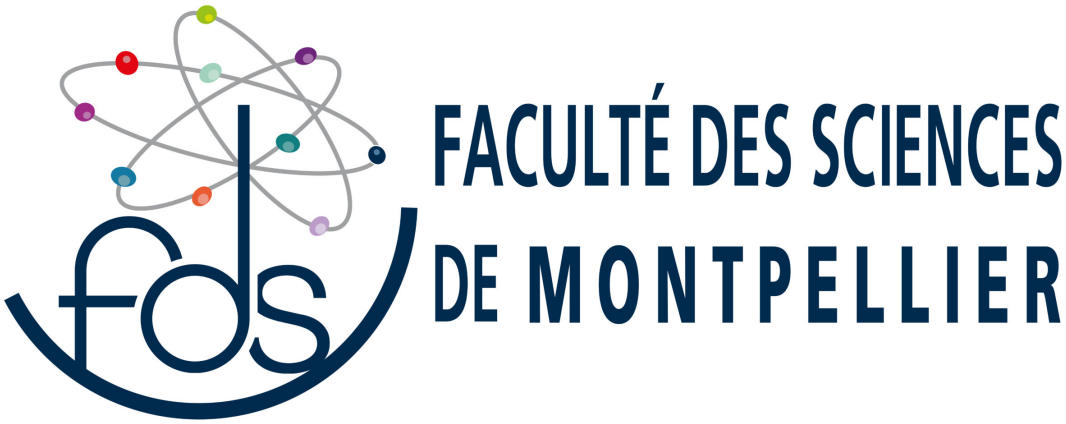
Université de Montpellier

M1 - Algebre - TP noté
Compte rendu du 17 octobre 2022

Etudiant :
Guillaume Bataille

Prof :
Annie Chateau

Année 2022-2023



Sommaire

1. Transformation

1.1 Différentes transformations via update_transformed()

```
1  Vec3 scale(1, 1, 3); //Mise à l'échelle non uniforme
```

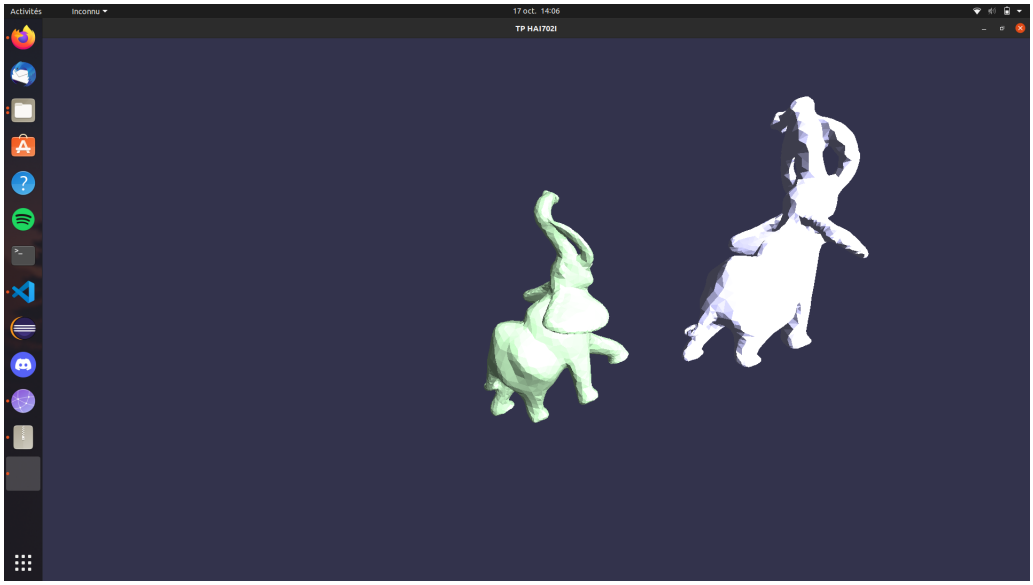


FIGURE 1.1 – Les normales des triangles ne sont pas conservé et ne sont plus normées (d'ou la sur exposition)

1.2 Idée pour résoudre le soucis des normales

Afin d'avoir des normales cohérentes, il faut appliquer aussi une matrice de transformation B sur les normales. Comme on applique la matrice de tranformation A sur les sommets du mesh, alors la transformation B doit appliquer A sur le vecteur orthogonal au plan Triangle construit a partir des points. En composant le fait que l'ancienne normale était bien orthogonale au plan (<v,nk> = 0) on peut avoir <Mv,Bnk> = 0 avec la transposée de l'inverse de la matrice de transformation.

