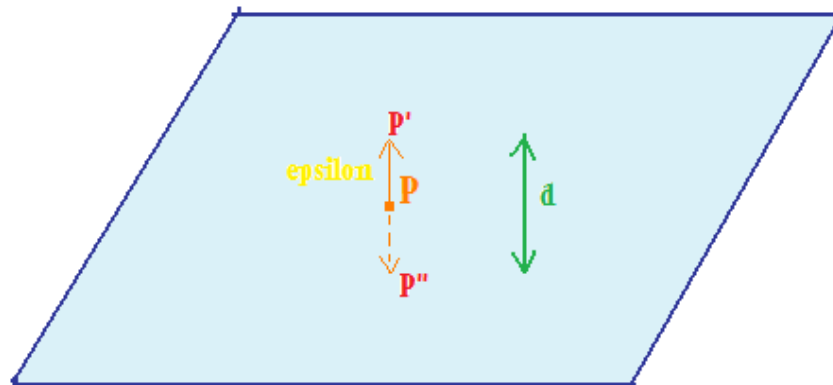
*** Complexité de l'algorithme:**

Cet algorithme est en temps polynomial: $2 * O(n^2)$ avec n =taille de l'image.

--> temps de calcul assez lent.



- La contrainte du seuil d'appartenance: on a défini un seuil **epsilon** qui permet d'étendre le choix des points qu'on associera aux différents plans tangents. En effet, pour chaque plan on choisit les points qui vérifient l'inégalité suivante:



d: la marge de variation dans l'espace

$d = 2 * \text{epsilon}$

$\text{epsilon} = \max(\text{abs}(A), \text{abs}(B), \text{abs}(C))$

- **$P(x_p, y_p, z_p)$ "appartient" à un plan tangent ssi:**

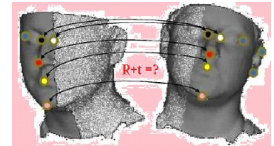
$-\text{epsilon} < A x_p + B y_p + C z_p + D < \text{epsilon}$

**** Idée sur l'algorithme de la détection des plans tangents stables*
avec la contrainte du seuil d'appartenance**

C'est le même algorithme que celui de la détermination des plans stables sauf que la fonction du test d'appartenance est redéfinie avec un seuil **epsilon** calculé pour chaque plan tangent.

*** Complexité de l'algorithme:**

Cet algorithme est en temps polynomial avec un temps d'exécution assez lent.



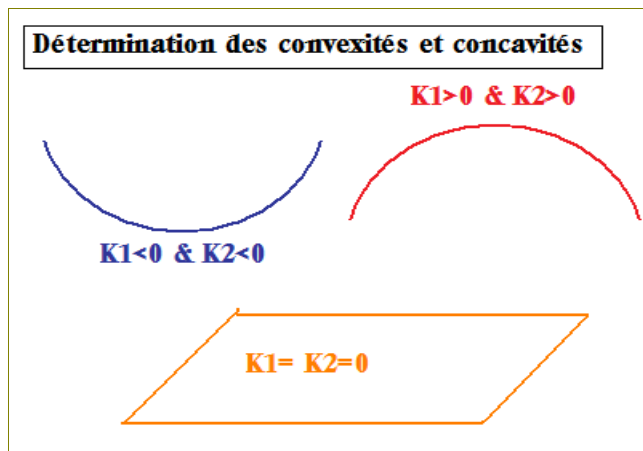
1.

6. Détermination des courbures (convexités & concavités):

Pour pouvoir déterminer les zones convexes, concaves et planes du visage, on utilise les outils de bases (κ_1 et κ_2 : les rayons de courbures).

Par suite, on fait un petit algorithme pour colorier les différents points de l'image selon le signe de $\kappa_1 * \kappa_2$.

La figure suivante expose les 3 cas à tester pour chaque point.

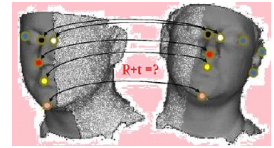


** Idée sur l'algorithme du coloriage des courbures **

Algorithme:

```
Pour tout point p appartenant à l'image faire
{
  Si p.getKappa(1)>0 & p.getKappa(2) >0 alors
    colorier p en rouge   (convexe)
  Sinon si p.getKappa(1)<0 & p.getKappa(2) <0 alors
    colorier p en vert   (concave)
  Sinon si p.getKappa(1)==0 & p.getKappa(2) ==0 alors
    colorier p en bleu   (plan)
  Sinon si p est un point qui a des voisins avec des z=0 càd appartenant à un
  contour
    alors colorier p en jaune
  Sinon colorier p en cyan
```

* Complexité de l'algorithme: Cet algorithme est en temps linéaire: $O(n)$



CHAPITRE 3: IDENTIFICATION DE LA TRANSFORMATION ENTRE DEUX POSITIONS DIFFERENTES

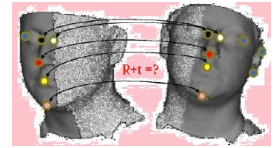
Une fois les particularités de l'image détectées, il faut pouvoir calculer la transformation entre ces deux images, autrement dit la transformation entre les particularités de chacune des images. La détection des particularités de l'image doit être la plus fiable possible afin d'estimer la position du visage le plus fidèlement possible.

1. Etude préliminaire:

Etant donné deux images, nous voulons estimer la position d'un visage par rapport à l'autre. Pour ce faire, nous allons raisonner sur des points intéressants parmi les points particuliers, c'est-à-dire des points dont nous avons la certitude qu'ils seront présents sur les deux images (nez, yeux, ...). Notons que si la détection s'effectue à l'aide de la méthode des plans stables, le raisonnement est le même que pour les points particuliers. Afin de détecter la transformation de la manière la plus optimale, nous avons choisi de faire une étude sur 3 points (soit 2 vecteurs).

Nous avons décidé de séparer la transformation étudiée en une translation et une rotation. Dans un premier temps, nous calculons la translation entre deux points. Puis la rotation est étudiée à l'aide de deux vecteurs.

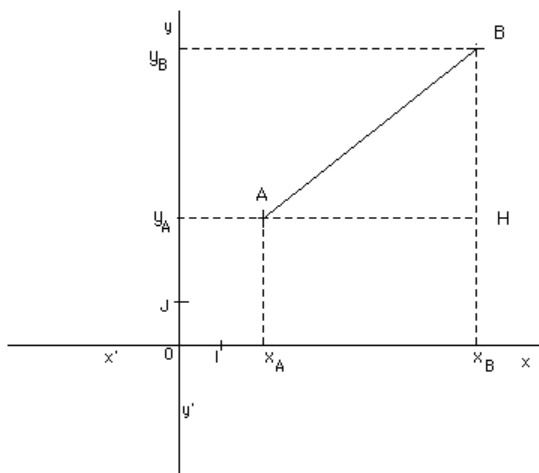




2. La translation:

2.1. Calcul:

La translation est calculée à partir de deux points pris dans chaque image. Après superposition des deux images, les coordonnées du deuxième point sont soustraites aux coordonnées du premier.



Par exemple, sur la figure ci-contre, la translation entre le point A et le point B sera calculée comme suit :

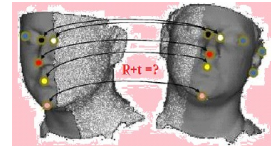
$$\left\{ \begin{array}{l} X = x_B - x_A \\ Y = y_B - y_A \\ Z = z_B - z_A \end{array} \right.$$

2.2. Application de la translation

Nous avons ainsi les coordonnées (X, Y, Z) d'un vecteur de translation que nous pourrions appliquer à n'importe quel point de l'image pour obtenir un nouveau point translaté.

Par exemple, soit un point de coordonnées (x, y, z), les coordonnées du translaté de ce point seront :

$$(x + X, y + Y, z + Z)$$



3. La rotation:

Pour le calcul de la rotation, nous raisonnerons en terme de points mais remarquons une fois de plus que le raisonnement est identique dans le cas d'une détection des particularités de l'image grâce à la méthode des plans stables.

Pour caractériser une rotation dans un espace à 3 dimensions, nous avons besoin de définir un axe et un angle de rotation.

3.1. Calcul:

Prenons trois points dans chaque image : x_1 , x_2 et x_3 dans l'image de départ et y_1 , y_2 , y_3 dans l'image résultat. Tout d'abord, un changement de référentiel est opéré afin que le point x_1 (respectivement y_1) soit la nouvelle origine du repère de l'image de départ (respectivement de l'image résultat). On construit ainsi deux vecteurs ayant une origine commune dans chaque image : v_1 et v_2 pour l'image de départ et w_1 et w_2 pour l'image résultat.

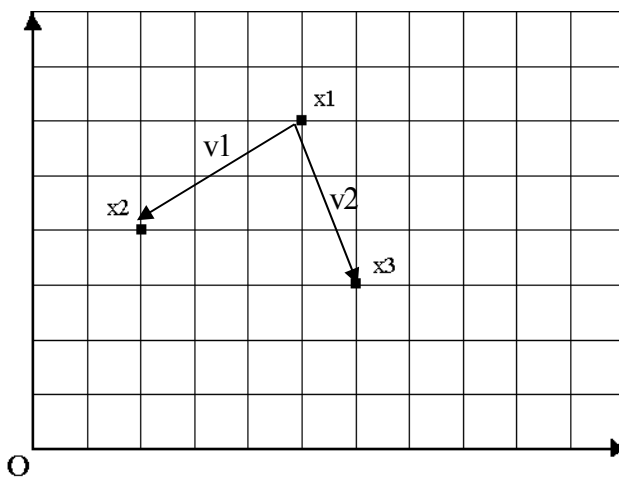


Image de départ

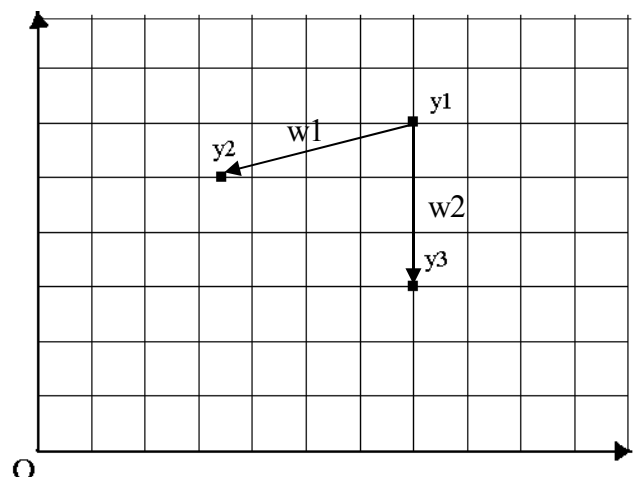
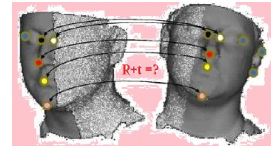


Image résultat

Nous superposons ensuite les deux images, et appliquons la translation calculée plus haut à la nouvelle origine de l'image résultat afin de superposer x_1 et y_1 . Remarquons qu'il est nécessaire, pour ne pas fausser les calculs de rotation, d'effectuer la translation avant la rotation afin que les vecteurs générés aient bien la même origine lors de la superposition des deux images.

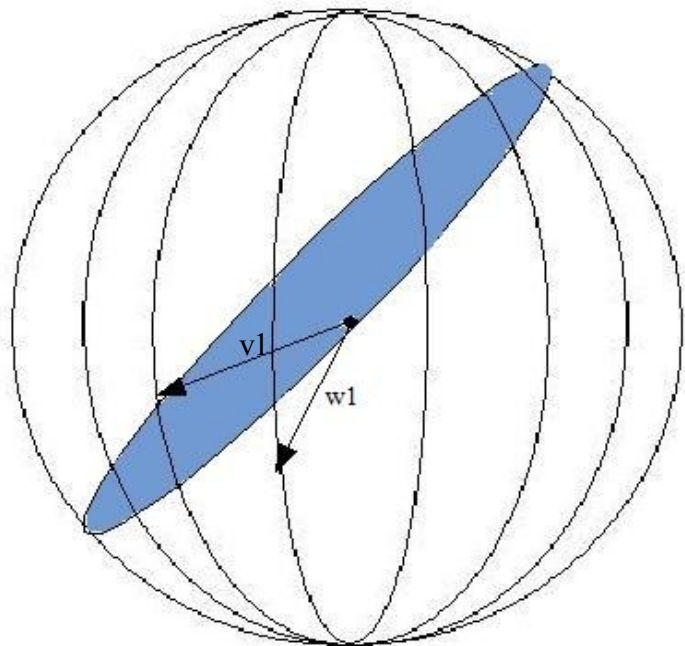


Il convient donc à présent de faire se correspondre les vecteurs de chaque image deux à deux, par exemple le vecteur correspondant à l'oeil droit dans l'image de départ avec le vecteur correspondant à l'oeil droit dans l'image résultat ; et c'est la rotation entre chaque paire de vecteurs que nous voulons calculée.

Intéressons nous dans un premier temps à l'axe de rotation. Pour trouver cet axe, nous générerons deux plans.

Soit v_1 et w_1 une paire de vecteurs correspondants. Considérons la sphère qui a pour centre le point d'application commun de ces deux vecteurs et pour rayon la norme de ces vecteurs (supposée égale).

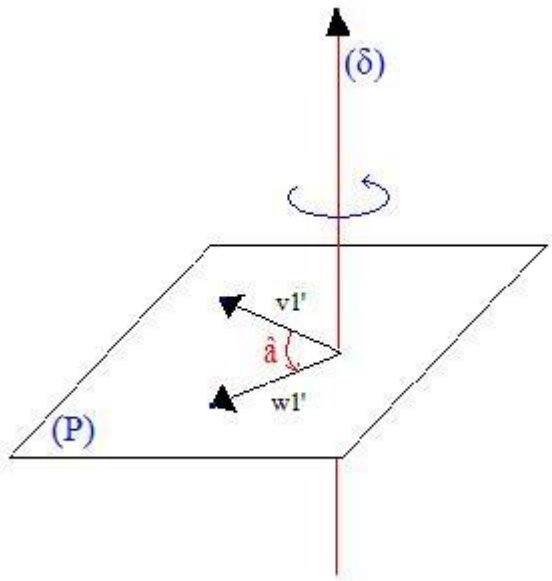
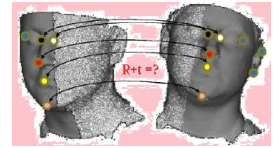
Nous générons ainsi un premier plan (en bleu sur la figure ci-contre) passant par le centre de la sphère et la médiatrice de l'angle entre les deux vecteurs. Nous pouvons alors affirmer que l'axe de rotation sera dans ce plan.



Cette opération est effectuée pour la deuxième paire de vecteurs correspondants v_2 et w_2 . L'axe de rotation est compris dans ces deux plans ; il correspond donc à l'intersection des deux plans.

Maintenant que nous avons trouvé l'axe autour duquel la rotation est effectuée, cherchons l'angle de rotation. Celui-ci est le même pour chaque paire, nous raisonnerons donc sur une seule paire.

Nous projetons tout d'abord les deux vecteurs sur le plan orthogonal à l'axe. L'angle de rotation correspond alors à l'angle présent entre chaque paire de vecteurs ainsi projetés.



Sur la figure ci-contre, le plan (P) est le plan orthogonal à l'axe de rotation (δ).

$v1'$ et $w1'$ sont les projetés orthogonaux respectivement des vecteurs $v1$ et $w1$.

L'angle $\hat{a}$ est l'angle de rotation.

3.2. Application de la rotation

Nous avons ainsi caractérisé la rotation en 3 dimensions par son axe et son angle. Voyons à présent comment appliquer cette rotation à un point quelconque.

Chaque point de l'image est caractérisé, entre autre, par ses coordonnées (x, y, z) . Considérons alors un point comme un vecteur dont le point d'application est l'origine de l'image.

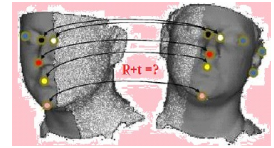
Benjamin Olinde Rodrigues (1795-1851) a énoncé une formule permettant de calculer la rotation d'un vecteur à partir de l'axe et de l'angle de rotation. Pour appliquer la rotation calculée plus haut, c'est cette formule que nous appliquerons à chaque point.

Soit un vecteur $v \in \mathbb{R}^3$. D'après la formule de Rodrigues, pour trouver le vecteur v_{rot} , image du vecteur v par la rotation d'angle $\hat{a}$ autour de l'axe δ (δ étant ici un vecteur unitaire dans $\mathbb{R}^3$), on applique la formule suivante :

$$v_{\text{rot}} = v \cdot \cos(\hat{a}) + \delta \times v \cdot \sin(\hat{a}) + \langle \delta, v \rangle \delta \cdot (1 - \cos(\hat{a}))$$

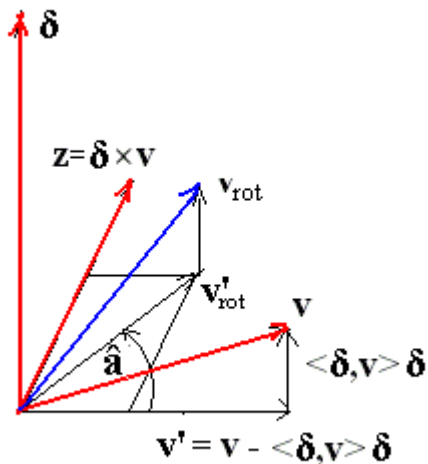
où : $\delta \times v$ est le produit vectoriel de δ par v ,

et : $\langle \delta, v \rangle$ est le produit scalaire des vecteurs δ et v .



****Démonstration de la formule :**

Sur la figure ci-dessous, le vecteur v' est le projeté du vecteur v sur le plan orthogonal à l'axe δ et z est le produit vectoriel des vecteurs δ et v .



L'application de la rotation du vecteur v' donne le vecteur v'_{rot} avec :

$$\begin{aligned} v'_{\text{rot}} &= v' \cdot \cos(\hat{\alpha}) + z \cdot \sin(\hat{\alpha}) \\ &= (v - \langle \delta, v \rangle \delta) \cdot \cos(\hat{\alpha}) + \delta \times v \cdot \sin(\hat{\alpha}). \end{aligned}$$

Remarque : Etant donné que le vecteur δ est unitaire, les vecteurs v' et z ont la même norme et elle vaut :

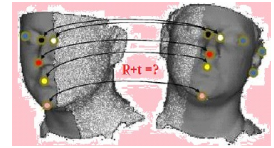
$$|v - \langle \delta, v \rangle \delta|.$$

Ainsi, l'application de la rotation au vecteur v donnera le vecteur suivant :

$$v_{\text{rot}} = (v - \langle \delta, v \rangle \delta) \cdot \cos(\hat{\alpha}) + \delta \times v \cdot \sin(\hat{\alpha}) + \langle \delta, v \rangle \delta$$

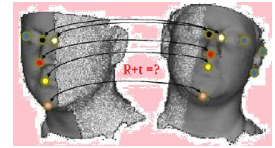
Soit :

$$v_{\text{rot}} = v \cdot \cos(\hat{\alpha}) + \delta \times v \cdot \sin(\hat{\alpha}) + \langle \delta, v \rangle \delta \cdot (1 - \cos(\hat{\alpha}))$$



II- IMPLEMENTATION EN JAVA





II- INTRODUCTON: IMPLÉMENTATION EN JAVA

 Pour commencer le projet, des packages nous ont été fournis contenant les classes fondamentales pour définir le type de données qui sera utilisé dans la suite. Parmi les package, on peut citer **Image** et **Geometry** qui seront enrichis par notre travail. Les classes qui seront essentiellement utilisées sont *ImageInt2D* et *Point2D* et *Point3D*:

- **ImageInt2D:**

ImageInt2D	
	imgorigin:Point2D
	width:int
	height:int
	data:int[]

Cette classe construit une image en 3D en utilisant un tableau d'entier 'data' unidimensionnel. En effet, l'index i de ce tableau est égal à $\text{width} * y + x$ avec x et y les coordonnées du point 3D et le contenu de la case mémoire pointée par cet index est égale à la 3ème dimension z . Les coordonnées des points 3D sont des entiers. On a aussi comme attribut

imgorigin (Point2D: origine de l'image), *width* (taille d'une ligne) et *height* (taille d'une colonne). La création de cette instance se fait à partir d'une image de format **PGM**.

