TP 2 : PCA et SVD

Au début de ce TP/TD, vous recevrez une archive `zip` contenant une base de code. Ce code permet d'afficher un **maillage triangulaire** à l'aide d'`openGL`.

1. Vous trouverez la description du code dans le sujet.
2. Vous devez faire évoluer ce code au fur et à mesure du TP, pour répondre aux questions. Vous rendrez votre code complété dans un fichier `.zip`.
3. Un compte rendu rédigé dans un fichier `pdf` de 4 pages maximum devra rendre compte de votre travail. Il contiendra des réponses rédigées aux questions éventuellement accompagnées de captures d'écran de la fenêtre de rendu quand cela est pertinent.

Le compte-rendu et le code seront à téléverser sur le moodle du cours : `https://moodle.umontpellier.fr/course/view.php?id=22845`.

Base de code

Installation

Téléchargez l'archive sur le moodle `https://moodle.umontpellier.fr/course/view.php?id=22845`. Pour compiler le code et l'exécuter :

```
$ make
$ ./tp
```

Interactions utilisateur

```
1 void key (unsigned char keyPressed, int x, int y)
```

La fonction `key` permet de d'interpréter les entrées clavier utilisateur. Les options de visualisation activées par des touches sont les suivantes, en appuyant sur la touche :

- `n` : activation/désactivation de l'affichage des normales,
- `1` : activation/désactivation de l'affichage du modèle d'entrée sur lequel vous effectuez les calculs de normale,
- `2` : activation/désactivation de l'affichage du modèle transformé,
- `s` : changement entre l'affichage avec les normales de face et de sommet (maillage plus lisse),
- `w` : changement du mode d'affichage (fil de fer/éclairé/non-éclairé/fil de fer + éclairé),
- `b` : activation/désactivation de l'affichage du repère ACP sur le modèle d'entrée,
- `z` : activation/désactivation de l'affichage, sur le modèle transformé, de la transformation du repère ACP du modèle d'entrée,
- `p` : activation/désactivation de l'affichage des plans ACP sur les modèle d'entrée,
- `+` : changement d'axe/plan pour la projection,
- `f` : activation/désactivation du mode plein écran.

Vous pouvez interagir avec le modèle avec la souris :

- Bouton du milieu appuyé : zoomer ou reculer la caméra,
- Clic gauche appuyé : faire tourner le modèle.

1 Transformation

Le fichier `tp.cpp` contient un `struct Transform` permettant d'appliquer une transformation linéaire $M \in \mathbb{R}^{3\times3}$ et une translation $t \in \mathbb{R}^3$ aux points du maillage $x_i \in \mathbb{R}^3$ avec $i = 1, \cdots, n$. La normale unitaire au triangle τ_k est noté $n_k \in \mathbb{R}^3$, $k = 1, \cdots, m$. Ainsi, le nombre de points du maillage est n et le nombre de triangles est m. Dans le code x_i est noté `i_position`, n_k est noté `k_vector`, M est notée `m_transformation` et la translation t est noté `m_translation`. Comme dans le TP précédent, vous pouvez observer un maillage initial et celui transformé.

```
//Transformation made of a matrix multiplication and translation
struct Transform {

protected :
    Mat3 m_transformation ; //Includes rotation + scale
    Mat3 m_vector_transformation ; //Transformation matrix for vectors
    Vec3 m_translation ; //translation applied to points
public :

    //Constructor, by default Identity and no translation
    Transform(Mat3 i_transformation = Mat3 : :Identity(), Vec3 i_translation = Vec3(0., 0., 0.))
            : m_transformation(i_transformation), m_translation(i_translation) {
        //Question 1.3 : TODO, calculer la matrice transformation m_vector_transformation à appliquer aux vecteurs normaux.
        //m_vector_transformation = ... ;
    }

    //Fonction pour appliquer la transformation à un point Vec3
    Vec3 apply_to_point(Vec3 const &i_position) {
        return m_transformation * i_position + m_translation ;
    }

    //Transformation à appliquer à un vecteur
    Vec3 apply_to_vector(Vec3 const &k_vector) {
        return m_vector_transformation * k_vector ;
    }

    //Transformation appliquer à un vecteur normalisé  : example une normale
    Vec3 apply_to_normalized_vector(Vec3 const &k_vector) {
        Vec3 result = k_vector ;
        //Question 1.4 : TODO, appliquer la matrice de transformation de vecteurs et normaliser
        return result ;
    }

    Vec3 const &translation() { return m_translation ; }

    Mat3 const &transformation_matrix() { return m_transformation ; }

    Mat3 const &vector_transformation_matrix() { return m_vector_transformation ; }
} ;
```

1.1. Testez différentes transformations M en mettant à jour la matrice `transformation` utilisée pour construire l'objet `mesh_transform` appliqué à l'objet :

```
Transform mesh_transform (transformation, translation) ;
```

Que constatez-vous si vous appliquez une transformation qui n'est pas orthogonale (en particulier sur le champs de vecteur normaux noté `k_vector` dans le code) ? Illustrer vos propos par une capture écran.