1.3 Fix les normales dans la struct transform

```
1      //Constructor, by default Identity and no translation
2      Transform(Mat3 i_transformation = Mat3::Identity(), Vec3 i_translation = Vec3(0., 0., 0.))
```

```
3               : m_transformation(i_transformation), m_translation(i_translation) {
4           // Question 1.3: TODO, modifier la ligne suivante pour que transformation m_vector_transformation soit l
5           // m_vector_transformation = i_transformation;
6           //NEW 1.3
7            m_vector_transformation = Mat3::inverse(i_transformation).getTranspose();
8       }
```

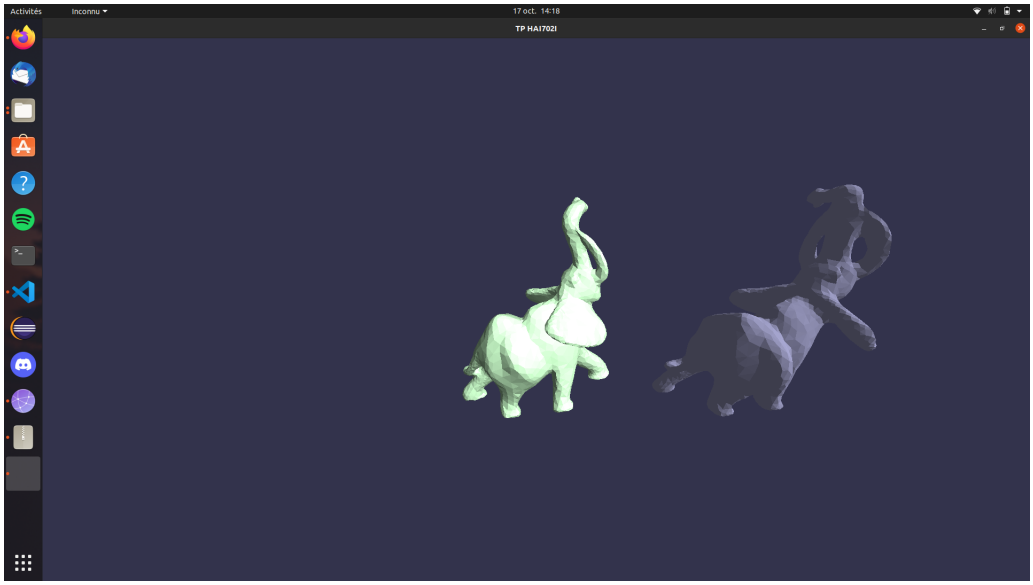


FIGURE 1.2 – Les normales ont l'air cohérentes, mais elles n'ont pas l'air normalisés !

Les normales n'ayant pas l'air normalisés, on a pas <Bnk,Bnk> = 1 car comme ||Bnk|| =/= 1 alors ||Bnk||*||Bnk|| =/= 1.