- **Point2D:**

Point2D	
	x:int
	y:int

Un Point2D est un point qui possèdent comme attribut deux entiers x et y qui sont ses coordonnées dans un espace bidimensionnel.

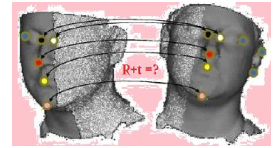
- **Point3D:**

Point3D	
	x:int
	y:int
	z:int

Un Point3D est un point qui est caractérisé par ses trois attributs en entier: x , y et z .

Après avoir établi toutes les définitions utiles pour étudier les deux masques 3D et rédigé les algorithmes de localisation des zones particulières, on implémentera ces théories en Java pour pouvoir calculer la rotation et la translation recherchée et visualiser les résultats voulus.

La programmation Java nécessite une étude préliminaire pour pouvoir définir la bonne hiérarchie des classes qui nous permettra de définir les instances et les méthodes d'accès. En effet, pour chaque étape du projet, on a essayé de trouver la hiérarchie convenable qui nous aidera à économiser de la mémoire et de



définir d'une manière claire et efficace toutes les classes nécessaires à la réalisation de ce projet.

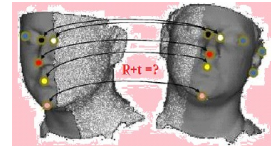
Pour cela, on a utilisé plusieurs propriétés du langage Java parmi lesquelles on peut citer:

- *Polymorphisme*
- *Héritage*
- *Généricité*

La première partie essentielle qui sera implémentée en Java est celle des *outils de base*, ensuite *la recherche des points particuliers*, puis *la définition de la rotation et la translation* en enfin *le développement d'une interface graphique* pour une exécution plus facile et rapide du projet.

On a aussi défini des interfaces et des classes pour *la génération des fichiers de formats **TXT** et **WRL*** qui seront utiles pour toutes les autres classes pour la visualisation 3D et l'accès rapide aux informations recherchées sur le masque étudié. L'intérêt de la création de cette hiérarchie sera expliqué dans *le chapitre 2* de cette partie.

Enfin, pour bien expliquer les méthodes essentielles de chaque classe, on générera l'**UML** du code Java à l'aide du programme "**Altova Umodel**".



CHAPITRE 1: CLASSES DES NOTIONS DE BASE

1. La mise en place:

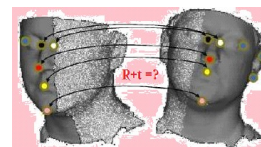
Lorsque l'on doit créer une nouvelle structure de donnée pour un programme devant s'intégrer voir se greffer à un autre il y a plusieurs contraintes à prendre en compte, surtout pour des programmes qui comme celui ci vont être amenés à faire beaucoup de calculs.

Le premier souci est l'utilisation de la mémoire et les temps d'accès, en effet bien que l'utilisation vulgarisée de la programmation objet que permet Java soit pratique et de plus en plus complète il est clair qu'une utilisation succincte s'impose, quitte parfois à renoncer à l'héritage entre classes pour éviter la duplication d'attributs, pour éviter l'explosion de la mémoire lorsque l'on traite des images ayant des millions de points. De plus il faut utiliser les structures proposées par Java qui permettent les temps d'accès les plus rapides.

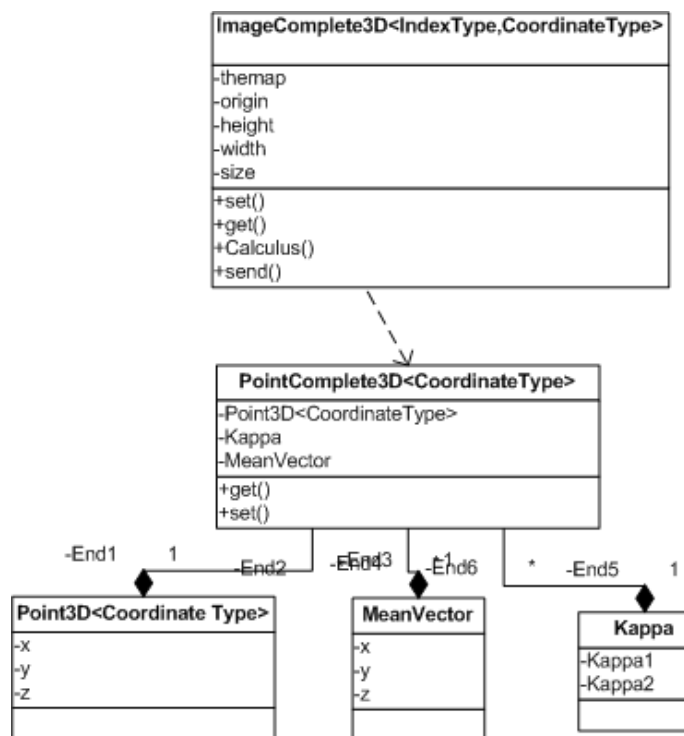
Ainsi on essaye de se baser le plus sur l'utilisation des pointeurs qui permettront de donner une structure hiérarchisée du programme sans pour autant saturer la mémoire lors des calculs. On remarque aussi que Java est très gourmand en mémoire, si il n'y a pas de maîtrise et de demande spécifique lors de l'instanciation des classes notamment les Collections, la machine Java a tendance à attribuer automatiquement des zones mémoire en prévision de leur utilisation ultérieure lorsque par exemple on remplit un certain pourcentage de la capacité de la Collection. On peut alors avoir en proportion une réservation mémoire de la taille de ce pourcentage.

Par ailleurs il faut trouver un compromis entre les constructeurs de base qui allouent par défaut une certaine capacité, mais qui sera inutilisée dans certains cas lorsque l'on ne connaît pas à l'avance la taille du vecteur et une taille définie mais qui pourrait être vite dépassée.

La seconde contrainte est le souci de généricité, en effet ce que permet la version 6.0 du JSE de Sun, c'est d'utiliser la généricité à son potentiel maximal, qui est aussi l'une des conditions pour l'utilisation des Design Patterns.



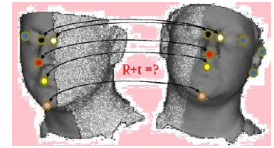
Le précédent programme n'ayant pas été fait pour cela, il a fallu donc repenser la structure de donnée dans sa totalité, on peut bien sur se demander si cela est vraiment nécessaire. Mais tout programmeur sait que la généricité est un avantage indéniable, et l'exemple présent est typique, les coordonnées en 3D des points, c'est à dire l'essentiel des données à traiter, peuvent être entières, flottantes, doubles ou encore byte, et l'accès à une donnée de la Collection représentant l'image doit pouvoir être fait selon les besoin du client, avec des coordonnées séparées, mais aussi en passant d'autres instances d'objet décrivant un aspect de la Collection, comme des *Point3D* ou des *PointComplete3D*.



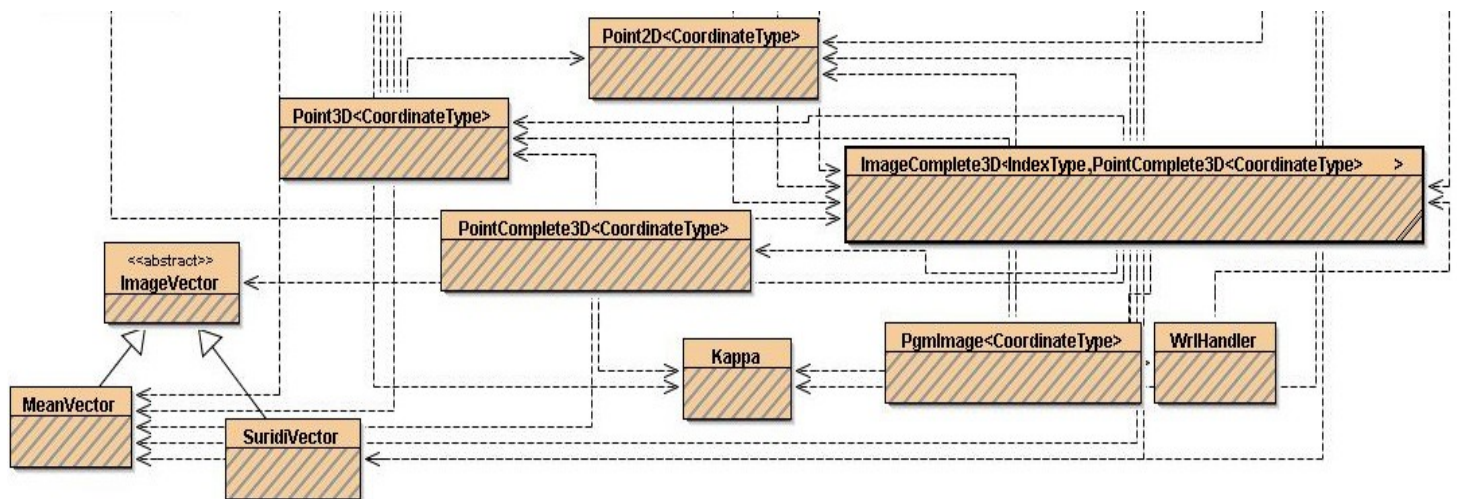
Cette nécessité fut confirmée lorsque la précision données de types entières se trouva être insuffisant devant le nombre de points. La généricité est donc nécessaire non seulement pour clarifier le programme et le rendre utilisable, mais pour qu'il puisse être utilisé aussi facilement par d'autre classes qui elles même sont génériques.

Une autre contrainte est la clarté, en effet on ne peut pas tout mettre dans la même classe sous prétexte que le programme est à vocation pédagogique. Le code doit être facilement réutilisable et c'est dans ce souci qu'il a été créé. On trouvera donc des classes pour la représentation de points, de vecteurs en 3D, mais aussi pour la génération de code ou de texte spécifique aux besoins du programme ou de la visualisation. On doit pouvoir aussi à tout moment accéder de manière rapide à tout attribut ou sous attribut d'une classe, ce genre d'implémentation est parfois répétitif et peut prendre du temps mais la facilité d'utilisation et de compréhension du code est multipliée.

Toujours dans ce même souci de réutilisation le code a bien sur été entièrement codé et commenté en anglais.



On arrive ainsi à la greffe de la structure suivante au programme actuel, uniquement pour la partie calcul des paramètres qui vont servir d'estimateurs pour le reste des fonctions.

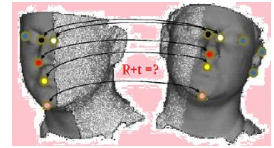


2. L'implémentation des méthodes en Java:

En tout ce sont plusieurs milliers de lignes de codes qui auront été écrites dans ce projet, on comprend facilement alors les précautions à prendre et les efforts d'efficacité qu'il faut faire sous peine de se retrouver à changer plusieurs centaines de lignes à chaque débogage. Dans le code toutes les méthodes sont commentées et rangées dans quatre catégories, les méthodes set, les méthodes get pour modifier ou accéder aux attributs privés des classes, les méthodes de visualisation ou sorting, pour accéder aux données selon certains paramètres, et les méthodes de calcul qui sont moins nombreuses mais les plus essentielles.

Les tableaux de données utilisés, ont été les Vector ou les ArrayList pour créer des listes de vecteurs notamment, ou encore des listes de points. En revanche par souci de généricité ce sont des HashMap qui ont été utilisées pour représenter les tableaux de pixels. L'un des grands avantages de ce type de programmation est que les possibilités sont multipliées (généricité, indexation, tests) alors que les temps d'accès sont constants, la mémoire utilisée par rapport aux Array classiques est grosso modo la même et la capacité mémoire peut être maîtrisée grâce au loadFactor (pourcentage de remplissage à partir duquel on réserve de nouvelles zones mémoire).

Pour les fonctions de calcul, l'utilisation de la mémoire dynamiquement a primé sur les accès répétés à la mémoire. En effet il faut faire un choix entre la rapidité d'exécution et la quantité de mémoire utilisée, et c'est le premier choix



qui semble ici le plus important.

Malheureusement, les calculs du vecteur normal du point et de la courbure ne sont pas des algorithmes optimisables. Pour tout vecteur, il faut calculer la somme de tous les vecteurs normaux des points du voisinage pour avoir une valeur assez robuste de la normale au plan global du voisinage. Quand au calcul de la courbure ce sont une série d'opérations arithmétiques, qu'il serait bien d'implémenter en assembleur, mais cela demande une utilisation in-depth de la programmation java.

C'est en testant ce genre d'opération qu'on se rend compte de la faiblesse du moteur java au niveau de la puissance de calcul, finalement, c'est bien un langage de haut niveau dont le moteur ne permet pas de faire des calculs puissants, mais qui est très pratique pour la gestion de structures de données complexes.

Un certain nombre de méthodes et de classes ont aussi été implémentées pour pouvoir générer des fichiers de différents types pour la visualisation. Les extensions utilisées ont été pgm, wrf et txt pour ranger les images, visualiser les données en 3D et ranger les résultats des calculs respectivement.

3. Vers une utilisation des patterns et de la généricité:

On voit facilement que cette partie du programme a été implémentée dans une optique de sérialisation et d'intégration de design patterns. Mais surtout, cela sera à l'avenir nécessaire pour toute future reprise de cette implémentation.

Finalement les lignes de code sont trop nombreuses et le débogage trop lent pour être efficace. Il faudrait par exemple utiliser des pattern Visitor, Factory ou Interpreter pour la visualisation en fichier. Un pattern Flyweight pour partager les données des pixels de l'image lorsque l'on instancie une ImageComplete3D.

D'une certaine manière on a déjà utilisé un pattern Composite mais cela est pour le moment insuffisant.

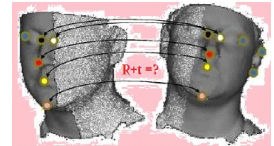
Il est assez difficile d'implémenter la généricité sans utiliser un grand nombre d'objets qui alourdisse le calcul et la taille mémoire.

La réponse adaptée au problème du type de coordonnée semble être <T extends Number> où Number est la classe mère du SE de Java pour tous les objets représentant un nombre : Double, Integer, Float, Byte, LongByte...

Le problème majeur est alors l'instanciation, et l'initialisation de ces nombres. On avait auparavant des constructeurs de type primitif :

```
public Point3D () { this.x = 0 ; this.y = 0 ; this.z=0 ; }
```

Seulement, les coordonnées sont ici de type T qui étend nombre et il est



impossible d'écrire

`x = new Number()` ; ou bien `x = new T()` ;

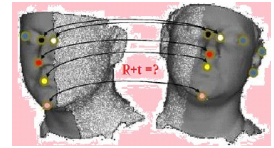
Il y a alors deux solutions, la première est d'utiliser les attributs de classe, en effet toute classe possède en réalité un objet attribut fantôme qui est de type 'class' qui contient tous les pointeurs vers les méthodes et les attributs. On peut donc détecter de quel type de nombre il s'agit et agir en adéquation. Mais cela demande de dupliquer du code.

La seconde méthode consiste à tout caster en double pour plus de précision, puis à recaster avec le type T les données. De cette manière on peut utiliser un type primitif pour initialiser les données, mais dans le cas où on choisit un type de coordonnée entier ou flottant, sur de grosses images la charge mémoire supplémentaire des double peut être un handicap.

L'utilisation des patterns peut se faire dès lors qu'un même attribut complexe est traité de plusieurs manières différentes mais produisent un objet de type commun. Par exemple, nous avons plusieurs méthodes qui prennent les valeurs de 'themap' qui sont les valeurs des pixels de l'image et leur index, pour transformer ces données en fichier.

Gagner en clarté revient alors à créer une interface Output qui sera implémentée par tous les transformateurs, et on aura plus qu'à écrire la simple méthode `accept(Output output)` dans les classes possédant l'attribut visé. Les différentes classes sont donc ici `WrlOutput`, `TxtOutput`, `PgmOutput` etc... et les méthodes sont `mapVisitor`, `kappaVisitor`, `vectorVisitor` etc...

L'importance de la généricité évoquée en début de projet n'a malheureusement été prise en compte qu'au dernier moment, mais cela est en réalité essentiel et il faut insister sur ce point. Car les méthodes de calcul discrets évoquées dans le projet ne peuvent être efficaces que sur des échantillons entiers très précis ou bien des échantillons float ou double tout simplement parce que autrement beaucoup de valeurs du rayon de courbure sont infinies.

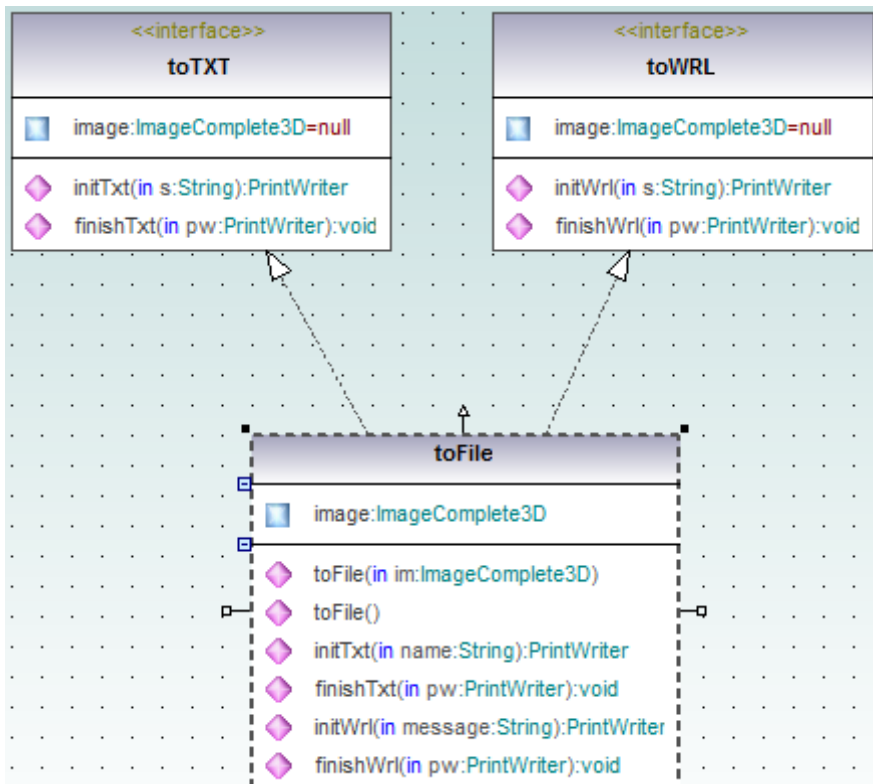


CHAPITRE 2: GÉNÉRATION DES FICHIERS DE FORMAT **TXT** ET **WRL**

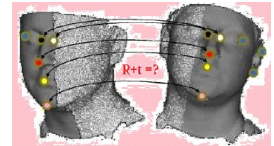
Dans 9 classes, on fait appel à des méthodes qui génèrent des fichiers de format .txt et .wrl parmi lesquelles on peut citer: ImageComplete3D, ParticularPointsList, StablePlanes, Rotation, Translation... Et toutes ces méthodes commencent et finissent par les même ligne de code (création d'un fichier, ouvrir un tube pour envoyer un flux (buffer) à écrire dans le fichier et enfin on ferme le fichier. Pour générer les fichier .wrl, on répète à chaque fois une quarentaine de lignes, il y a uniquement que quelques lignes qui changent (à peu près 5) qui sont utilisé pour le coloriage des points de l'image. Donc au lieu de copier-coller chaque fois ces lignes dans chaque classe, on a crée une classe **toFile** qui sert à faire l'initialisation et la terminaison des fichiers qui seront créés.

