TP2.1)

Exercice 1)

1) On crée une copie de l'exercice 1 du tp1 afin d'avoir les fonctions : cross_product, dot_product, vector_length et vector_normalize.
2) On doit s'assurer que vector_normalize évite la division par 0, pour ce faire on rajoute une condition dans cette fonction :

```python
def vector_normalize(vector):
    """
    Retourne la version normalisée de vector
    """
    length = vector_length(vector)
    if length == 0:
        return Vector3(0,0,0)
    return Vector3(
                   vector.x / length,
                   vector.y / length,
                   vector.z / length
                   )
```

On retourne le vecteur nul dans le cas où la norme du vecteur est nulle.

3) Dans cette question, on doit développer la matrice de mise à l'échelle autour d'un axe arbitraire, pour ce faire, on utilise cette forme de matrice :

$$\mathbf{S}(\hat{\mathbf{n}}, k) = \begin{bmatrix} \mathbf{p}' & \mathbf{q}' & \mathbf{r}' \end{bmatrix} = \begin{bmatrix} 1+(k-1)n_x^2 & (k-1)n_x n_y & (k-1)n_x n_z \\ (k-1)n_x n_y & 1+(k-1)n_y^2 & (k-1)n_y n_z \\ (k-1)n_x n_z & (k-1)n_y n_z & 1+(k-1)n_z^2 \end{bmatrix}$$

Voici la fonction développée à cet effet :

```
def scaling_matrix(axis, k):
    """Génère une matrice de mise à l'échelle le long d'un axe arbitraire."""

    n = vector_normalize(axis)
    return np.array([
        [1 + (k-1) * n.x*n.x, (k-1)*n.x*n.y, (k-1)*n.x*n.z],
        [(k-1)*n.x*n.y, 1 + (k-1)*n.y*n.y, (k-1)*n.y*n.z],
        [(k-1)*n.x*n.z, (k-1)*n.y*n.z, 1 + (k-1)*n.z*n.z]
    ])
```

4) Dans cette question, on doit faire une matrice de rotation autour d'un axe arbitraire. Pour ce faire, on utilise cette forme de matrice :

$$= \begin{bmatrix} n_x^2(1-\cos\theta)+\cos\theta & n_x n_y(1-\cos\theta)-n_z\sin\theta & n_x n_z(1-\cos\theta)+n_y\sin\theta \\ n_x n_y(1-\cos\theta)+n_z\sin\theta & n_y^2(1-\cos\theta)+\cos\theta & n_y n_z(1-\cos\theta)-n_x\sin\theta \\ n_x n_z(1-\cos\theta)-n_y\sin\theta & n_y n_z(1-\cos\theta)+n_x\sin\theta & n_z^2(1-\cos\theta)+\cos\theta \end{bmatrix}$$

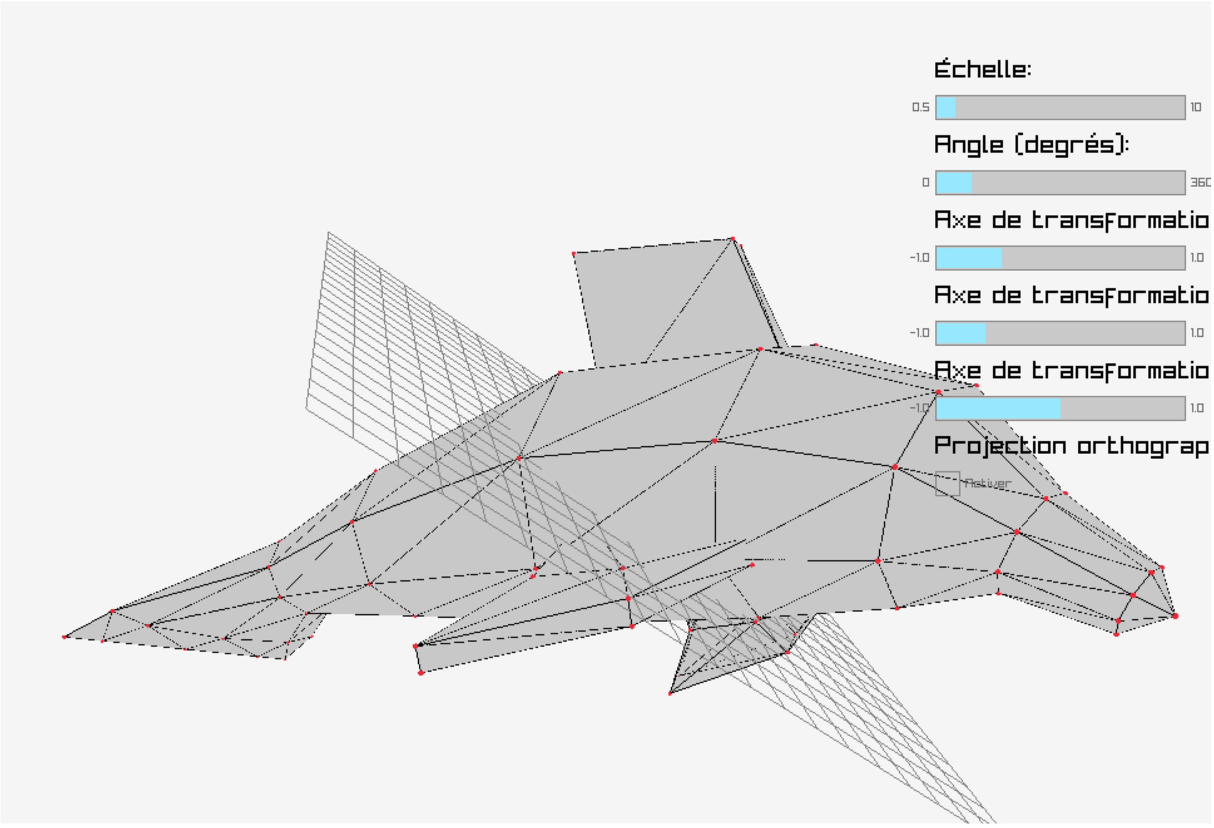
Voici la fonction développée :