1.4 Normalize des normales transformées

```
1
2       //Transformation appliquer à un vecteur normalisé : example une normale
3       Vec3 apply_to_normalized_vector(Vec3 const &k_vector) {
4           Vec3 result = m_vector_transformation * k_vector;
5           //Question 1.4: TODO, compléter
6           //NEW 1.4
7           result.normalize();
8           return result;
9       }
10
```

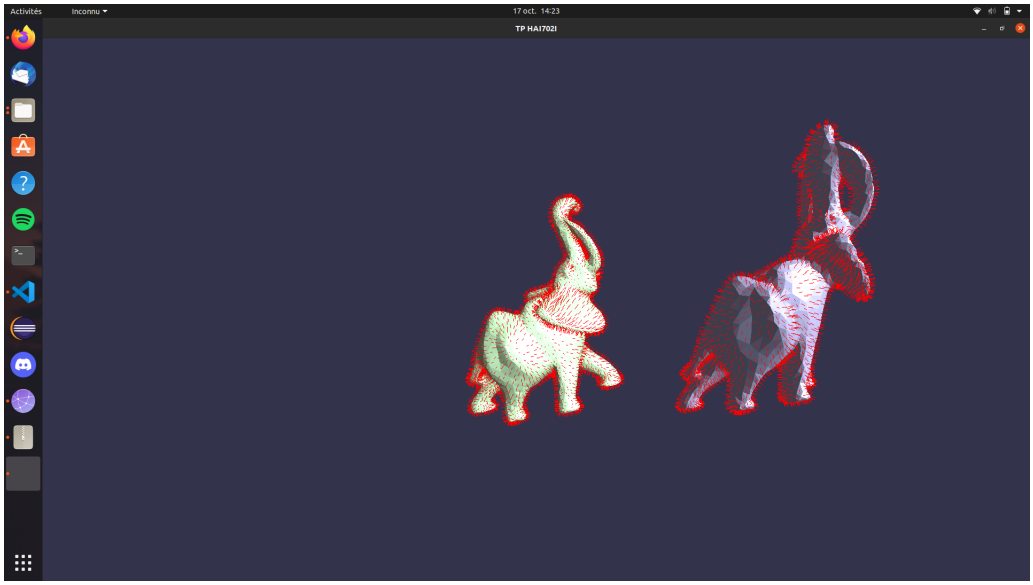


FIGURE 1.3 – Les normales ont l'air correctes et elles sont cette fois-ci normalisées

2. Analyse en composantes principales

2.1 Compute_principal_axis

Compute_principal_axis est une fonction qui semble créer le repère orthonomé principal s'alignant sur les valeurs passés en paramètres (des valeurs propres) en fonction d'une liste de sommets (mesh.vertices).

p est créé en fesant la moyenne de tout les Vec3 (vertice) de ps (du mesh) ce qui va servir de centre.

C est calculé en fesant la même chose que p, la moyenne de tout vertice, mais en utilisant un tenseur et la valeur de p pour passer d'un vec3 a un mat3. C servira donc à determiner les axes du repères. Enfin, C est diagonalisé.

2.2 Fonction diagonalisation dans Vec3.h

```
      //Question 2.2: TODO, trouver la fonction gsl_eigen pour faire le calcul de la décomposition
        //NEW 2.2 voir aide du 2.2 dans le sujet
        // gsl_eigen_..
        gsl_eigen_symmv(covariance_matrix,gsl_eigenvalues, gsl_eigenvectors, workspace);

      //Question 2.2: TODO, trouver la fonction gsl_eigen pour
        // ordonner des vecteurs et valeurs propres par ordre décroissant (plus grande valeur propre en premier)
        // gsl_eigen_...
        //NEW 2.2 voir aide du 2.2 dans le sujet
        gsl_eigen_symmv_sort(gsl_eigenvalues, gsl_eigenvectors, GSL_EIGEN_SORT_VAL_DESC);

```

Maintenant, on a comme racine propre dans le terminal : Racine des valeurs propres **0.287739 0.148409 0.106629**.