Par suite, on économisera des lignes de code et on implémentera un structure plus claire et plus efficace.

- **Interface toTXT**: cette interface définit les méthodes qu'on doit définir dans les classes qui l'implémentent: `initTxt(String)` et `finishTxt(PrintWriter)`.
- **Interface toWRL**: cette interface définit les méthodes qu'on doit définir dans les classes qui l'implémentent: `initWrl(String)` et `finishWrl(PrintWriter)`.



* **Classe toFile**: elle implémente les deux interfaces `toWRL` et `toTXT`. En effet, en créant l'objet `toFile`, on peut initialiser un fichier de format txt (ou wrl) et le finaliser en y écrivant le message voulu.



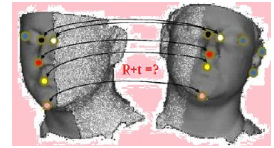
Les méthodes utilisés dans les classes qui demandent la création d'un fichier .txt ou .wrl ont le corp suivant:

Une méthode toTxt dans une classe quelconque

- 1 `toFile out=new toFile();`
`PrintWriter pw = out.initTxt(name);`] Initialisation
- 2 `pw.println("MESSAGE A RENTRER DANS LE FICHIER.");`] Le message à rentrer dans le fichier
- 3 `out.finishTxt(pw);`] Fermer le flux d'accès au fichier

Une méthode toWRL dans une classe quelconque

- 1 `toFile out=new toFile(image);`
`PrintWriter pw = out.initWrl(name);`] Initialisation
- 2 `pw.println(" 0.3 0.4 1,");`] La couleur d'un point 3D
- 3 `out.finishWrl(pw);`] Fermer le flux d'accès au fichier



CHAPITRE 3: RECHERCHE DE PARTICULARITES

1. EGI:

L'une des méthodes permettant la reconnaissance d'un visage est de travailler sur ses particularités, comme par exemple chercher à différencier entre les points qui appartiennent à des parties planes du visage comme les joues et le front, les points qui appartiennent à des parties cylindriques comme le nez et les points qui appartiennent à des parties convexes comme les yeux.

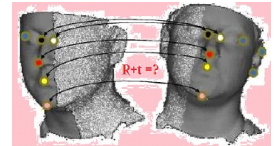
L'idée est de colorier ces points par trois couleurs différentes, les points qui appartiennent à des plans sont coloriés en rouge, les points qui existent dans des parties cylindriques sont coloriés en vert et les points qui appartiennent à des parties cylindriques sont coloriés en bleu et le reste des points qui ne correspondent pas à ces cas seront coloriés en noir.

Une fois cette étape est accomplie, on cherche la rotation entre trois points de la première image et la deuxième image modifiée par rapport à la deuxième par cette rotation, les points étudiés appartiennent respectivement à un plan, un cylindre et à une partie convexe du visage.

L'implémentation en java de cette idée consiste à définir une classe **GaussianImage** dans le package **image** qui a pour attribut une image de type **ImageCompelte3D** et qui définit une méthode faisant la différence entre les particularités définies ci-dessus. Pour le coloriage, c'est la méthodes **toWrlDestribution()** qui s'en occupe.

Et enfin pour tester, on définit une image de type **GaussianImage** dans le main de la classe **TEST** de notre projet et on appelle la méthode **toWrlDestribution()** qui va créer une image **EGI.wrl** coloriée selon le contraintes demandées.

La première idée consistait à créer les classes Cylinder, Convex, Plan et Circle dans le package geometry où la classe Cyinder crée un cylinder, la classe Convex crée un objet convexe et la classe plan crée un plan sous forme de vecteur contenant des points de type PointComplete3D. Et enfin la classe Circle est la classe qui définit un cercle comme étant un plan sur lequel tous les vecteurs normaux d'un point PointComplete3D seront projetés. Cette classe est supprimée dans un deuxième temps pour la remplacer par la méthode **getProjection()** dans la classe GaussianImage qui calcule les différentes projections.



❖ Implémentation en java :

Les packages modifiés sont seulement le package geometry et le package image.

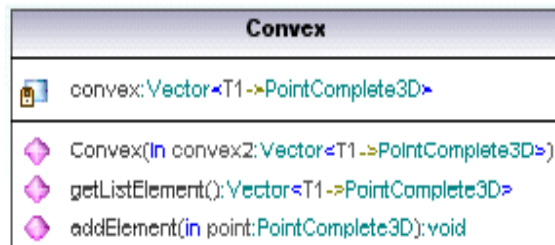
Package geometry

-Classe Convex-

Deux principales méthodes:

*getListElement (qui retourne les points dans des parties convexes du visage)

*addElement: ajoute un élément dans l'ensemble

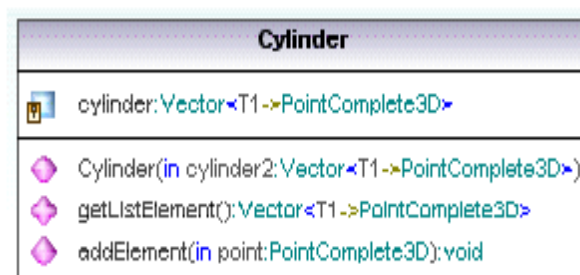


-Classe Cylinder-

Deux principales méthodes:

*getListElement (qui retourne les points dans des parties convexes du visage)

*addElement: ajoute un élément dans l'ensemble

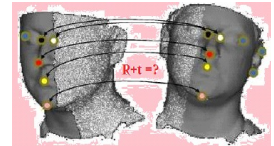


-Classe Plan-

Deux principales méthodes:

*getListElement (qui retourne les points dans des parties convexes du visage)

*addElement: ajoute un élément dans l'ensemble



Plan	
	plan: Vector<T1->PointComplete3D>
	Plan(in plan2: Vector<T1->PointComplete3D>)
	getListElement(): Vector<T1->PointComplete3D>
	addElement(in point: PointComplete3D): void

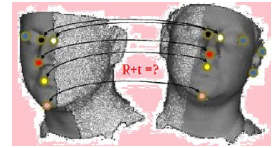
Ces trois classes ont pour rôle d'identifier la nature de chaque point (s'il appartient à un plan, un cylindre ou à une forme concave ou convexe du visage humain).

Package image

Dans ce package, on a conçu la classe suivante : GaussianImage

GaussianImage	
	image: ImageComplete3D
	GaussianImage(in image: ImageComplete3D)
	getHashMap(): HashMap<T1->Integer, T2->PointComplete3D>
	getNeighbors(in p: PointComplete3D): Vector<T1->PointComplete3D>
	getNeighbors_Normal(in p: PointComplete3D): Vector<T1->MeanVector>
	getVectorNormal_Max(in p: PointComplete3D): MeanVector[*]
	getVectorNormal(in p: PointComplete3D): ImageVector
	getProjection(in c: MeanVector, in p: PointComplete3D): ImageVector
	getProjections(in p: PointComplete3D): Vector<T1->ImageVector>
	getMaxProj(in p: PointComplete3D): double
	getType(in p: PointComplete3D): int
	getTypeDestibution(): int[*]
	toVnlDistributions(in name: String): void

Dans cette classe, on prend comme attribut une image de type ImageCompelte3D (qui l'image d'entrée). Les méthodes définies dans cette classe sont:



1-`getHashMap()`: elle renvoie dans une `hashMap` les points de l'image d'entrée.

2-`getNeighbors(PointComplete3D p)`: elle renvoie un vecteur contenant les voisins du point `p` en entrée.

3-`getNeighbors_Normal(PointComplete3D p)`: elle renvoie un vecteur des vecteurs normaux des voisins du point `p`.

4-`getVectorNormal_Max(PointComplete3D p)`: elle retourne les vecteurs normaux qui sont les plus distants parmi les vecteurs normaux qui correspondent aux voisins du point `p`.

5- `getVectorNormal(PointComplete3D p)`: cette fonction retourne le vecteur normal du cercle perpendiculaire au plan défini par les deux vecteurs renvoyés par la méthode précédente et sur lequel on va faire la projection des vecteurs normaux des voisins du point `p`.

6-`getProjection(PointComplete3D p, MeanVector c)`: elle renvoie la projection du vecteur `c` sur le cercle défini ci-dessus.

7-`getProjections(PointComplete3D p)`: cette méthode renvoie sous forme d'un vecteur tous les projections qui correspondent aux vecteurs normaux des voisins du point `p`.

8-`getProjMax(PoinComplete3D p)`: elle renvoie la distance maximale entre les différentes projections données par `getProjections(PointComplete3D p)`.

9-`getType(PointComplete3D p)`: renvoie sous forme d'un entier la nature du point `p`.

10-`getTypeDestribution()`: renvoie sous forme d'un tableau les types des points de l'image d'entrée(attribut de cette classe)

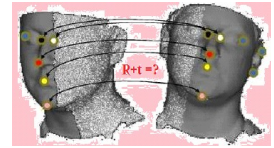
11-`toWrlDestribution()`: colorie les points de l'image selon leur type et crée l'image coloriée sous forme d'une image `wrl`.

La première idée consiste à parcourir l'image comme étant des points (`pointComplete3D`) et de vérifier à chaque fois la nature du point courant à l'aide des méthodes `isPlan(PointComplete3D p)`, `isConvex(PointComplete3D p)` et `isCylinder(PointComplete3D p)`, ces trois fonctions sont booléennes et renvoient vrai "true" si le point en paramètres est respectivement appartient à un plan, forme convexe et à une forme cylindrique.

Ensuite,on utilise trois autres fonctions en se basant sur les trois précédentes: les nouvelles fonctions sont :

1- `getPlanDistribution()` : cette fonction renvoie un vecteur contenant tous les points de l'image qui appartiennent à des plans différents.

2- `getCylinderDistribution()` :cette fonction renvoie aussi un vecteur mais contenant les points qui appartiennent à des cylindres.



3- **getConvexDistribution()**: cette fonction renvoie les points qui appartiennent à des formes convexes dans un vecteur.

A la fin on appelle ces fonctions dans la méthode **toWrlDistribution()** pour colorier les points selon leur type.

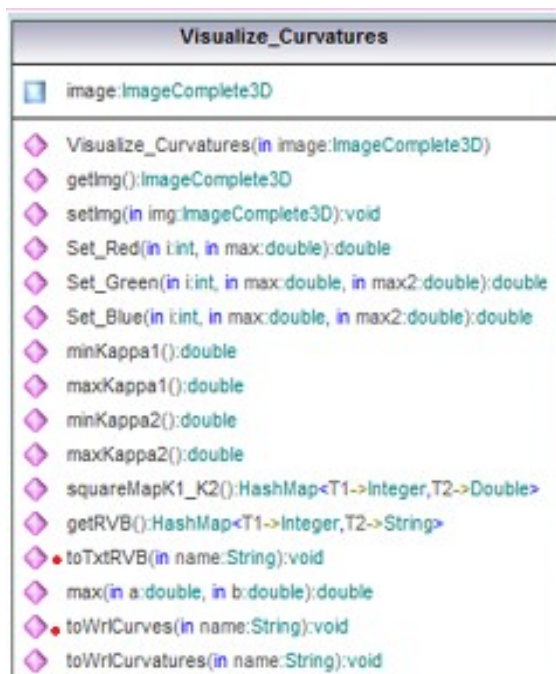
Ces méthodes sont après remplacées par les méthodes **getType(PointComplete3)** et **getTypeDestribution()**. La première donne le type du point entré en paramètres sous forme d'un entier (1--> pour un plan, 2--> pour un convexe, 3--> pour cylindre et 4--> pour autres formes).

Ensuite la deuxième fonction met dans un tableau d'entier le type correspondant à chaque point de l'image.

Et à la fin on appelle cette fonction sous forme d'un tableau dans la fonction **toWrlDistribution()** pour le coloriage.

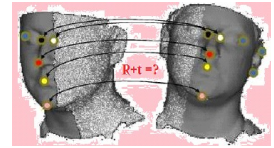
2. Classe pour la visualisation des courbures:

Pour la visualisation des courbures, on a conçu la classe suivante: **Visualize_Curvatures** avec ses attributs et ses méthodes définies.



Les méthodes marquées avec un point rouge: **toTxtRVB(String)** et **toWRLCurves(int)** génèrent respectivement un fichier de format txt qui contiennent les coordonnées des points 3D avec les couleurs **RGB** (Red Green Blue) qui lui sont associées et un fichier de format wrl qui colorie les différentes courbures de l'image 3D.

Il y a aussi les méthodes de recherche des maxima et des minima des rayons de courbures (kappa1 et kappa2): **minKappa1()**, **maxKappa1()**, **minKappa2()** et **maxKappa2()**.



• Méthodes des définitions des couleurs RGB:

Pour pouvoir colorier les courbures, on a défini chaque couleur R, G et B en utilisant les formules mathématiques dans les méthodes suivantes:

- **set_Red**
- **set_Green**
- **set_Blue**

En outre, cette classe possède comme **attribut** une ImageComplexe3D.

3. Classes des plans tangents et plans stables:

Les instances TangentPlane et StablePlanes sont créées par les classes suivantes:

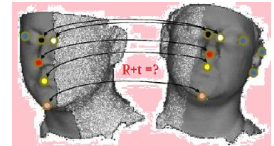
• Classe TangentPlane:

TangentPlane	
	point:PointComplete3D
	epsilon:double
	TangentPlane(in p:PointComplete3D)
	getEpsilon():double
	getPoint():PointComplete3D
	setEpsilon(in a:double):void
	setPoint(in p:PointComplete3D):void
	calculateNormal(in p1:PointComplete3D, in p2:PointComplete3D):MeanVector
	PlaneEquation():double["]
	findEpsilon():double
	belongsTo(in p:PointComplete3D):boolean
	belongsTo(in p:Point3D):boolean
	belongsToEpsilon(in p:PointComplete3D):boolean
	belongsToEpsilon(in p:Point3D):boolean
	toString():String

Cette classe possède comme **attribut**: **PointComplete3D** et un **Double** (epsilon). En effet, comme on a expliqué précédemment dans la partie théorique un plan sera défini par un point complet pour qu'on puisse accéder directement à son *vecteur normal* calculé. En outre, le epsilon est défini à part de l'équation du plan tangent et il est initialisé dans le constructeur par la méthode **findEpsilon()**.

En utilisant la fonction **belongsTo(PointComplete3D)**, on vérifie qu'un point appartienne ou non au plan au sens strict d'appartenance. Cependant, en utilisant la fonction **belongsToEpsilon(PointComplete3D)**, on vérifie que le point appartienne avec une marge d'extension spatiale égale à **epsilon**.

La fonction **toString()** est 'overrides' la classe Object. Elle renvoie l'équation du plan tangent.



- **Classe StablePlanes:**

Cette classe possède comme attribut **ImageComplete3D** car elle doit parcourir tous les points de l'image pour la recherche des plans stables.

Les méthodes essentielles de cette classe sont:

- * **MapPointsPlanes():**

Elle renvoie une *HashMap<TangentPlane, Vector<PointComplete3D>* qui associe à chaque plan tangent une liste contenant les points 3D appartenant à ce plan.

- * **NumberOfPoints():**

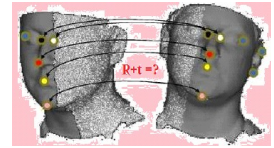
Elle utilise la méthode précédente en renvoyant une *HashMap<TangentPlane, Integer>* qui associe à chaque plan tangent un entier qui est la taille du vecteur des points 3D y appartenant.

- * **FindThreeStablePlanes():**

Cette méthode utilise aussi le résultat de la méthode précédente car elle parcourt la HashMap envoyé en cherchant les trois premiers plans qui contiennent le maximum des points du masque étudié.

StablePlanes	
	image:ImageComplete3D
	StablePlanes(in image:ImageComplete3D)
	getImg():ImageComplete3D
	setImg(in img:ImageComplete3D):void
	MapPointsPlanesTxt(in name:String):HashMap<T1->TangentPlane, T2->Vector<T1->PointComplete3D>>
	MapPointsPlanes():HashMap<T1->TangentPlane, T2->Vector<T1->PointComplete3D>>
	numberOfPoints(in name:String):HashMap<T1->TangentPlane, T2->Integer>
	findThreeStablePlanes(in name:String):TangentPlane[*]
	toWrlStablePlanes(in name:String, in nameTxt:String):String
	toTxtAllPlanes(in name:String):void
	toTxtMapPlanes(in name:String, in nameTxt:String):void

Il y a aussi les méthodes *toWrlStablePlanes(String, String)* et la méthode *toTxtAllPlanes(String)*.



4. Classes des Plans tangents stables avec contraintes:

Pour définir les plans tangents avec contraintes, on a défini deux classes:

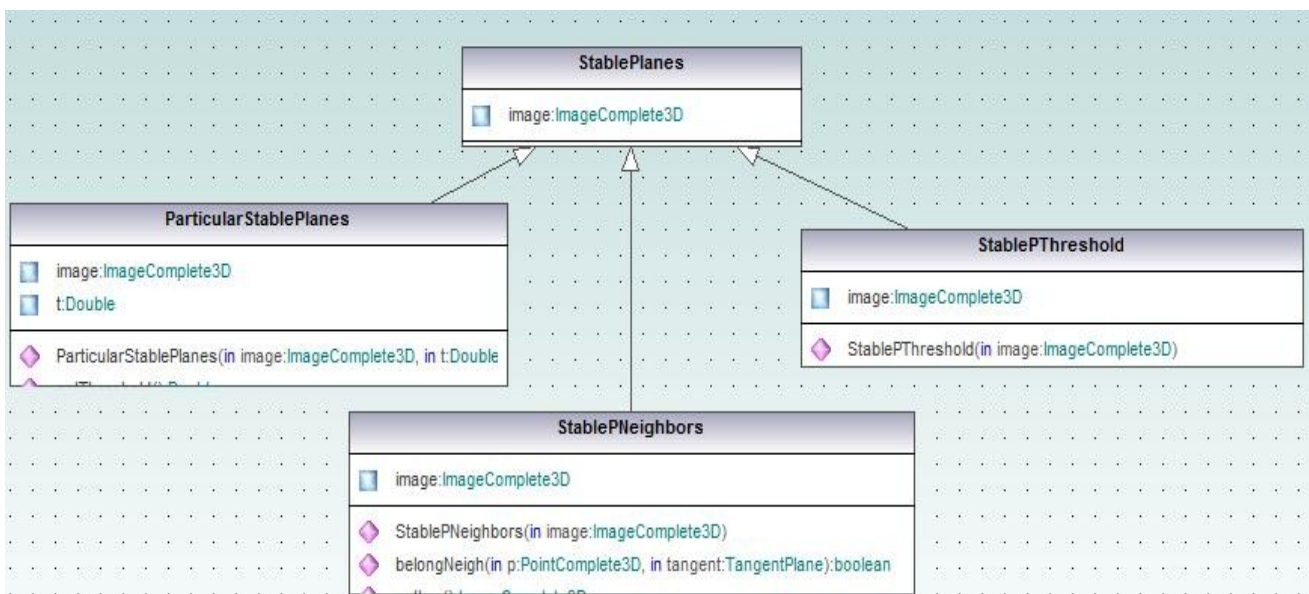
- StablePThreshold
- ParticularStablePlanes
- StablePNeighbors

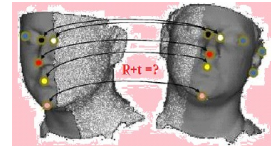
Ces trois classes **héritent** de la classe StablePlanes car il y a presque uniquement la méthode *MapPointsPlanes()* qui sera changée et la méthode *belongTo()* car se sont les contraintes d'appartenance qui change.

A titre d'exemple, dans la classe **StablePNeighbors** qui obéit à **la contrainte du voisinage**, on a défini la méthode *belongNeigh(PointComplete3D, tangentPlane)* qui vérifie si le voisinage d'un point appartient à l'image.