1.2. Fixons $k \in \{1, \cdots, m\}$ et considérons un vecteur $v \in \tau_k$. On a donc $\langle v, n_k \rangle = 0$ car n_k est le vecteur unitaire normal à τ_k. Rédigez un petit raisonnement pour trouver quelle matrice $B \in \mathbb{R}^{3\times3}$ (dans le code `m_vector_transformation`) il faut appliquer au vecteur normal n_k pour avoir $\langle Mv, Bn_k \rangle = 0$ (c'est à dire, que le champs de vecteur normal transformé soit orthogonal au triangle transformé $M\tau_k$). A-t-on $\langle Bn_k, Bn_k \rangle = 1$?

1.3. Avec la question précédente, compléter le constructeur de `Transform` en implémentant `m_vector_transformation` la matrice de transformation à appliquer aux vecteurs.

1.4. Compléter la fonction `apply_to_normalized_vector` permettant d'appliquer la transformation à un vecteur unitaire. Attention, elle doit renvoyer un champs de vecteur unitaire. Illustrer cela par une capture d'écran.

2 Analyse en Composantes Principales

L'Analyse en Composantes Principale sera calculée sur les points du maillage d'entrée et la base affichée sera alignée sur les directions principales. Les points du maillage seront ensuite projetés sur celles-ci (touche `b`) ainsi que sur les plans correspondants (touche `p`). La fonction suivante est dans le fichier `tp.cpp`

```
void compute_principal_axis(std : :vector < Vec3 > const & ps,
                            Vec3 &eigenvalues,
                            Vec3 &first_eigenvector,
                            Vec3 &second_eigenvector,
                            Vec3 &third_eigenvector ) {

    Vec3 p(0., 0., 0.) ;
    for(unsigned int i=0 ; i<ps.size() ; ++i)
        p += ps[i] ;
    p /= (float)ps.size() ;

    Mat3 C(0, 0, 0, 0, 0, 0, 0, 0, 0) ;
    for( unsigned int i=0 ; i<ps.size() ; ++i ) {
        Vec3 const& pi = ps[i] ;
        C += Mat3 : :tensor( pi - p, pi - p ) ;
    }
    C = (1./ps.size()) * C ;

    C.diagonalize(eigenvalues, first_eigenvector, second_eigenvector, third_eigenvector) ;
}
```

2.1. Rédigez un petit paragraphe qui explique ce que fait la fonction `compute_principal_axis`. En particulier, expliquez en détails le calcul de `p` et `C`.

2.2. Compléter la fonction de calcul de la diagonalisation dans `Vec3.h`, chercher sur `https://www.gnu.org/software/gsl/doc/html/eigen.html`

2.3. Compléter les fonctions de projection `project` dans `tp.cpp` sur une droite et sur un plan.

2.4. Utiliser le résultat de l'ACP pour afficher une ellipsoide alignée sur les directions principales et mise à l'échelle en utilisant les racines des valeurs propres (touche `e`). Justifier le choix de la taille des demi-axes de l'ellipsoïde.

2.5. Calculer les variances des points projetés sur le premier axe principal et sur le premier plan principal (fonction `main` de `tp.cpp`). Quelle est la proportion d'inertie expliquée pour ces deux sous espaces ?

2.6. Activez l'affichage du repère ACP sur le modèle d'entrée (touche `b`) puis l'affichage du repère transformé sur le modèle déformé (touche `z`). Que constatez vous ? Expliquer pourquoi les axes ACP du modèle transformé ne sont pas égaux à la transformation des axes ACP du modèle d'entrée ?

3 Décomposition en Valeurs Singulières

On utilise la Décomposition en Valeurs Singulières pour calculer la transformation rigide permettant de replacer le maillage transformé sur le maillage d'entrée en (touche `t`). Pour cela, la matrice de covariance croisée des deux maillages sera calculée et la matrice de rotation déduite grâce à la SVD. Notez que nous avons une correspondance exact entre les points des deux maillages.

```
void find_transformation_SVD(std : :vector < Vec3 > const &ps,
                             std : :vector <Vec3> const &qs,
                             Mat3 &rotation,
                             Vec3 &translation ) {

    Vec3 p(0, 0, 0), q(0, 0, 0) ;
    for( unsigned int i = 0 ; i<ps.size() ; ++i ) {
        const Vec3 &pi = ps[i] ;
        const Vec3 &qi = qs[i] ;

        p += pi ;
        q += qi ;
    }
    p /= (float)ps.size() ;
    q /= (float)qs.size() ;

    Mat3 C(0, 0, 0, 0, 0, 0, 0, 0, 0) ;
    for(unsigned int i=0 ; i<ps.size() ; ++i ) {
        const Vec3 &pi = ps[i] ;
        const Vec3 &qi = qs[i] ;
        C += Mat3 : :tensor( qi - q, pi - p ) ;
    }

    //Calcul de la rotation grace à la SVD
    C.setRotation() ;

    //Retourner la rotation
    rotation = C ;
    //et la translation calculée
    translation = q - rotation * p ;
}
```

3.1. Rédigez un petit paragraphe qui explique ce que fait la fonction `find_transformation_SVD`. En particulier, expliquez en détails le calcul de `rotation`.

3.2. Compléter la fonction de calcul de la SVD dans `Vec3.h`, chercher sur `https://www.gnu.org/software/gsl/doc/html/eigen.html`.

3.3. Illustrer le résultat obtenu par des captures d'écran.