2.3 Compléter la fonction projet()

```
//Calcul de la projection d'un point sur un plan
Vec3 project(Vec3 const &input_point, Plane const &i_plane) {
    Vec3 result = input_point;
    //Question 2.3: TODO, projeter input_point sur le plan i_plane
    //result= input_point _ Vec::cross((inp.))
    result = input_point-Vec3::dot((input_point-i_plane.point),i_plane.normal)*i_plane.normal;
    return result;
}
```

```cpp

// Calcul de la projection d'un point sur une droite (définie par un vecteur et un point)
Vec3 project(Vec3 const &input_point, Vec3 const &i_origin, Vec3 const &i_axis) {
    Vec3 result = input_point;
    //Question 2.3: TODO, projeter input_point sur l'axe
    result = i_origin+Vec3::dot((input_point-i_origin),i_axis)*(i_axis/i_axis.squareLength());
    return result;
}
```

2.4 Affichage de l'ellispoide

Maintenant qu'on à les valeurs propres de calculées, nous pouvons utiliser les composantes principales(compute_principal_axis) afin d'avoir le centre et le rayon de notre ellipsoide affiché lorsque l'on appuie sur e.

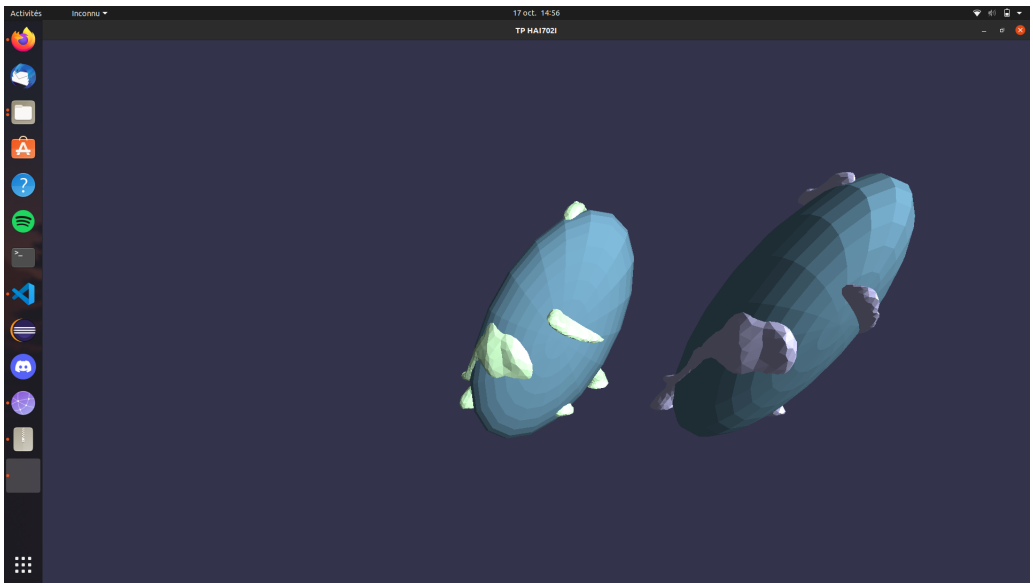


FIGURE 2.1 – Affichage de l'ellipsoide, le choix de 2 pour la taille des demi axes me semble correspondre au valeurs propres obtenues. Cela permet de bien englober le mesh.

2.5 Calcul de la variance

```cpp

    // Calcul des variances sur les sous espaces principaux:
    Vec3 variance(0., 0., 0.);
    for (unsigned int i = 0; i < 3; i++) {
        //Projection des points sur l'axe courant (utiliser le vecteur normaliser pour chaque axe)
        std::vector <Vec3> projection_on_basis;
        project(mesh.vertices, projection_on_basis, basis.origin(), basis.normalized_axis(i));
```

```
8
9           // Question 2.5: TODO Compléter
10          // NEW 2.5
11
12          for (size_t j = 0; j< projection_on_basis.size(); j++){ // Pour toute projection
13          // variance[i] =...
14          variance[i]+= pow((centroid[i]- projection_on_basis[j][i]),2); // On elève au carré la distance
15          // entre le centre et la proj
16      }
17      // Puis on normalise
18      variance[i] /= projection_on_basis.size();
19      }
```

Voici les variances obtenues : **0.130336 0.0562597 0.0972845**.

La somme de ses variances nous donne une inertie de : **0.2838802**.