Pour **la contrainte du seuil d'appartenance**, en définissant la méthode *MapPointsPlanes()* on utilise la méthode de test *belongToEpsilon(PointComplete3D)*.

Enfin pour **la contrainte des points particuliers**, on a ajouté comme attribut un **double t** (qui est le seuil défini pour la recherche des yeux). Et dans la méthode *MapPointsPlanes()* on teste si le point est particulier ou non. Puis les étapes de recherches des trois plans stable particuliers sont exactement les mêmes pour la recherche des plans stables.





5. Classe pour la localisation des points particuliers:

ParticularPointsList	
	threshold:double
	nose:PointComplete3D
	image:ImageComplete3D
	ParticularPointsList()
	ParticularPointsList(in image:ImageComplete3D, in t:double)
	getThreshold():double
	getNose():PointComplete3D
	getImg():ImageComplete3D
	setThreshold(in t:double):void
	setImg(in image:ImageComplete3D):void
	shapeIndex():HashMap<T1->PointComplete3D,T2->Double>
	findEyes():Vector<T1->PointComplete3D>
	squareK1_K2():ArrayList<T1->Double>
	squareMapK1_K2():HashMap<T1->Integer,T2->Double>
	findNosePoint():PointComplete3D
	findNose():Vector<T1->PointComplete3D>
	toWriParticularPoints(in name:String):void
	toTxtParticularPoints(in name:String):void

Pour localiser le nez et les yeux selon les algorithmes énoncés dans la partie théorique, on a conçu une classe `ParticularPointsList` qui recherche deux listes (Vector) qui contient le nez et son voisinage et une autre qui contient les yeux.

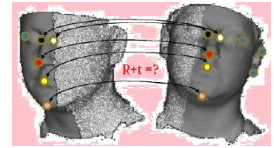
Ces listes sont renvoyées par les méthodes `findNose()` et `findEyes()`.

Pour localiser les yeux, on doit calculer le shape index pour cela on crée une `HashMap<PointComplete3D, Double>` qui associe à chaque `Point3D` la valeur de son shape index. Cette `HashMap` est renvoyé par la méthode `shapeIndex()`.

On dispose comme attribut d'un seuil '*threshold*' qui est utilisé pour

fixer une marge pour la recherche des yeux.

Cette classe génère des fichiers de format TXT et WRL.



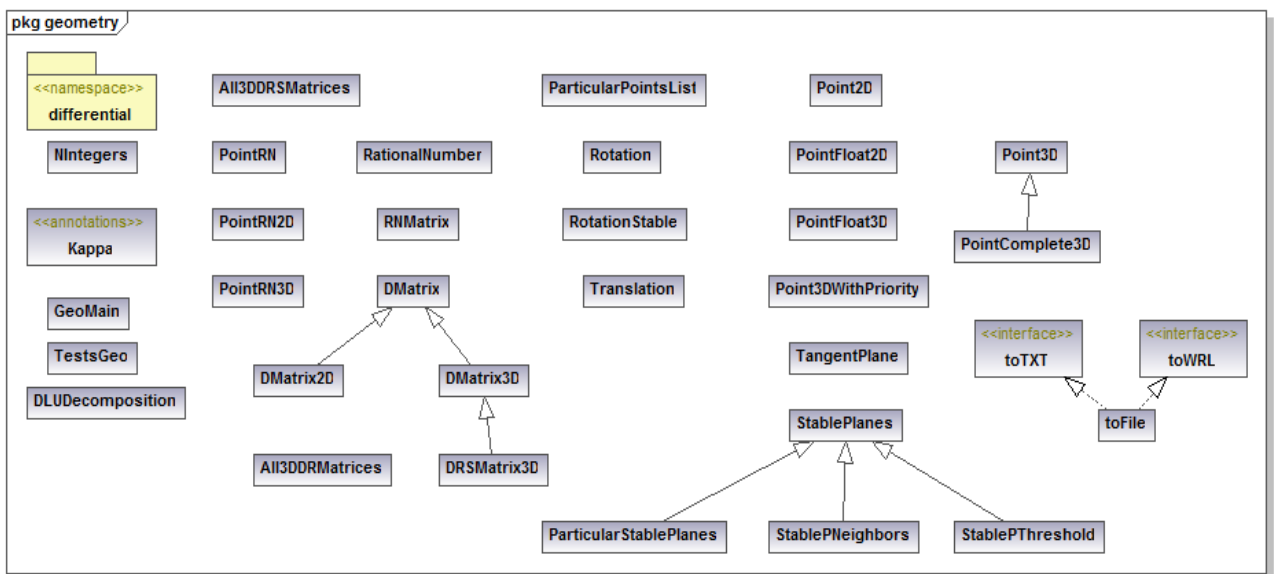
CHAPITRE 4: IDENTIFICATION DE LA TRANSFORMATION

Nous avons créé trois classes dans le package geometry pour trouver la transformation entre deux images :

- La classe Translation
- La classe Rotation
- La classe RotationStable

Ces trois classes ne font pas toutes appel à des méthodes identiques de détection des particularités de l'image.

Rappelons qu'il est nécessaire pour un bon fonctionnement que la translation soit effectuée **avant** la rotation.

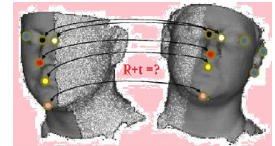


Generated by UModel

www.altova.com

1. Translation:

Dans notre implémentation, une translation est caractérisée par ses coordonnées entières (x, y, z) et par la liste de points particuliers (ParticularPointsList) issues de chaque image (l'image de départ et l'image résultat). Ainsi, pour définir une translation, nous utilisons la méthode des points particuliers afin de détecter les particularités de l'image.



Voici la description de la classe Translation avec toutes ses méthodes :

Translation	
	x:int
	y:int
	z:int
	orig:ParticularPointsList
	nvt:ParticularPointsList
	Translation(in x:int, in y:int, in z:int, in List1:ParticularPointsList, in List2:ParticularPointsList)
	Translation(in imageOrig:ImageComplete3D, in imageNvt:ImageComplete3D)
	getX():int
	getY():int
	getZ():int
	setX(in a:int):void
	setY(in b:int):void
	setZ(in c:int):void
	getCoordinates():int[?]
	transformedImage(in img:ImageComplete3D, in t:Translation):ImageComplete3D

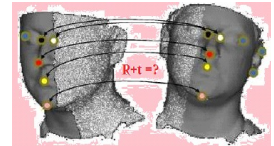
Cette classe nous permet donc de construire une translation soit en imposant des coordonnées et une liste de points particuliers, soit en calculant une translation déjà existante entre deux images (imageCompelte3D). C'est cette dernière construction qui nous intéressera le plus dans notre projet, la première construction servant surtout pour les tests.

Par ailleurs, la méthode *transformedImage* permet d'appliquer une translation à une imageCompelte3D ; elle donne en sortie l'image de départ translatée.

Nous pouvons ainsi calculer la translation entre deux images (grâce à la méthode décrite dans la section 1) puis, après avoir appliqué cette translation à l'image de départ, nous vérifions le résultat en superposant l'image obtenue et l'image résultat.

2. Rotation:

Une fois la translation calculée et appliquée à l'image de départ, nous recherchons la rotation existant entre cette image et l'image résultat. Nous avons développé pour cela deux classes.



Tout d'abord la classe Rotation est caractérisée par son axe, son angle (de type *double*) et les listes de points particuliers issues de l'image de départ et de l'image résultat. Le calcul de la rotation dans cette classe est donc basé sur la méthode de détection des points particuliers.

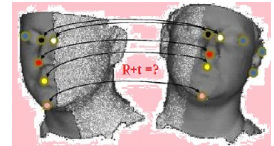
L'axe de rotation est de type *SuridiVector*, c'est donc un vecteur dont les coordonnées sont de type *double* et sur lequel nous pouvons effectuer des opérations telles que le produit scalaire ou encore le produit vectoriel.

Rotation	
	axis:SuridiVector
	angle:double
	orig:ParticularPointsList
	nvl:ParticularPointsList
	Rotation(in ax:SuridiVector, in an:double, in p1:ParticularPointsList, in p2:ParticularPointsList)
	Rotation(in imageOrig:ImageComplete3D, in imageNvl:ImageComplete3D)
	getAngle():double
	getAxis():SuridiVector
	setAngle(in a:double):void
	setAxis(in vect:SuridiVector):void
	toTxtRotation(in name:String):void
	transformedImage(in img:ImageComplete3D, in rot:Rotation):ImageComplete3D

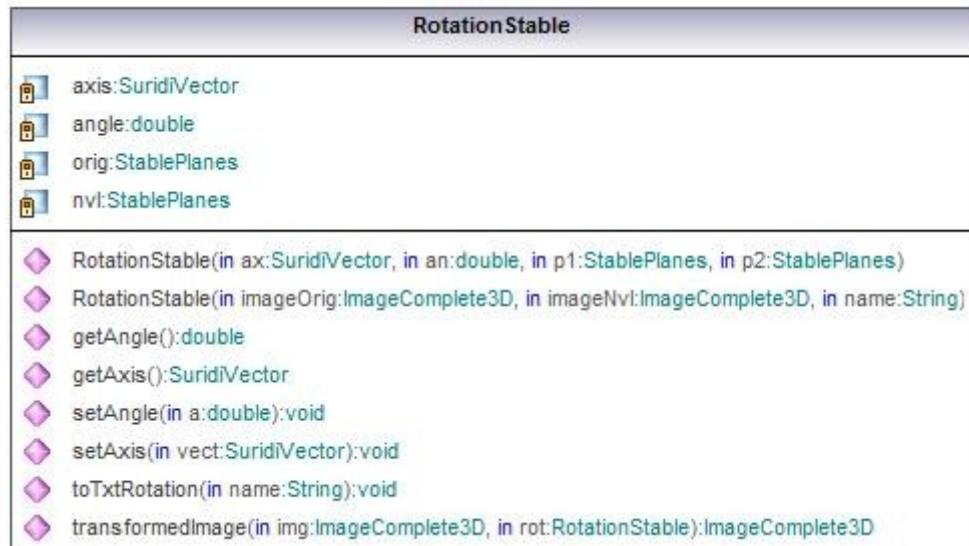
Dans notre conception, nous pouvons construire une rotation soit à partir d'un axe, d'un angle et de deux listes de points particuliers, soit en calculant la rotation existante entre deux imageCompelte3D. Ici aussi, la première construction a été utilisée surtout pour effectuer les tests et c'est la deuxième construction qui nous intéressera le plus dans ce projet.

Nous avons implémenté la *méthode transformedImage* qui permet d'appliquer une rotation à une image et qui donne en sortie l'image transformée par cette rotation. Ainsi, comme pour la translation, nous pouvons vérifier le calcul de la rotation en superposant l'image obtenue après application de la rotation calculée et l'image résultat.

Une deuxième classe, la classe RotationStable, a été créée pour le calcul de la rotation en s'appuyant cette fois sur la méthode des plans stables pour repérer des zones particulières dans l'image.

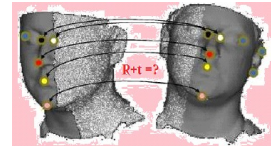


Voici l'UML de cette classe :



Cette classe est caractérisée par un axe et un angle (comme pour la classe Rotation) mais aussi par les plans stables issus de l'image de départ et de l'image d'origine (à la différence de la classe Rotation).

Les constructeurs et les méthodes sont les mêmes que pour la classe Rotation (mise à part le type utilisé pour détecter les particularités de l'image) et on utilise ici aussi une méthode *transformedImage* pour vérifier le calcul de la rotation.

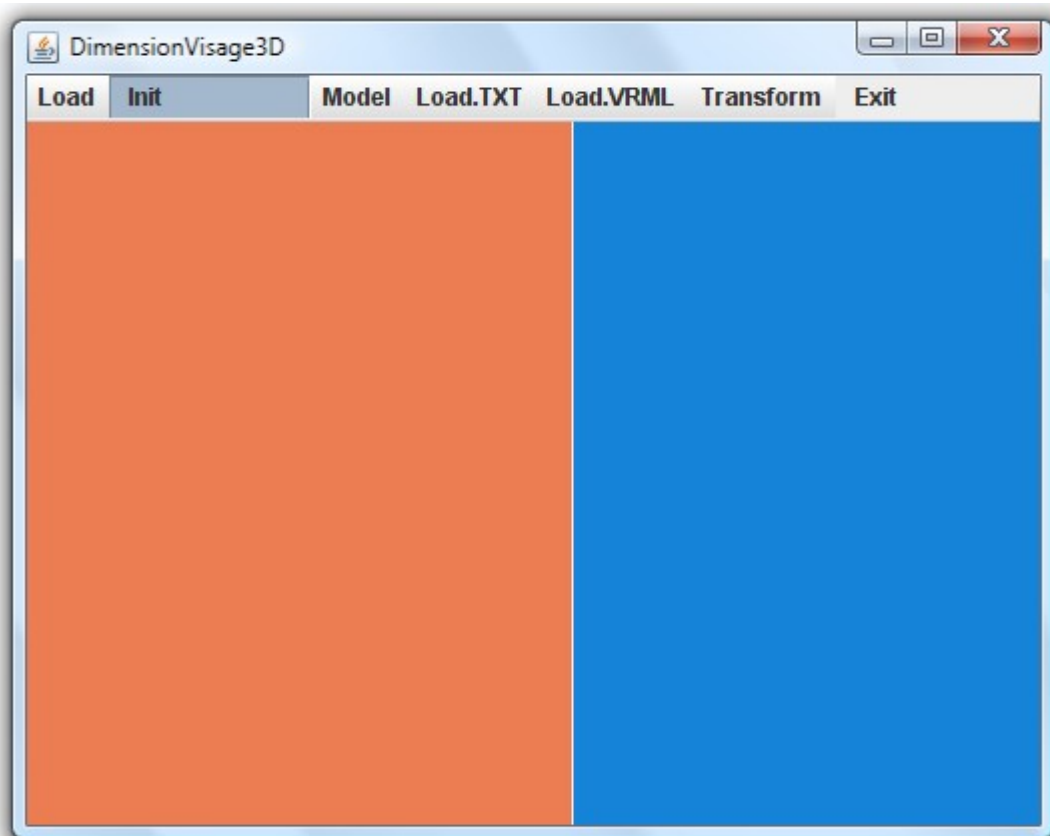


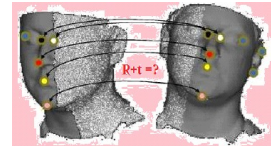
CHAPITRE 5: INTERFACE GRAPHIQUE

Le but du développement d'une interface graphique est de faciliter à l'utilisateur l'accès à toute sorte de données utiles qu'on a générées lors de notre étude de ce projet (fichier .txt, fichier .vrm). L'interface graphique sera faite à l'aide du package SWING qui est utilisé pour ce genre d'applications.

L'interface sera incluse dans un JFrame et dans un Frame on utilise beaucoup de component de Java: Panel, JMenu, JMenuItem etc.

Dans la partie suivante, on vous expliquera l'action associé à chaque bouton et on donnera une vue générale sur les classes importantes créées en Java. L'interface développée est simple qui sert à exécuter le programme et accéder d'une manière efficace et rapide aux informations obtenues.



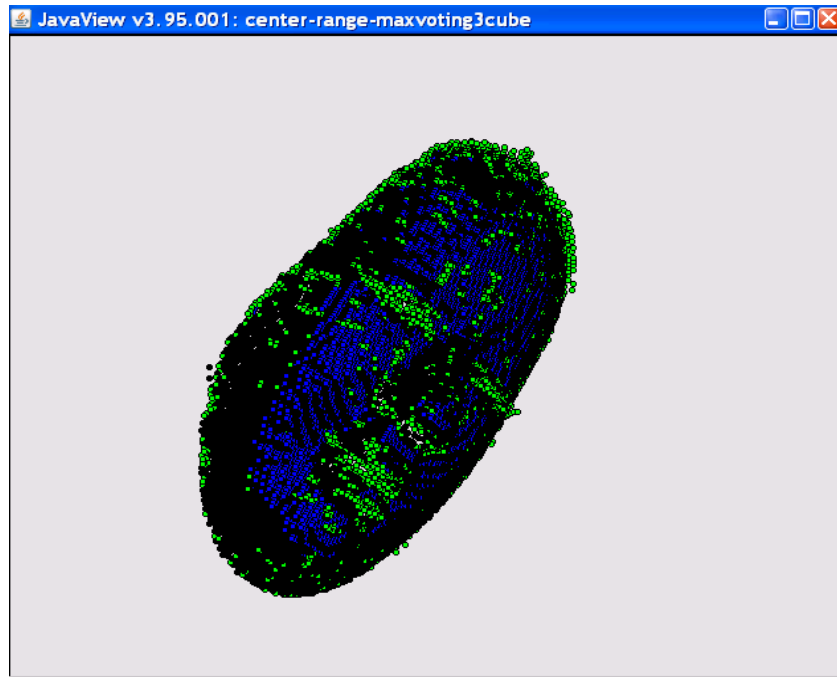


A titre d'exemple, pour concevoir la fonction qui nous permet de choisir les deux images initiales, nous avons utilisé l'instance **JFileChooser**. Pour pouvoir afficher ces deux images en format BMP, nous avons utilisé l'instance **ImageIcon**. En plus, pour pouvoir faire la saisie de données par clavier, nous avons utilisé la méthode **JOptionPane.showInputDialog**. Et en ce qui concerne l'affichage des différents fichiers générés (.wrl & .txt), on a utilisé la méthode **Runtime.getRuntime().exec()**.

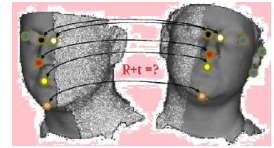
1. Utilisation de JavaView:

JAVAVIEW est un logiciel qui peut afficher les images3D créées en JAVA. Il peut afficher les images en 3D clairement. Puis, il peut emboîter les fichiers VRML dans l'applet. En utilisant la bibliothèque de JAVAVIEW, on peut réaliser beaucoup de manipulations avec les images3D en JAVA.

Au début, nous avons préféré d'utiliser JAVAVIEW pour montrer les images3D. Voici une capture d'écran qui représente le image3D avec JAVAVIEW.



Mais dans le programme codé, on utilise des fichiers de format **wrl** qu'on ouvre avec un programme intitulé "VRML VIEW". Pour utiliser la bibliothèque de JAVAVIEW, il faut faire les fichiers VRML en format de **jvx**. Donc finalement, nous avons utilisé la méthode **Runtime.getRuntime()** pour afficher les images3D.