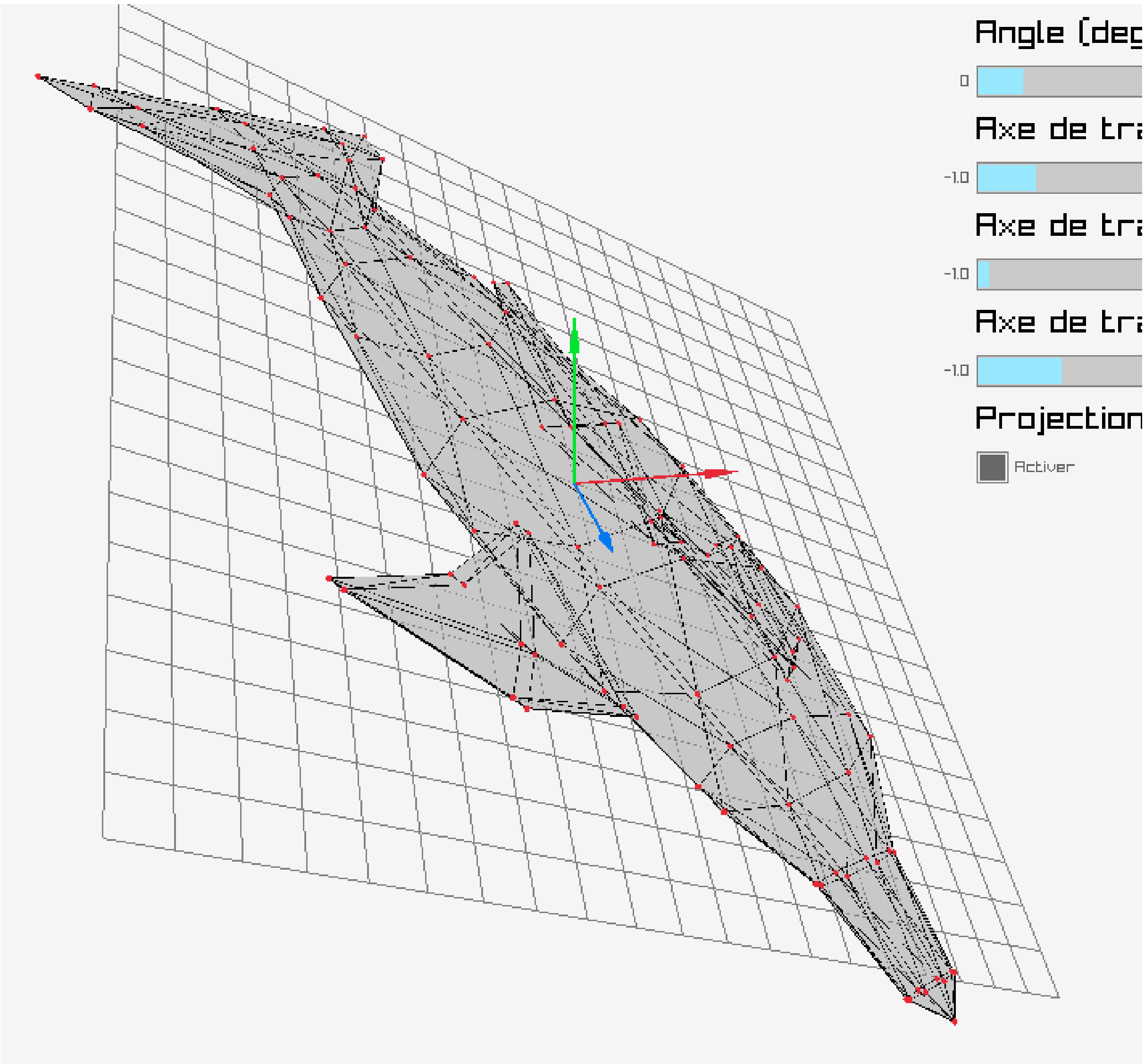
```
def rotation_matrix(axis, theta):
    """Génère une matrice de rotation autour d'un axe arbitraire."""

    n = vector_normalize(axis)
    cos_theta = np.cos(theta)
    sin_theta = np.sin(theta)
    return np.array([
        [n.x*n.x*(1-cos_theta) + cos_theta, n.x*n.y*(1-cos_theta)-n.z*sin_theta, n.x*n.z*(1-cos_theta) +
n.y*sin_theta],
        [n.x*n.y*(1-cos_theta) + n.z*sin_theta, n.y*n.y*(1-cos_theta) + cos_theta, n.y*n.z*(1-cos_theta)
- n.x*sin_theta],
        [n.x*n.z*(1-cos_theta) - n.y*sin_theta, n.y*n.z*(1-cos_theta) + n.x*sin_theta, n.z*n.z*(1-
cos_theta) + cos_theta]
    ])
```