2.6 Affichage des repères centrées

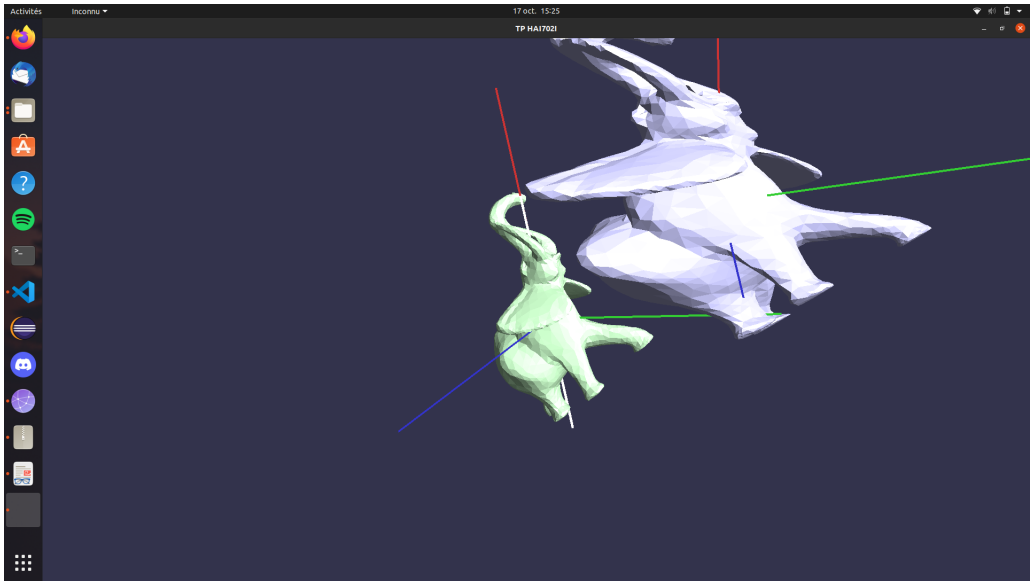


FIGURE 2.2 – Affichage des guizmo/repère centré des deux maillages

En effet, nous pouvont constater une différence entre les deux repères. Cela s'explique par le calcul des valeurs ACP (valeurs propres et valences) qui varie en fonction des positions de tout les points (de la moyenne de ses points). Comme ses points diffère entre le maillage de base et le transformé, il y a une divergence dans le calcul de ses axes.

3. Décomposition en valeur singulières

3.1 find_transformation_SVD()

find_transform_SVD est une fonction qui, avec la liste des points du maillage non transformé et la liste des points du maillage transformé, va en déduire la rotation et la translation qu'il y a eu entre les deux (afin eventuellement de confondre le centre (0,0,0) de leurs repères respectifs).
L'ensemble des sommets est toujours extrapolé pour le centre de chacun des repères (p et q).
C est aussi toujours calculé a partir d'un tenseur mais cette fois-ci, c'est les postion du maillage p (p[i]) et celle du maillage q (q[i]) qui paramètre la création de celui ci.
Cela est fait de sorte à ce qu'il est généré en fonction de la distance entre le point q courant et la moyenne de tout les points q calculé au dessus (la même pour p).

Nous nous retrouvons donc avec C, une matrice contenant(entre guillemets) la moyenne des positions de p et q et avec l'appel à C.setRotation(), on fait appel à la SVD qui définit C comme resultante d'une certaine rotation, c'est celle que l'on cherche et qu'on stocke dans rotation.

3.2 Fonction de calcul de la SVD

```
1        //Question 3.2: TODO, trouver la fonction gsl_eigen pour faire le calcul de la décomposition SVD
2        // gsl_linalg_...
3        // NEW 3.2 voir aide 3.2 en ligne
4        gsl_linalg_SV_decomp(u,v,s,workspace);
```