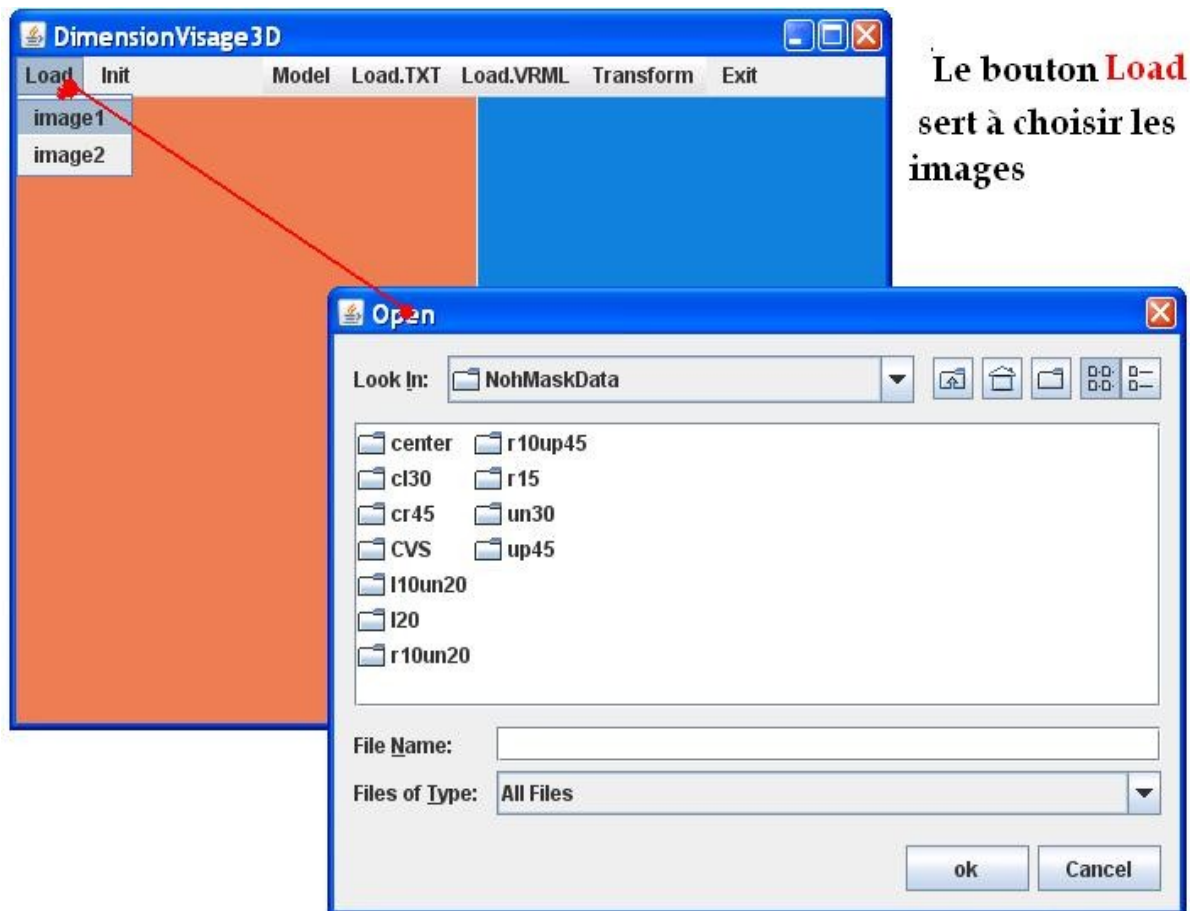


Mais l'utilisation de JAVAView était bénéfique car on a appris comment l'utiliser et on a découvert les avantages de son utilisation.

2. Les boutons de l'interface graphique:

◆ Bouton 'Load':

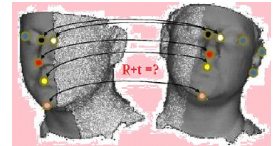
Le bouton 'Load' nous permet de choisir les deux images initiales à comparer en parcourant une arborescence de différents dossiers.



◆ Bouton 'Init':

Une fois nous choisissons les deux images, nous utilisons le bouton 'Init' pour générer toutes les instances des différentes classes nécessaires pour créer les fichiers .TXT et .WRL qu'on pourra exploiter en cliquant sur les autres boutons.

ESTIMATION DE LA POSITION D'UN VISAGE EN 3D



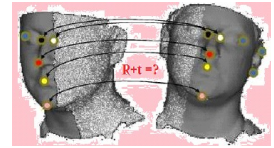
Le bouton **Init**
sert à initialiser ce
programme

♦ Bouton 'Model':

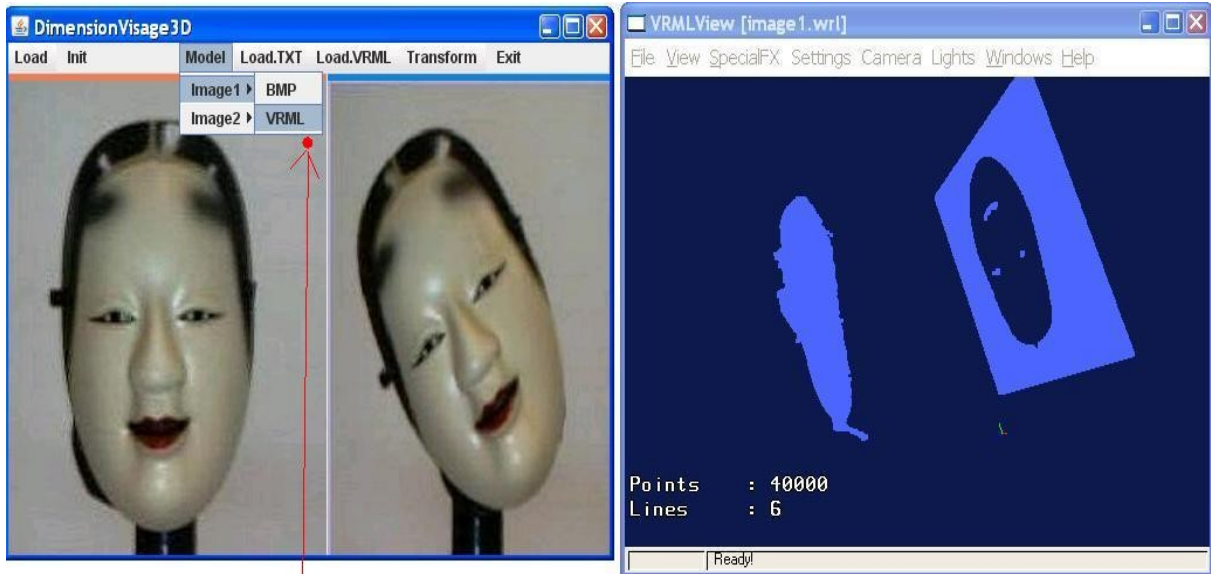
Ce bouton sert à afficher un menu et en choisissant l'option BMP, les deux images seront affichées en **BMP**.



Le bouton
**Model-image1
-BMP**
sert à ouvrir les
images qu'on
choisit dans le
format de
BMP



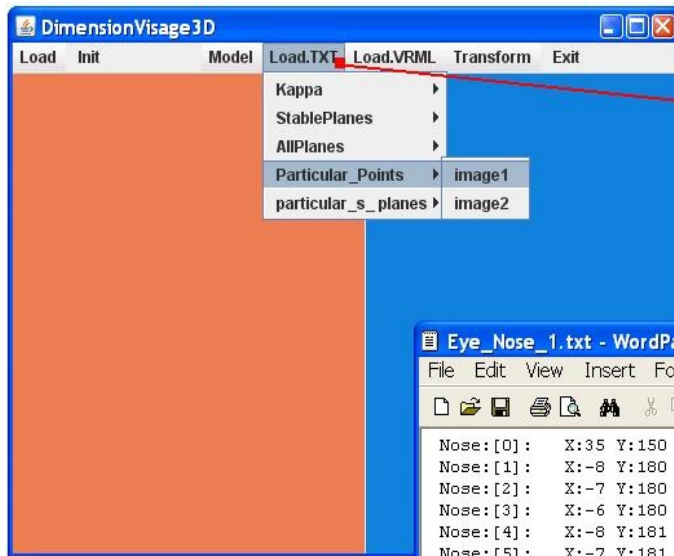
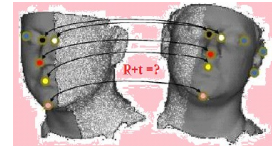
Et en cliquant l'option VRML, l'image 'choisie' sera affichée en VRML.



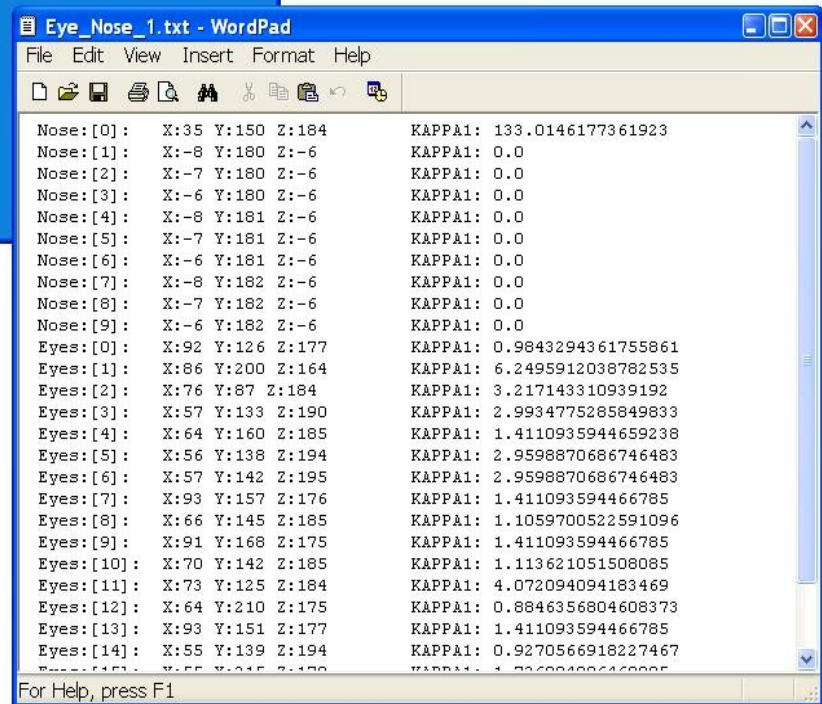
Le bouton **Model-Image1-VRML** sert à ouvrir les images que on choisit dans le format de VRML

◆ Bouton 'Load.TXT':

Puis, le bouton 'Load.TXT' ouvre les fichiers .TXT générés pour plusieurs objects créés comme le bonton '**AllPlanes**' qui ouvre le fichier texte qui contient les équations de tous les plans tangents à l'image. En plus le bonton '**StablePlanes**' sert à afficher des informations sur les plans stables déterminés. Pour le bonton '**ParticularPoints**', on affiche le fichier texte qui contient les points particuliers avec leur coordonnées et leurs indices dans l'image(le nez et les yeux). En outre, on peut afficher les équations des plans particuliers stables en cliquant sur le bouton '**Particular_s_Planes**'. On pourra développer ce menu défilant en lui ajoutant d'autres fonctions qui génèrent d'autres fichiers dont on aura besoin pour l'analyse des résultats.

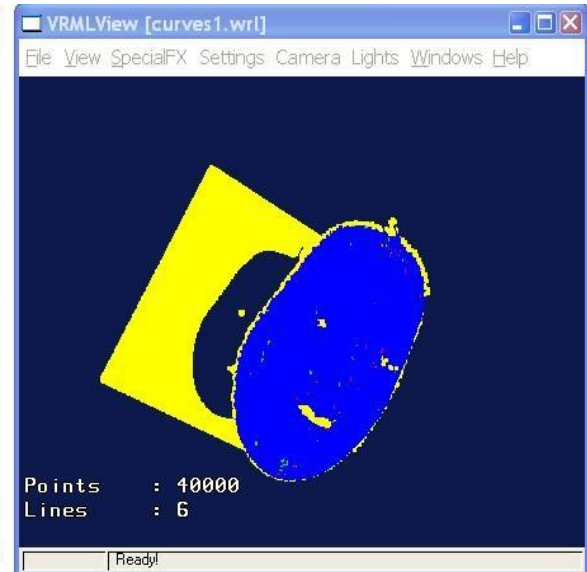
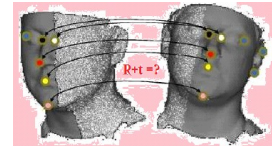


Le bouton **Load.Txt** sert à ouvrir tous les fichiers.txt générés par notre programme.



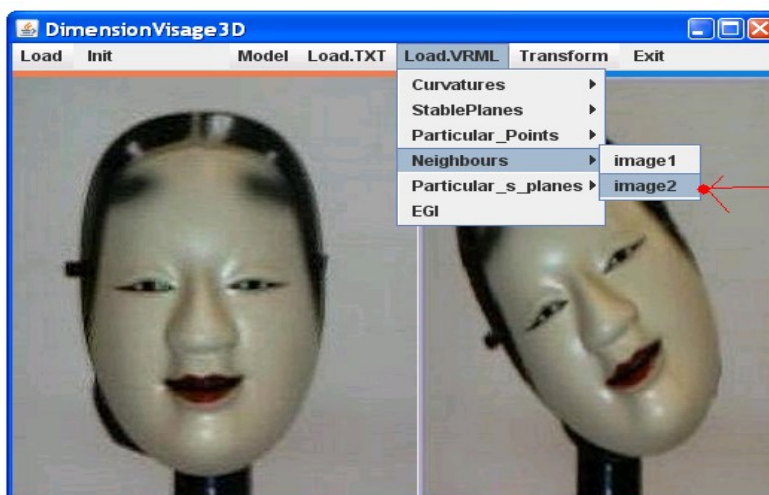
◆Bouton 'Load.VRML':

Pour le bouton 'Load.VRML', on ouvre les fichiers .wrl générés lors de l'exécution du programme. Par exemple, le bouton '**Curvatures**' sert à afficher courbures de l'image sélectionnée (convexités, concavités, plans) en format 3D à l'aide du programme VMRL VIEW. En outre, on peut afficher les points qui appartiennent à un même plan stable avec la même couleur en choisissant le bouton '**StablePlanes**'. De même pour '**Particular.Points**' qui figure dans ce menu défilant, on affiche le nez et les yeux colorisés. Comme vous pouvez remarquer dans la figure de la page suivante, il existe le bouton '**EGI**' qui nous permet d'afficher le visage selon l'algorithme qu'on a déjà expliqué dans la partie théorique.



Le bouton **Load.VRML** sert à ouvrir les images qui ont été calculés sous le format de VRML

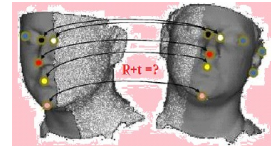
Dans ce menu, figure aussi le bouton 'Neighbours', on peut choisir un point quelconque dont on tape l'indice par clavier et il nous affiche son voisinage colorié.



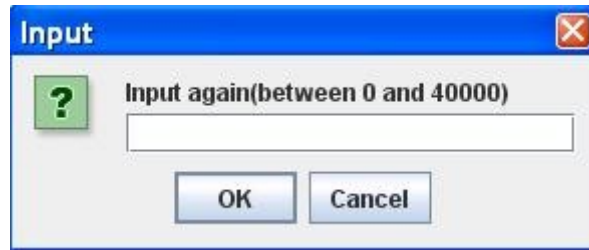
Le bouton **Load.VRML-Neighbours-image** sert à ouvrir la boîte de dialogue



Ici on importe le chiffre pour calculer les images

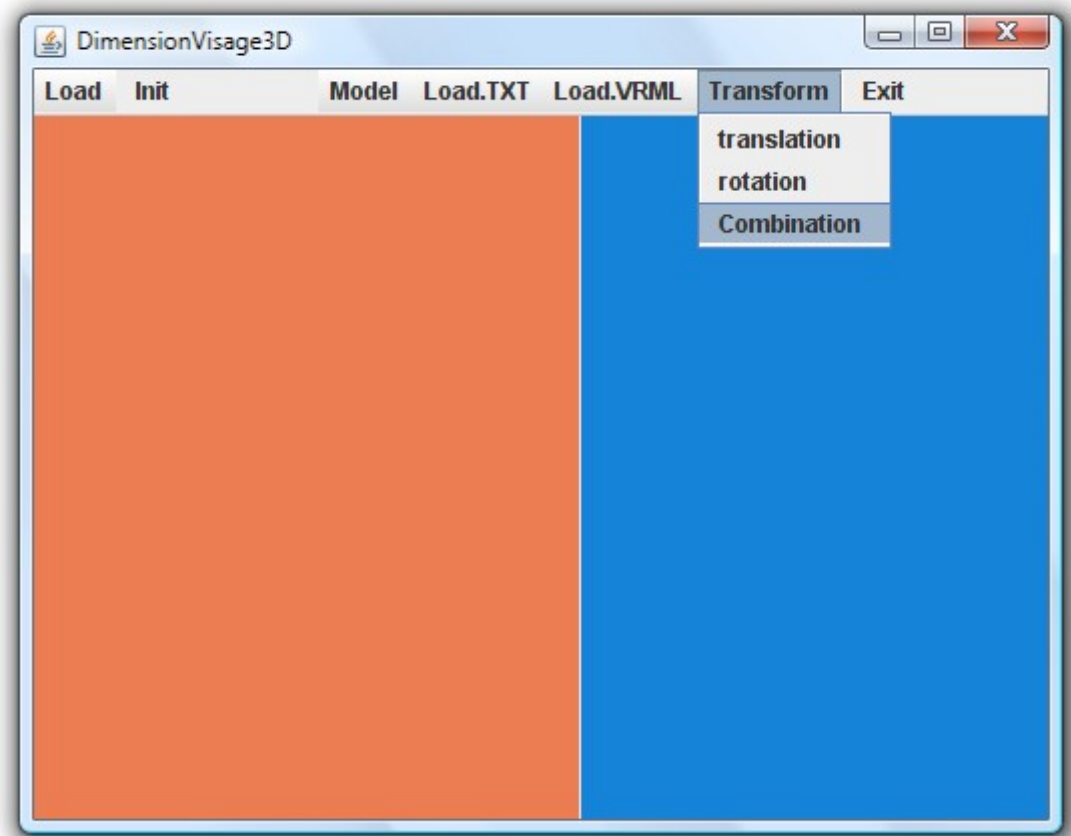


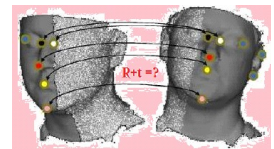
Si le point est plus grand que 40000, la fenêtre sera réafficher jusqu'à ce que le point soit entre 0 et 40000.



♦Bouton ' Transform':

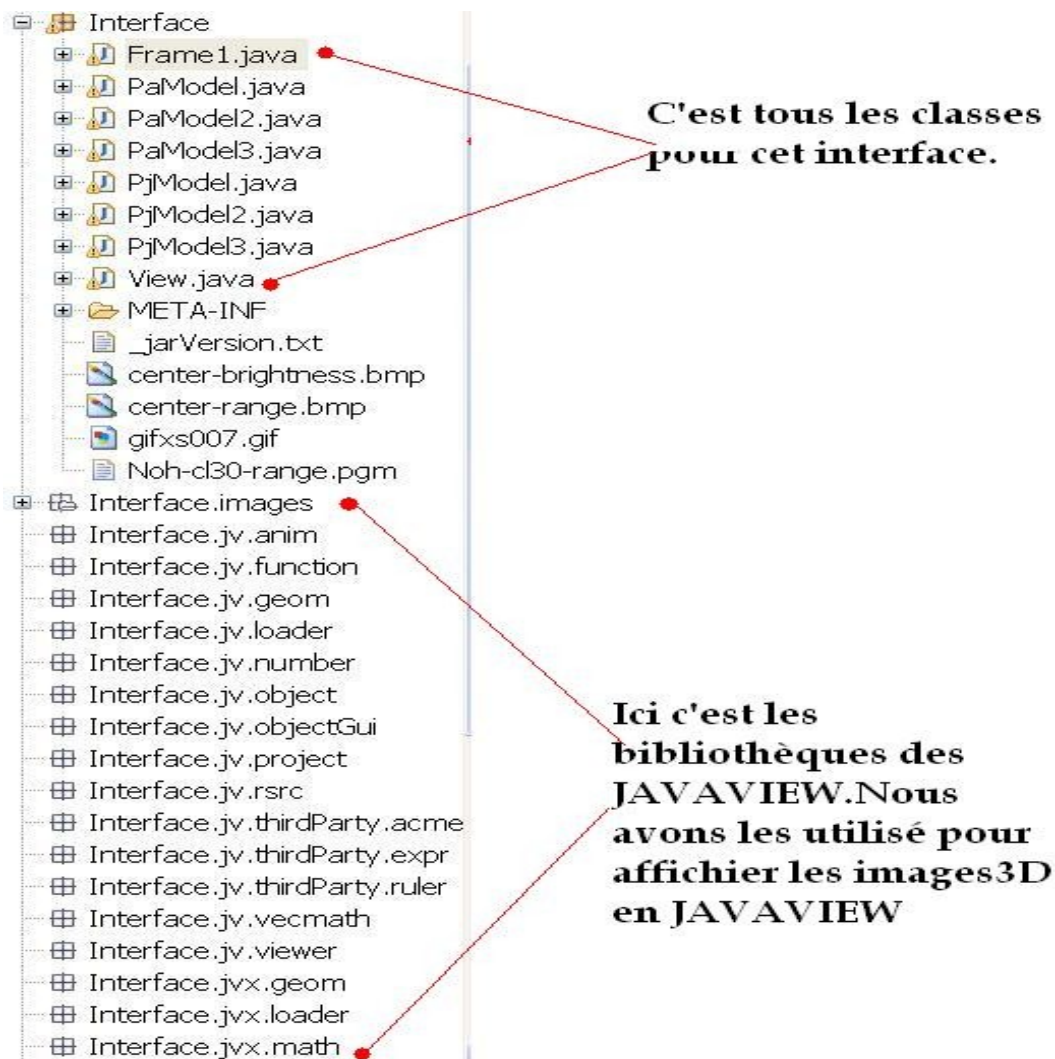
Pour le bouton 'Transform', on peut faire la '**translation**' et '**la rotation**' des images en 3D (la transformation géométrique est établie de l'image1 à l'image2). Quant au bouton '**combination**', il visualise en 3D la superposition de la position initiale avec l'image transformé pour qu'on puisse visualiser l'erreur.