5) Enfin, on doit développer la fonction orthographic_projection_matrix, qui n'est autre que la matrice de scaling mais avec un facteur k=0
La fonction :

```python
def orthographic_projection_matrix(axis):
    """Génère une matrice de projection orthographique pour la projection sur un plan normal à un axe
donné."""

    n = vector_normalize(axis)
    return np.array([
        [1-n.x*n.x, -1*n.x*n.y, -1*n.x*n.z],
        [-1*n.x*n.y, 1-n.y*n.y, -1*n.y*n.z],
        [-1*n.x*n.z, -1*n.y*n.z, 1-n.z*n.z]
    ])
```

Screenshot :





Exercice 2)

1)

- Explication des opérations : En python, l'opérateur @ permet de multiplier des matrices entre elles, ce qui a pour effet ici d'enchainer les transformations les une après les autres.
- Ordre des transformations : L'ordre des transformations à un impact sur le résultat final car les transformations en 3D ne sont pas commutatives.

Considérons l'exemple suivant : Mise à l'échelle > rotation > projection, cette suite de transformation conserve la géométrie 3D avec mise à l'échelle puis rotation avant d'applatir la forme.

En revanche, si on considère ce cas : Projection > mise à l'échelle > rotation, le résultat sera une version applatie de l'objet ce qui altère complétement la géométrie
3D de l'objet.

2) Parce que les sommets d'origines représentent les coordonnées initiales d'un objet 3D avant qu'une transformation quelconque ne lui soit appliquée. Ces sommets servent donc de référence.

3) Afin de conserver la géométrie ainsi que l'orientation du plan. Ces deux produits vectoriels sont utilisés afin de définir deux vecteurs perpendiculaires à un vecteur normal donné

Exercice 3)

1) Dans cette question, on souhaite ajouter une transformation de cisaillement. Dans un premier temps, il faut créer les paramètres de l'opération de cisaillement, on décide d'implémenter un cisaillement sur le plan xy, donc de la matrice suivante :

$$\mathbf{H}_{xy}(s,t) = \begin{bmatrix} 1 & 0 & s \\ 0 & 1 & t \\ 0 & 0 & 1 \end{bmatrix}$$

Pour les pointeurs :

```
shear_s_ptr = pr.ffi.new('float *', 0.0)
shear_t_ptr = pr.ffi.new('float *', 0.0)
```

Création de la matrice de shearing au sein de la boucle principale de rendu :

```
shearing_mat = shearing_matrix_xy(shear_s_ptr[0], shear_t_ptr[0])
```

Ajout de la matrice a apply_transformations, avant la projection :

```
apply_transformations(mesh, rotation_mat, scaling_mat, projection_mat, shearing_mat)
```

Enfin, la matrice :

```
def shearing_matrix_xy(s, t):
    return np.array([
        [1,0,s],
        [0,1,t],
        [0,0,1]
    ])
```

2) Explication :
   La projection orthographique ne tient pas compte la mise à l'échelle sur un axe arbitraire car les points d'un objet sont projetés de manière parallèle. La vérification se trouve dans le fichier TP2.1/exo3.txt
   Screenshot :