3.3 Visuel resultant des calculs de SVD

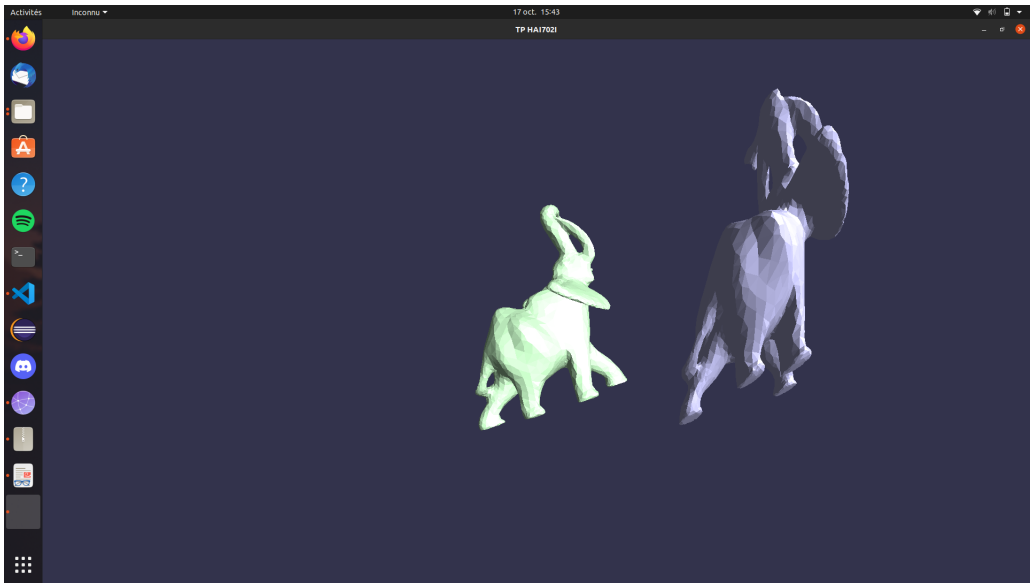


FIGURE 3.1 – Visuel comparatif avant l'appel au SVD

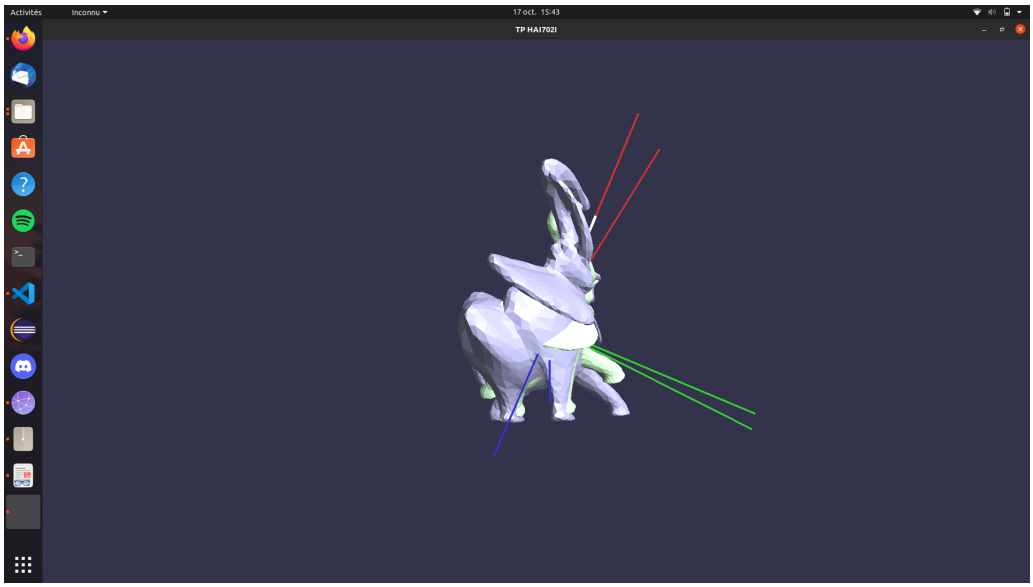


FIGURE 3.2 – Le mesh transformé est "rapatrié" avec le même centre de repère et la même "orientation" que celui de base. J'ai affiché les guizmos pour bien se rendre compte que, malgrès cela, les repères respectifs sont toujours bien distincts