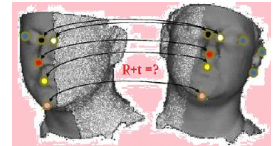


3. Les classes créées en JAVA:

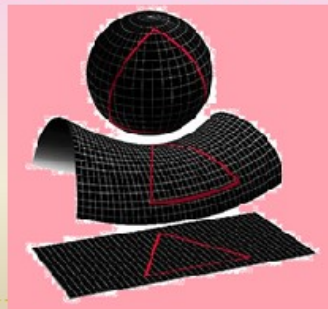
Pour l'implémentation en Java de l'interface graphique, on a crée le **package interface** qui contient la classe Frame1 qui établit l'interface et View qui sert pour la tester.

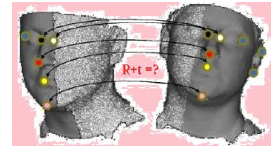


On a aussi importé la bibliothèque JavaView pour pouvoir afficher les images en 3D avec le programme JavaView.



► III- VISUALISATION & ► ANALYSE DES RESULTATS





III- INTRODUCITON: VISUALISATION & ANALYSE DES RÉSULATS

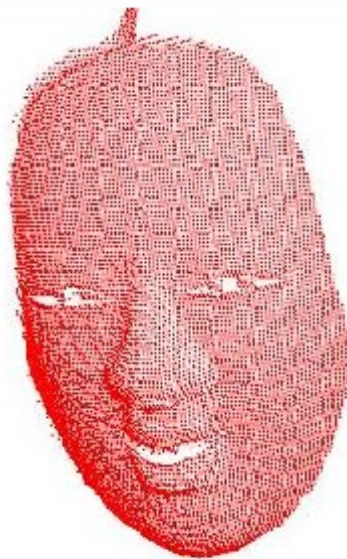
Dans cette partie, on exposera les différentes visualisations générées et on interprètera et analysera avec des diagrammes les résultats obtenus.

La visualisation 3D est basée sur la création de fichier **.wrl** et de les visualiser en utilisant le programme **VRML VIEW**.

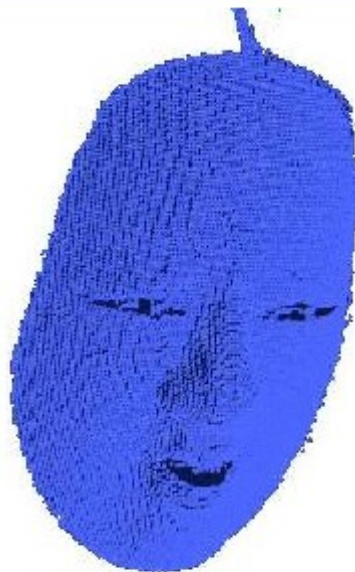


En outre, les **diagrammes** seront créés à l'aide du programme **Excel** en important des fichiers de format TXT associés aux différentes instances créées. Ces diagrammes seront analysés et commentés. Ils nous aideront à comprendre les résultats visuels obtenus pour les différents algorithmes établis dans la partie théorique.

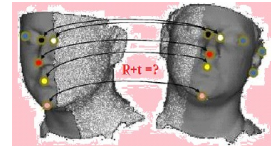
Exemple de visualisation en 3D à l'aide du VRML



Pose initiale



Pose finale

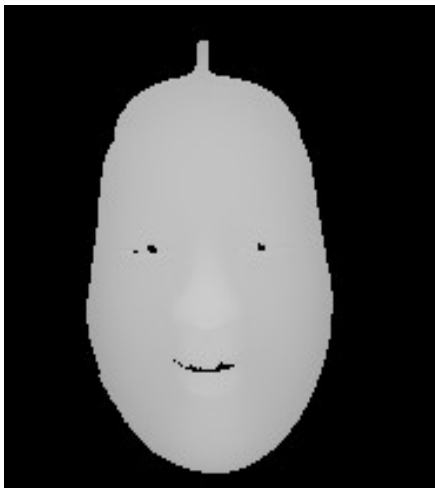


CHAPITRE 1: VISUALISATION DES PARTICULARITES

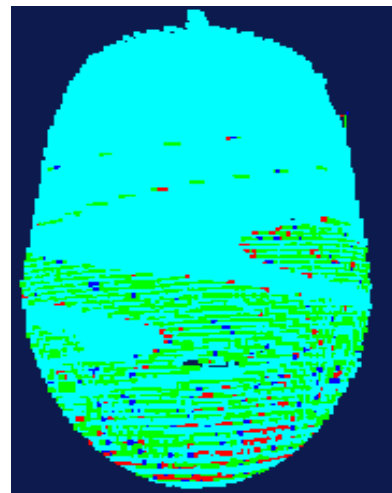
1. EGI:

Le but de cette partie est de visualiser les différentes parties du visage coloriées selon leur type: On a visé à colorier les parties planes comme le front, les joues et la partie inférieure du visage par la couleur rouge, les parties cylindriques comme le nez en vert et enfin les parties convexes en bleu.

Ce travail comme on l'a expliqué dans les parties précédentes dépend du calcul des vecteurs normaux: En premier temps, les vecteurs normaux n'étaient pas normalisés la visualisation a donné le résultat suivant:



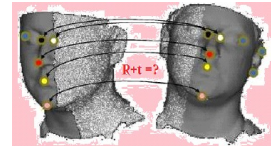
-- image en entrée--



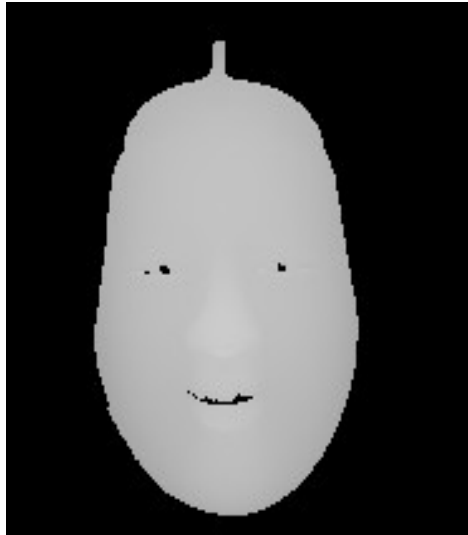
--image de sortie--

Ce résultat est bien évidemment faux parce qu'on observe une couleur verdâtre qui domine, cette couleur correspond aux points qui n'appartiennent ni à un plan, ni à un cylindre, ni à une forme convexe du visage. Il y a aussi un peu petit peu de rouge et de bleu c'est à dire qu'il y a très peu de points qui se trouvent dans des plans ou dans des parties convexes. On remarque aussi l'existence de la couleur verte dans des zones qui devraient être coloriées en rouge.

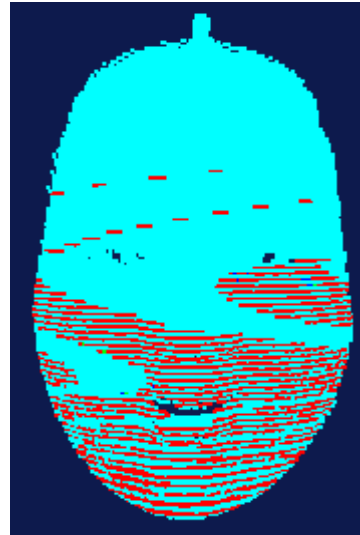
Toutes ces erreurs sont évidentes vu que les vecteurs normaux ne sont pas normalisés, dans ce cas on peut pas avoir la bonne distance maximale entre ces vecteurs ou entre leur projection sur le plan du cercle et donc on peut pas distinguer entre les points de l'image.



Après la normalisation, le résultat est amélioré mais avec des erreurs remarquables:



--image d'entrée--

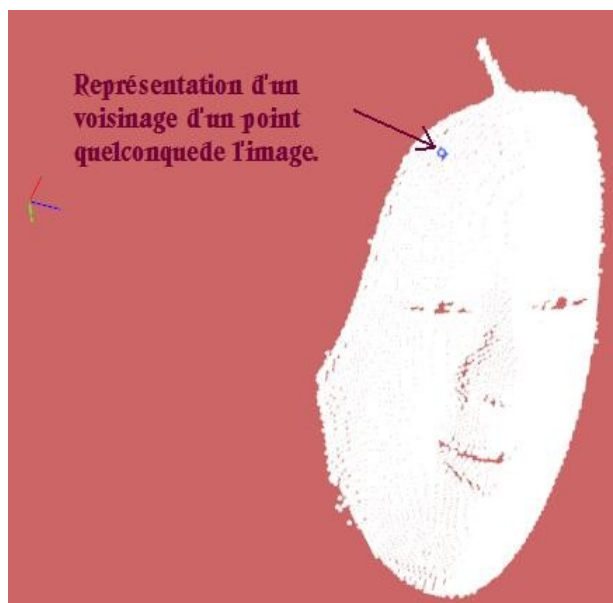


--image de sortie--

On remarque cette fois qu'il a quand même pas mal de rouge dans l'image, un tout petit peu de bleu et une majorité de points en vert. Même après des autres optimisations dans le code, le résultat reste toujours le même (comme ci-dessus).

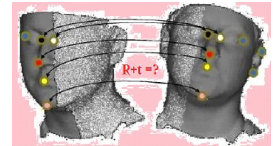
Parmi ces optimisations: essayer de modifier le code complètement ou renormaliser les vecteurs normaux (cas de doute de la bonne normalisation) ou éliminer les cas où les vecteurs normaux ont des z nuls.

2. Vérification de la bonne détection du voisinage:



Cette visualisation d'un voisinage d'un point 3D (zone colorié en **bleu**) appartenant à l'image met en évidence que l'algorithme de la détermination des huit voisins est bien défini.

Comme on a déjà présenté dans la partie II de la programmation en Java, en tapant l'indice du point, on peut afficher son voisinage colorié (dans la menu **Load.VRML**).



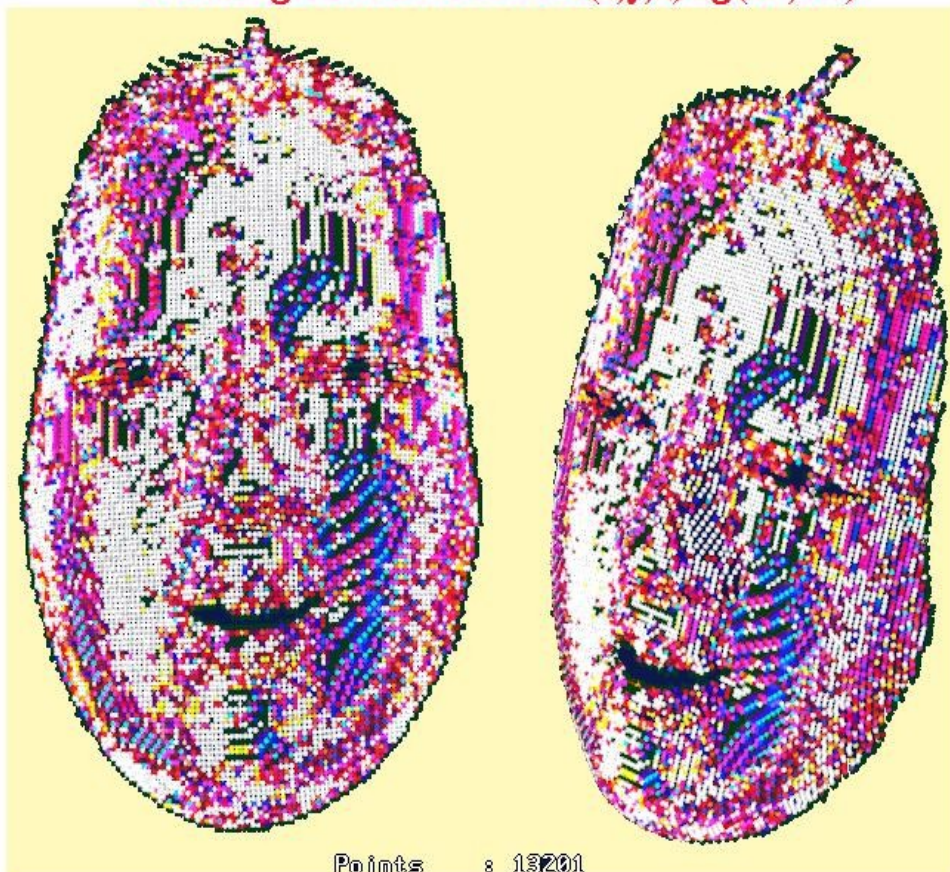
3. Visualisation des courbures:

- Premier test de visualisation de coloriage suivant les valeurs des rayons de courbures κ_1 et κ_2 :

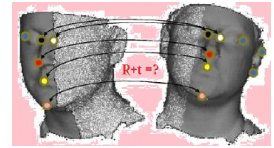
Ce test primitif de coloriage des points de coordonnées (x,y,z) en fonction de leurs κ_1 et κ_2 : $f(x,y,z)=g(\kappa_1,\kappa_2)$ nous a donné pour deux vues différentes (centre et rotation avec 45°) presque la même coloration (voir dans la figure suivante la zone où la couleur bleue domine). Cette visualisation était une première tentative pour voir la variation des couleurs en fonction des courbures. Dans le test suivant, le coloriage sera défini d'une manière rigoureuse basée sur des définitions géométriques.

Voici les deux figures obtenues lors du coloriage RGB selon les formules mathématiques énoncées ci-dessous:

Coloring the curvatures $f(x,y,z)=g(\kappa_1,\kappa_2)$

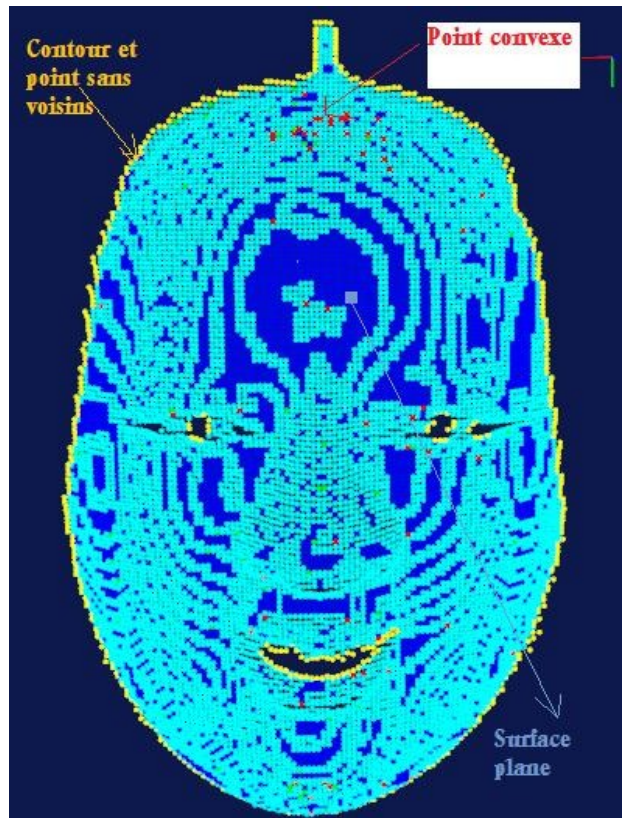


```
red=(Math.abs(max-p.getKappaValue(2))/Math.abs(2*max+max2));
green=Math.abs(max-p.getKappaValue(1))/Math.abs(4*max);
blue=Math.abs ((p.getKappaValue(1)+p.getKappaValue(2)))/(Math.abs(max)+(Math.abs(max2)));
max= max des kappa1; max2= max des kappa2
```



- Coloriage selon l'algorithme énoncé dans la partie théorique:

Les couleurs utilisées pour colorier le visage ci-dessous sont celles qui sont définies dans l'algorithme de la partie théorique. La position est une position centrale.

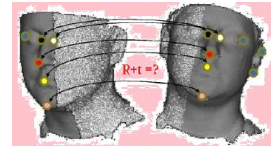


--> Les parties **convexes** (point en **rouge**) et **concaves** (point en **vert**) sont dispersées d'une manière asymétrique par contre le visage est en position centrale. En plus on remarque qu'uniquement des points sont coloriés et non pas des zones entières.

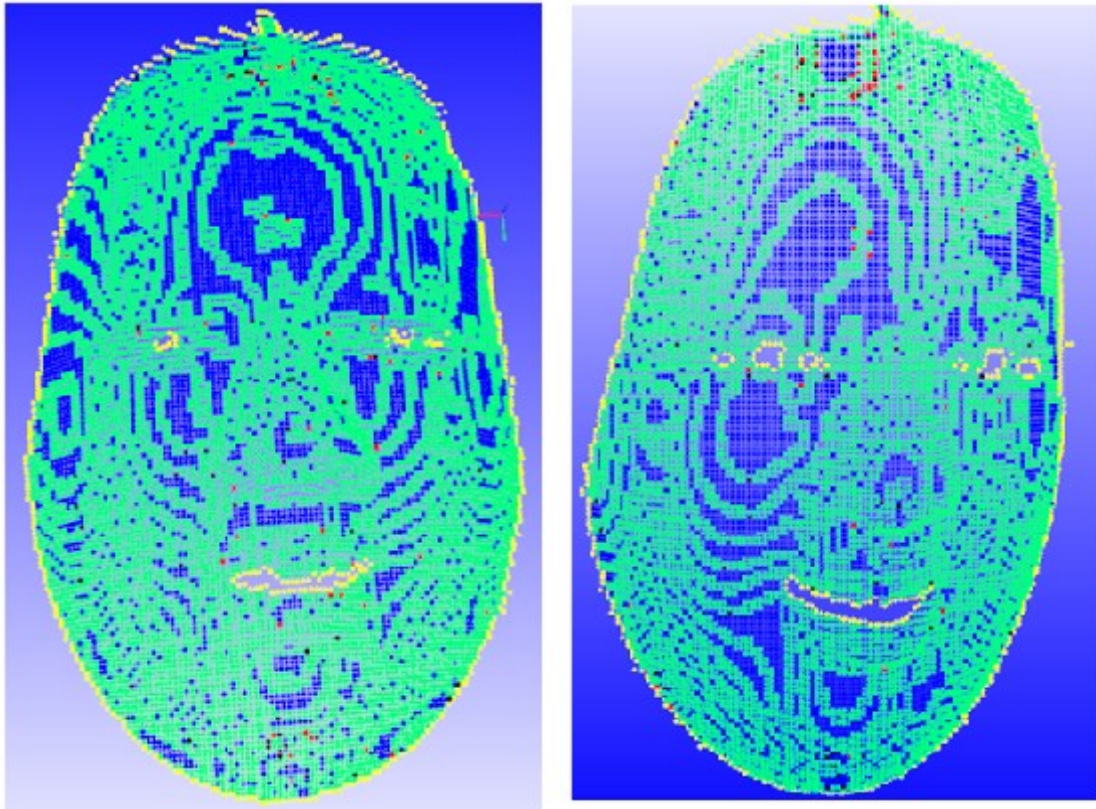
Grâce à la définition du voisinage, on a pu déterminer le **contour** (en **jaune**) des zones qui "n'ont pas des voisins" (càd les points qui ont des voisins avec un $z=0$). La bonne détection du contour vérifie qu'on a bien définie le voisinage.

Comme vous pouvez remarquer, les zones **en bleu foncé** représente les zones **planes** du masque étudié.

Les résultats de visualisation obtenus sont intimement liés aux valeurs trouvées des rayons de courbures (κ_1 et κ_2).



Voici deux autres vues différentes (position centrale et position avec rotation de 30° à gauche) :

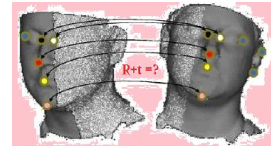


*rouge $\rightarrow k_1 > 0 \ \& \ k_2 > 0$ * noir $\rightarrow k_1 < 0 \ \& \ k_2 < 0$
 *bleu $\rightarrow k_1 = k_2 = 0$ * jaune: points avec des voisins qui ont un $z=0$

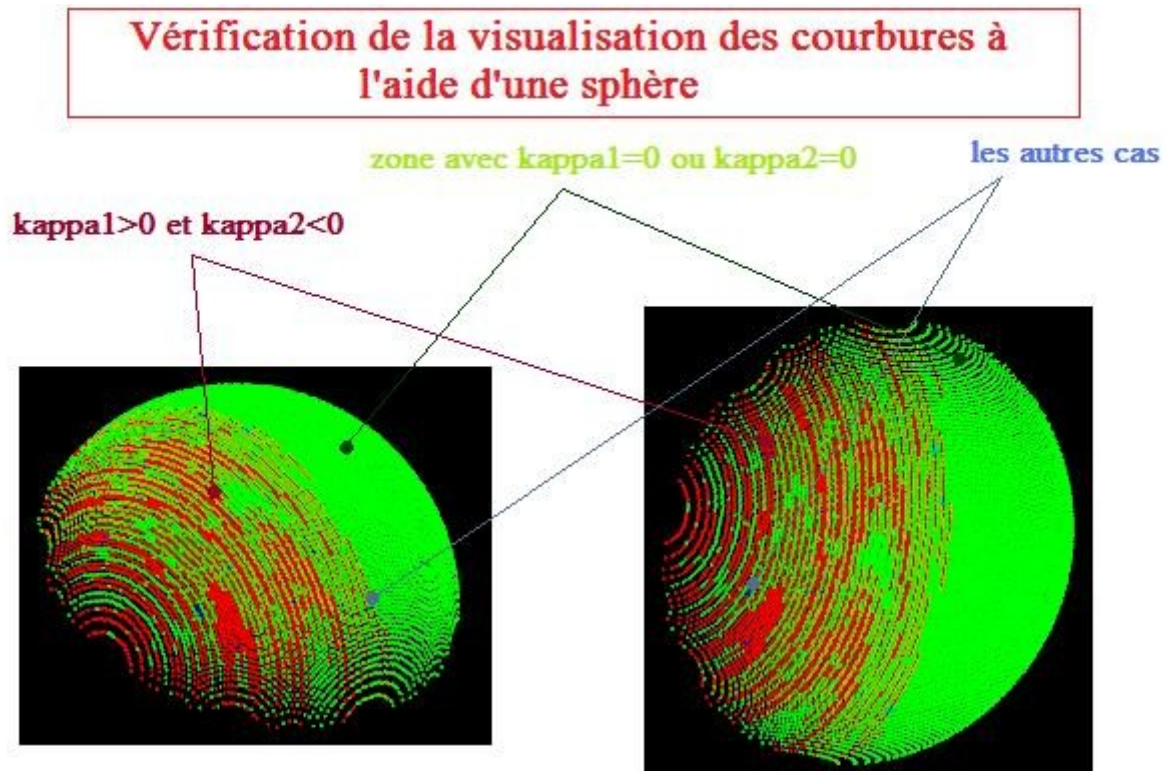
◆ Analyse et interprétations:

- La détection des courbures pour d'autres positions est complètement différente de la vue centrale. Les zones planes en **bleu foncé** ont remarquablement changé et ne sont plus symétriques. Ce qui nous a incité à tester les valeurs des rayons de courbures en utilisant **une sphère**. En effet sur une sphère, on a toujours l'égalité suivante qui reflète une propriété mathématique:

$$\text{kappa1} = \text{kappa2} = \text{constante}$$



- le test de la sphère nous a donné le résultat suivant:



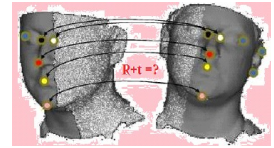
* Algorithme de coloriage de la sphère pour vérifier les valeurs des rayons de courbure:

```
if(p.getKappaValue(2)<0 && p.getKappaValue(1)>0) colorier en rouge
else if(p.getKappaValue(2)=0 || p.getKappaValue(1)=0) colorier en vert
else if (p.getEdge()==true) colorier en jaune
else colorier en bleu
```

- p= PointComplete3D
- getKappaValue(1) --> valeur de κ_1
- getKappaValue(2) --> valeur de κ_2
- getEdge() --> les points dont le voisinage contient un $z=0$

--> On remarque bien que la distribution des valeurs des rayons de courbures n'est pas équiprobable. En effet, on ne dispose pas d'une unique couleur (vert) mais d'autres couleurs (rouge et bleu). Mais le vert est dominant sur la sphère (demi sphère dans la figure ci-dessus).

En conclusion, on peut dire que l'*asymétrie* du coloriage des courbures n'est est justifiée par l'*asymétrie* du coloriage de la sphère.



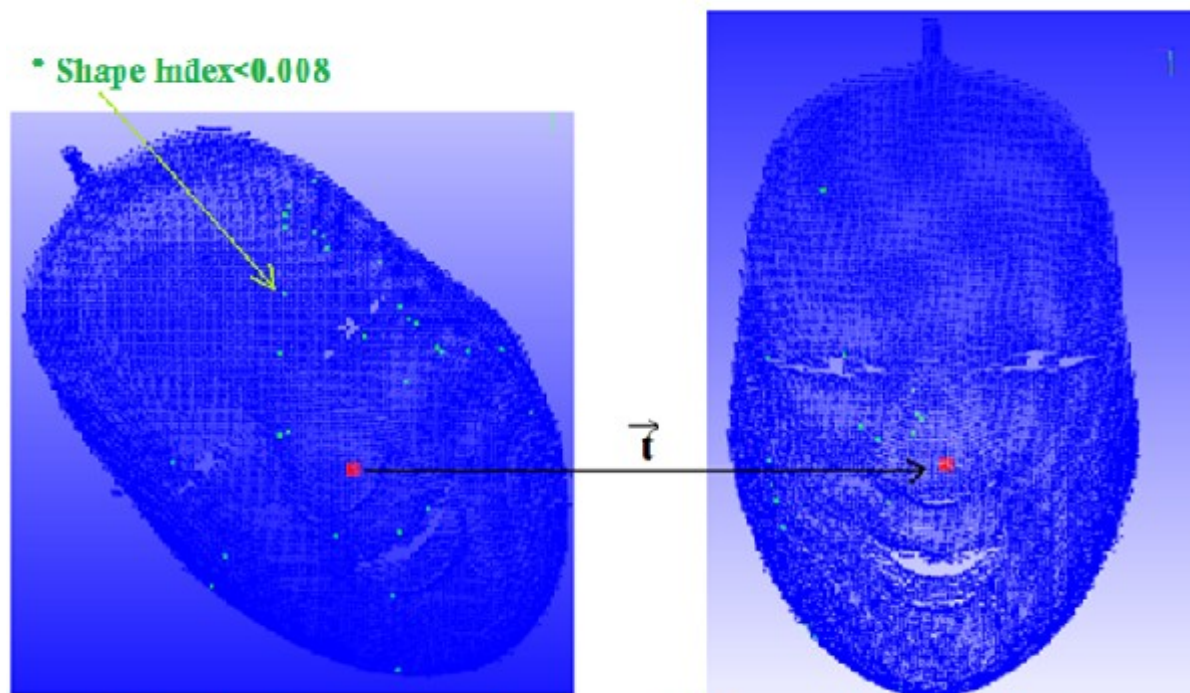
4. Visualisation des points particuliers:

➡ Détection du nez et des yeux::

Les figures obtenues lors de l'exécution de l'algorithme de la détection du nez pour deux positions différentes (vue centrale et une autre avec une rotation de 45°) sont les suivantes. Comme vous pouvez remarquer, en localisant le nez correctement on peut déterminer les coordonnées du vecteur de translation t avec ses coordonnées.

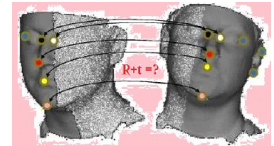
Le **bout du nez** pour ces deux positions est bien détecté (le point maximum et son voisinage sont coloriés en **rouge**).

Les **yeux** sont représentés par les points coloriés **en vert** --> la détection des yeux est *fausse*. Ce résultat peut mettre en interrogation [la méthode du shape index](#) mais puisque ce résultat dépend des valeurs des rayons des courbures (les 'kappa'), on peut dire que les erreurs de calcul des 'kappa' influent sur cette visualisation.



* Détection du nez pour deux vues différentes (45° Rot suivant y et vue de face).

* $\vec{t}$ = vecteur de translation calculé
avec Tx:-42 Ty:31 Tz:-6 pour coordonnées

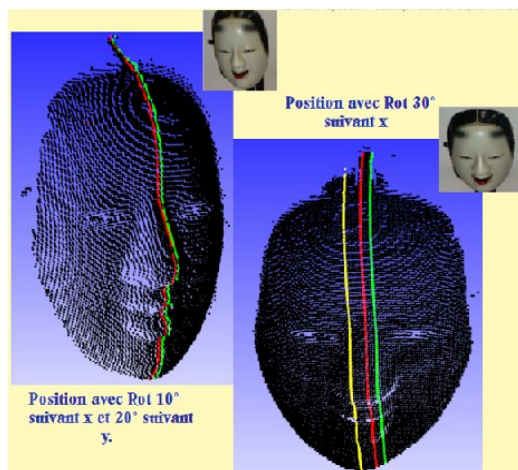


On a aussi généré un fichier de format TXT qui contient les coordonnées des points particuliers. Voici un extrait de ce fichier pour une position avec 45° de rotation suivant z et un seuil pour la détection des yeux égal à 0.008:

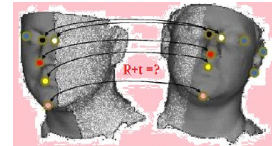
```
Nose:[0]: X:59 Y:70 Z:190
Nose:[1]: X:58 Y:69 Z:190
Nose:[2]: X:59 Y:69 Z:190
Nose:[3]: X:60 Y:69 Z:190
Nose:[4]: X:58 Y:70 Z:190
Nose:[5]: X:59 Y:70 Z:190
Nose:[6]: X:60 Y:70 Z:190
Nose:[7]: X:58 Y:71 Z:189
Nose:[8]: X:59 Y:71 Z:190
Nose:[9]: X:60 Y:71 Z:190
Eyes:[0]: X:94 Y:89 Z:185
Eyes:[1]: X:73 Y:179 Z:177
Eyes:[2]: X:73 Y:83 Z:191
Eyes:[3]: X:85 Y:71 Z:187
Eyes:[4]: X:107 Y:113 Z:179
Eyes:[5]: X:94 Y:87 Z:185
Eyes:[6]: X:35 Y:37 Z:176
Eyes:[7]: X:97 Y:80 Z:182
Eyes:[8]: X:68 Y:149 Z:186
Eyes:[9]: X:95 Y:90 Z:185
Eyes:[10]: X:24 Y:100 Z:177
Eyes:[11]: X:38 Y:45 Z:182
Eyes:[12]: X:30 Y:48 Z:179
Eyes:[13]: X:97 Y:87 Z:183
Eyes:[14]: X:38 Y:164 Z:172
Eyes:[15]: X:32 Y:43 Z:178
Eyes:[16]: X:67 Y:121 Z:191
Eyes:[17]: X:48 Y:102 Z:194
Eyes:[18]: X:43 Y:120 Z:195
Eyes:[19]: X:90 Y:57 Z:183
Eyes:[20]: X:17 Y:72 Z:169
Eyes:[21]: X:54 Y:96 Z:191
Eyes:[22]: X:58 Y:67 Z:190
```

5. Visualisation des plans stables:

En appliquant l'[algorithme de la recherche des plans stables](#) énoncé dans la partie I théorique, on a obtenu les résultats suivants pour deux positions :



- Position avec rotation de 10° suivant l'axe des x et 20° suivant y (*à gauche*)
- Position avec rotation de 30° suivant l'axe des x (*à droite*).



Pour bien expliquer ces résultats, on va représenter pour deux positions différentes des diagrammes de nuage de points qui représentent le nombre de points contenus dans un plan en fonction des plans tangents. Pour cela, on a généré un fichier de format TXT qui associe à chaque plan représenté par un entier le nombre de points qu'il contient. Le fichier a été importé sous Excel.

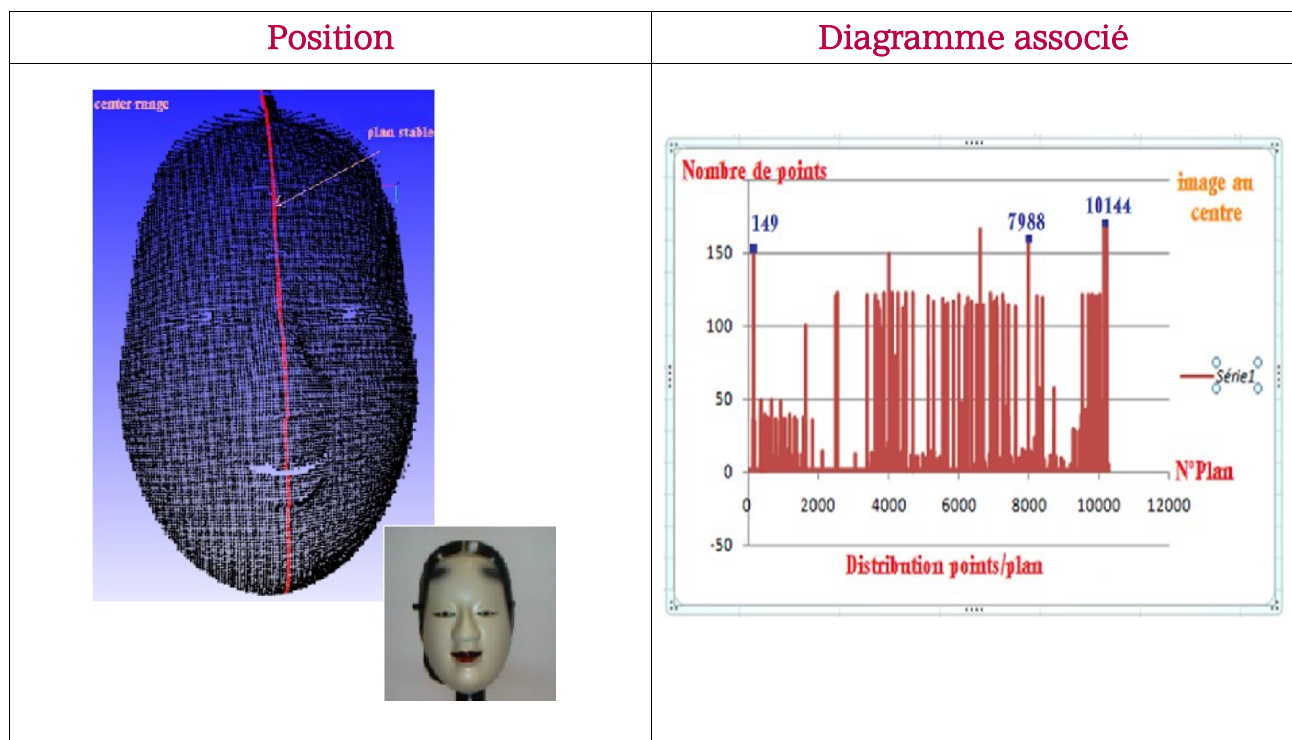
Voici la fonction utilisée:

$$f(p) = n$$

- p: plan
- n: nombre de points qui y appartiennent

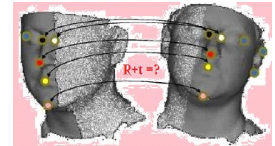
Cette fonction nous permettra d'analyser le critère de stabilité et de vérifier si ces plans caractérisent vraiment une position du masque étudié.

• Vue de face:

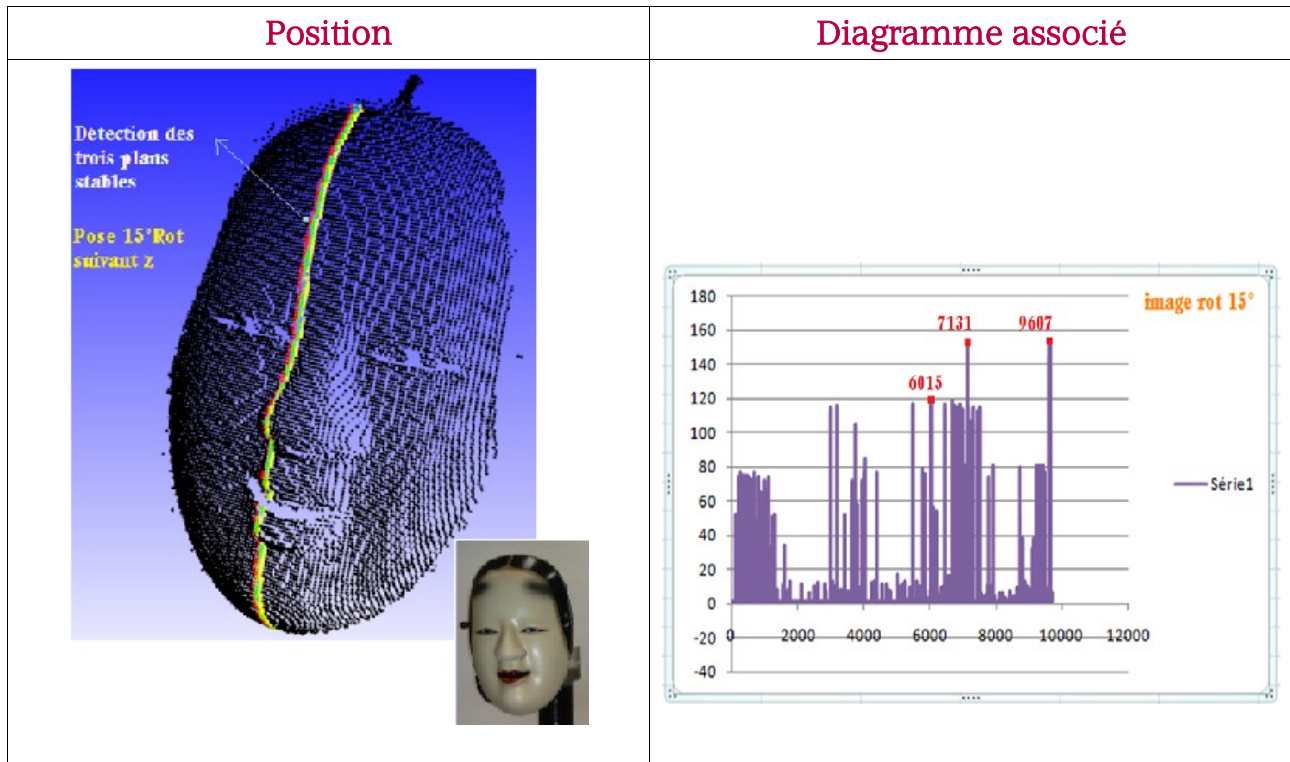


** Analyse et commentaires:

- On remarque l'existence de minorité de plans qui représentent des pics --> (contiennent le maximum de points). Il y a **5 plans** qui sont remarquablement différenciés des autres.
- Les valeurs maximales de $f(p)$ sont au dessus de 150 points. Les nombres qui sont marqués en bleu correspondent à l'indice du plan tangent correspondant.



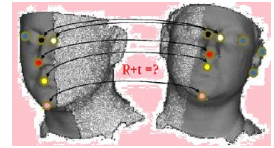
• Vue avec rotation de 15° suivant l'axe des x:



** Analyse et commentaires:

- De même, on remarque l'existence de plans 'particuliers' qui représentent des *maxima pour la fonction $f(p)$* . Dans ce cas de figure, il existe 3 plans différenciés des autres.
- Ces diagrammes justifient *la particularité* de ces plans détectés et par suite on peut postuler qu'ils vérifient **un critère de stabilité**. En plus, on remarque que pour ces deux positions, ils présentent *une symétrie* pour le masque. Et on arrive à identifier *des zones identiques* (plan vertical de symétrie du visage) pour deux positions différentes.

--> Les trois vecteurs normaux ou les trois points 3D qui leur sont associés et qui définissent les trois plans tangents stables seront utilisés pour déterminer *la rotation*.



CHAPITRE 2: ANALYSE DE LA TRANSFORMATION GEOMETRIQUE

Visualisation de la translation:

Le vecteur de translation est généré dans un fichier TXT comme on a expliqué lors de la détection du nez. ([cliquer ici](#))

En plus, on visualise la translation en 3D en créant la transformée de l'image dans un fichier VRML.

Visualisation de la rotation:

La rotation ne peut pas être visualisé en 3D pour des raisons qu'on expliquera dans la partie analyse. Mais on a pu généré un fichier TXT qui contient *l'angle de la rotation en radian* et les coordonnées du vecteur directeur de *l'axe de la rotation*.

Voici deux exemples pour la caractérisation de la rotation pour des positions différentes et en plus avec deux méthodes différentes:

Détermination de l'angle et l'axe

Position de face VehA 45° Rot

1- Méthode des points particuliers:

Angle of Rotation= 2.115435754614661
Axis of Rotation: 3D X: -0.9647220301267312 Y: 0.017899848137461444 Z: -0.9999945869216185

2- Méthode des plans stables:

Angle of Rotation= 2.003404612864847
Axis of Rotation: 3D X: 0.33509663292248476 Y: 0.04302875085895152 Z: 0.999999974155097

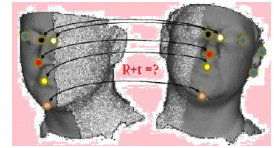
De la position 30° suivant y vehA une autre avec 15° suivant x

1- Méthode des points particuliers:

Angle of Rotation= 0.48532727526254654
Axis of Rotation: 3D X: -0.36280508013887763 Y: 0.529575516422101 Z: 0.9999998606007212

2- Méthode des plans stables:

Angle of Rotation= 2.003404612864847
Axis of Rotation: 3D X: 0.33509663292248476 Y: 0.04302875085895152 Z: 0.999999974155097



◆ Analyse et commentaires:

Si le calcul de la translation fonctionne comme il faut, la rotation quand elle ne peut pas être vérifiée.

En effet lors du calcul de la rotation, nous manipulons des données comme l'angle de rotation et les coordonnées de l'axe de rotation qui sont de type *double*. Or, lorsque l'on applique une rotation à une image afin de vérifier le calcul, nous effectuons des opérations entre des objets de type *double* et des objets de type *int*. Nous sommes donc obligés de caster les objets de type *double* en *int*, ce qui provoque une perte d'information non négligeable :

```
PointComplete3D pOuter = new PointComplete3D(
    (int)(rot.getAxis().getY()*p.getZ() - rot.getAxis().getZ()*p.getY()),
    (int)(rot.getAxis().getZ()*p.getX() - rot.getAxis().getX()*p.getZ()),
    (int)(rot.getAxis().getX()*p.getY() - rot.getAxis().getY()*p.getX()));
```

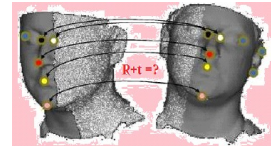
Ce cast est nécessaire en raison du type des données. En effet, les images sont codées à l'aide d'entiers stockés dans un tableau (cf la classe Point3D et sa classe fille PointComplete3D). Les coordonnées de chaque point sont donc des entiers. De plus, les opérations effectuées sur l'axe et l'angle de rotation (telles que le calcul du cosinus par exemple) nécessite obligatoirement un objet de type *double* pour l'implémentation.

Ainsi, la rotation peut être calculée mais pas vérifiée. Une solution pourrait consister à changer le type des données afin de permettre à l'utilisateur de pouvoir à son gré utiliser des données du type qu'il désire.

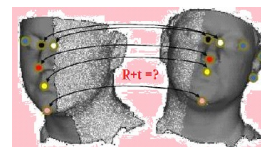
Cependant, bien que nous ne pouvons pas appliquer la rotation calculée à une image, nous pouvons cependant avoir une piste quant au bienfondé du calcul en vérifiant que l'angle de rotation calculé par nos soins est bien proche de l'angle de rotation entre l'image de départ et l'image résultat lorsque celui-ci est connu.

Il faut tout de même préciser que cette comparaison n'est possible que si la rotation est plane et que l'axe de rotation calculé est normal à ce plan. En effet le changement de référentiel opéré pour le calcul empêcherait sinon la comparaison.

Par exemple, nous pourrions vérifier le calcul de la rotation entre les deux images suivantes :



En prenant comme nouvelle origine pour la génération des deux vecteurs le nez (et l'axe de rotation serait donc le vecteur normal au nez), l'angle trouvé avec notre implémentation pourrait aisément être comparé à l'angle connu entre ces deux images.



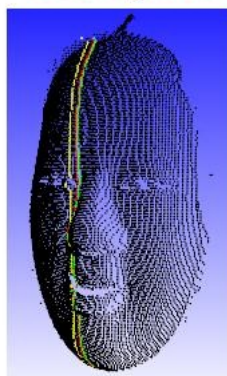
CHAPITRE 3: FAILLES ET SOLUTIONS POSSIBLES

Dans cette partie, on discutera *les failles* du projet en analysant les résultats obtenus pour plusieurs échantillons (masques). Par conséquent, on essaiera de proposer *des solutions* probables. Dans des parties précédentes, on a déjà présenté quelques explications des résultats obtenus. Cependant, cette partie sera plus déductive et plus générale.

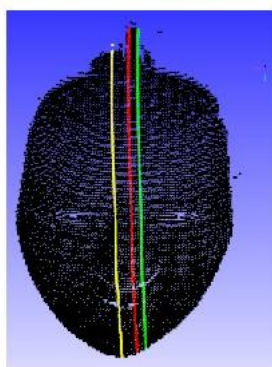
Points Particuliers et Plans Stables

Voici un les résultats de visualisation obtenus pour 4 positions différentes pour la détermination des *plans stables* et la localisation du *nez* et *des yeux*.

Comparaison des résultats pour plusieurs échantillons



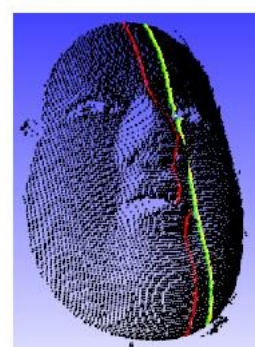
Rot 15° selon z



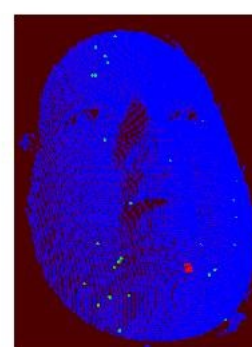
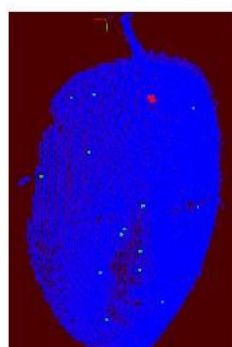
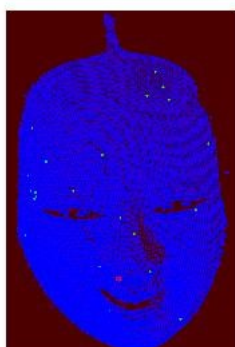
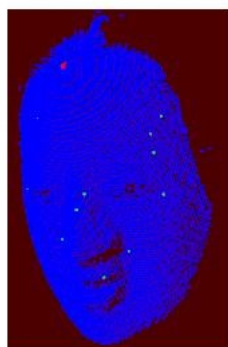
Rot 30° selon x



Rot 10° selon z, 20° selon x



Rot 10° selon z, 45° selon x

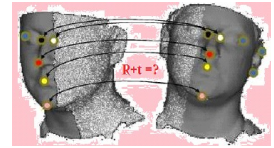


Failles

1- Mauvaise détection du nez pour certaines positions

Analyse

- Ce résultat dépend de [l'algorithme de la localisation du nez](#) et des valeurs calculées pour [les rayons de courbures](#) car on cherche le maximum de $(k_1^2 + k_2^2)$. *<cliquer sur les liens>*



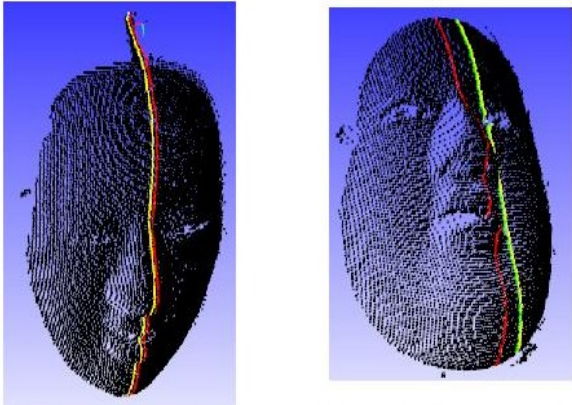
1- Solutions

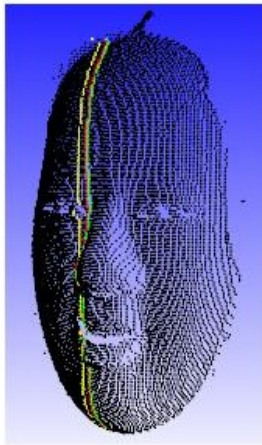
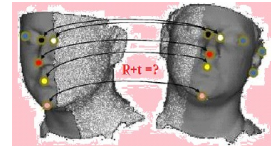
*** Les solutions qui peuvent être proposées sont les suivantes ***

- 1- vérifier l'algorithme de la détection du nez.
- 2- chercher un autre algorithme plus efficace.
- 3- vérifier que les outils de bases qu'on utilise (les valeurs des kappa) sont bonnes.
- 4- **redéfinir les outils de bases**: les rayons de courbures sont calculés à partir de quatre valeurs des vecteurs normaux.
- 5- **changer la structure des données**: on utilisera une matrice tridimensionnelle au lieu d'un tableau unidimensionnel pour enregistrer les coordonnées (x,y,z) des points 3D. Ainsi, pourrait-on changer les valeurs des coordonnées en valeurs réelles en dépit de l'espace mémoire qu'on occupera et du temps de calcul qui augmentera. Mais les calculs des vecteurs normaux et des rayons de courbures seront plus précis et la résolution de l'image s'améliorera.

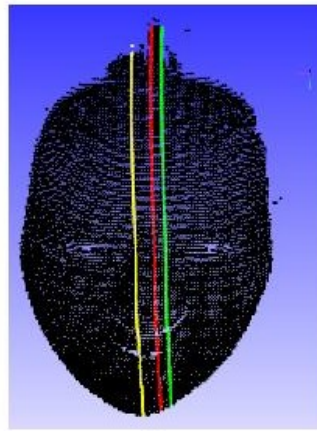
En contre partie, on remarque **une bonne détection des plans stables** pour des positions différentes --> ce qui justifie que ces plans présentent un critère de stabilité. Par suite, on espère bien trouver la bonne valeur de l'angle de la rotation et de son axe à l'aide de cette méthode.

D'où le tableau suivant représentant les plans stables détectés pour deux positions différentes et la rotation qui résulte de cette détection:

Plans Stables		Rotation: axe et angle
		Angle of Rotation= 1.9707434864384785 Axis of Rotation: X: -0.7785869797768854 Y: -0.6809872641228815 Z: 0.9999982282867865



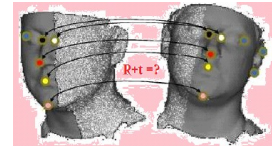
Rot 15° selon z



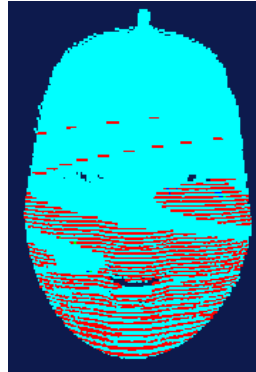
Rot 30° selon x

Angle of Rotation= 2.003404612864847
Axis of Rotation: X: 0.33509663292248476
Y: 0.04302875085895152
Z: 0.999999974155097

---> L'angle de rotation est bien déterminé en radian. Mais vu qu'on ne peut pas afficher la superposition de la première position transformée avec la 2ème position à cause de *la structure des données* comme on a déjà mentionné dans la partie III, chapitre II; on peut estimer que le résultat est bon car les plans stables sont bien détectés.



La méthode de EGI



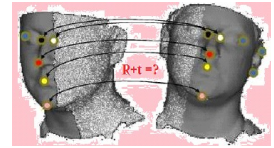
Faibles

1-Le front et les joues doivent être coloriés en rouge ---> représentent des formes planes!
2-Le nez doit être vert---> dans ce cas il est à moitié verte et à moitié rouge.

Solutions

La détection de ces particularités dépendent fortement du calcul des vecteurs normaux et de leur normalisation et des algorithmes de calculs des différentes distances maximales qui font la différence entre les points du visage ---> les solutions possibles sont alors:

- 1- Vérifier le calcul des vecteurs normaux.
- 2- Vérifier que ces vecteurs sont bien normalisés
- 3-Vérifier que les algorithmes dont on a parlé donnent les bons résultats ou essayer de chercher d'autres algorithmes plus efficaces.



** CONCLUSION **

A partir de la photo de deux masques nous avons donc réussi, après conversion de la photo d'un masque (2D) en une représentation de l'image en niveaux de gris (3D), à repérer la position d'un visage en 3D par rapport à un autre.

Nous avons tout d'abord mis en avant certaines caractéristiques des points de l'image (comme le vecteur normal ou les kappa par exemple), ceci dans le but de pouvoir détecter au mieux les particularités de l'image. De la bonne détection de ces particularités dépend l'estimation correcte de la transformation entre les deux images qui est réalisée à l'aide du calcul de la translation et de la rotation existantes entre les deux images.

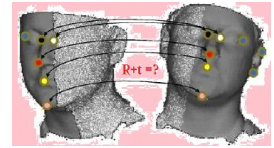
Cependant, notre implémentation possède quelques limites. Tout d'abord, pour la détection des particularités de l'image à l'aide de la méthode des plans stables, nous avons obtenu un temps d'exécution de 2 minutes en implémentant un algorithme non linéaire. Il serait intéressant de pouvoir diminuer la complexité du temps de calcul des plans tangents qui est actuellement en $O(n^2)$.

Par ailleurs, nous avons remarqué que la détection des particularités de l'image, notamment la détection qui s'appuie sur la méthode de l'EGI, pouvait être améliorée en redéfinissant certains outils de base ; par exemple, nous pourrions augmenter le nombre de points dans le voisinage utilisé pour le calcul du vecteur normal, ce qui nous permettrait d'obtenir des résultats plus précis.

De plus, nous avons utilisé ici trois méthodes de détection des particularités de l'image (celle des plans stables, des points particuliers et celles de l'EGI) mais d'autres méthodes basées sur des outils mathématiques, que nous avons choisies de ne pas développer dans notre projet, pourrait être intéressantes à implémenter. A titre d'exemple, on peut citer la méthode d'ICP (Iterative Closest Point).

Remarquons également que les problèmes rencontrés lors de l'estimation de la position sont essentiellement dûs au fait que les calculs sur les points de l'image 3D sont effectués en *discret* sur des *coordonnées entières*. Ce qui nous a empêché de visualiser la transformée de l'image par rotation comme on a déjà expliqué.

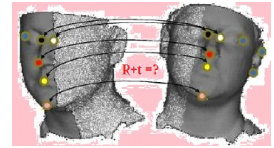
Les difficultés que nous avons rencontrées lors de la réalisation de ce projet nous ont permis de mettre en avant trois questions primordiales que l'on doit se poser lors de l'implémentation des objets manipulés dans une application graphique :



- Quel codage mathématique des objets utiliser ?
- Comment définir les outils mathématiques utilisés pour réaliser des manipulations ?
- Comment choisir la structure et le type des données manipulées?

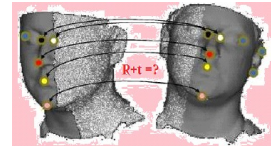
Notons aussi qu'il est du plus grand intérêt de structurer au maximum les données que nous manipulons et de tenir le compromis entre le temps d'exécution et l'espace mémoire d'une part et l'amélioration de la résolution et des structures des données d'autre part.

Enfin, nous avons pris conscience de toutes ces difficultés et dans le soucis de rendre le code plus facilement réutilisable, nous avons modifié la structure des données afin de permettre à l'utilisateur de choisir le type de données qu'il souhaite utiliser (*int*, *double*, *float*, etc) et peut être d'aboutir à des résultats meilleurs que ceux qu'on a obtenus.



REMERCIEMENTS

Toute l'équipe qui a travaillé sur ce projet adresse ces profonds remerciements à notre suiveur Mme *Yukiko Kenmochi*.



BIBLIOGRAPHIE

♦ Rapport scientifique de recherche:

- doc 1: "Three-Dimensional Model Based Face Recognition" , article scientifique rédigé par *Xiaoguang Lu, Dirk Colbry and Anil K.Jain*.
- doc 2: "Pose estimation of two-pose 3D models using the base tangent plane and stability constraints", article scientifique rédigé par *Soon-Yong and Murali Subbarao*.
- doc 3: "Estimating Curvatures and Their Derivatives on Triangle Meshes", article rédigé par *Szymon Rusinkiewicz and Princeton University*.
- doc 4: "Curvature-based face Surface Recognition Using Spherical Correlation", rédigé par *Hiromi T. Takana, Masaki Ikeda and Hisako Chiaki*.
- doc 5: "Matching Range Images of Human Faces", rédigé par *Jhon C. Lee and Evangelos Milios*.

♦ Site web:

http://en.wikipedia.org/wiki/Rodrigues%27_rotation_formula

www.sciencedirect.com

<http://people.csail.mit.edu/bkph/AIM/AIM-740-OPT.pdf>

<http://people.csail.mit.edu/bkph/papers/EGI.pdf>

<http://www.cs.jhu.edu/~misha/Fall04/EGI1.ppt#5>

<http://www.commission3.isprs.org/laserscanning2005/papers/180.pdf>

<http://csidoc.insa-lyon.fr/these/1996/moron/c1.pdf>

<http://pagespro->

orange.fr/gilles.costantini/Lycees_fichiers/CoursT_fichiers/PSPV.pdf

<http://promenadesmaths.free.fr/produitvectoriel.htm>