TP 2.2)

Exercice 1)

2) Dans cette question, on souhaite maintenant implémenter les matrices de transformations développées jusqu'à présent mais en matrices 4x4 homogènes.

Voici les trois fonctions développées :

```python
def rotation_matrix_homogeneous(axis, theta):
    """Génère une matrice homogène de rotation autour d'un axe arbitraire (4x4)."""
    axis = vector_normalize(axis)
    nx, ny, nz = axis.x, axis.y, axis.z
    cos_theta = np.cos(theta)
    sin_theta = np.sin(theta)


    return np.array([
        [nx*nx*(1-cos_theta) + cos_theta, nx*ny*(1-cos_theta)-nz*sin_theta, nx*nz*(1-cos_theta) + ny*sin_theta, 0],
        [nx*ny*(1-cos_theta) + nz*sin_theta, ny*ny*(1-cos_theta) + cos_theta, ny*nz*(1-cos_theta) - nx*sin_theta, 0],
        [nx*nz*(1-cos_theta) - ny*sin_theta, ny*nz*(1-cos_theta) + nx*sin_theta, nz*nz*(1-cos_theta) + cos_theta, 0],
        [0,0,0,1]
    ])
```

```python
def scaling_matrix_homogeneous(axis, k):
    """Génère une matrice homogène de mise à l'échelle le long d'un axe arbitraire (4x4)."""
    axis = vector_normalize(axis)
    nx, ny, nz = axis.x, axis.y, axis.z
    # TODO :
    return np.array([
        [1 + (k-1)*nx*nx, (k-1)*nx*ny, (k-1)*nx*nz, 0],
        [(k-1)*nx*ny, 1 + (k-1)*ny*ny, (k-1)*ny*nz, 0],
        [(k-1)*nx*nz, (k-1)*ny*nz, 1 + (k-1)*nz*nz, 0],
        [0,0,0,1]
    ])
```

```python
def orthographic_projection_matrix_homogeneous(axis):
    """Génère une matrice homogène de projection orthographique sur un plan normal à un axe donné (4x4)."""
    axis = vector_normalize(axis)
    nx, ny, nz = axis.x, axis.y, axis.z
    # TODO :
    return np.array([
        [1-nx*nx, -1*nx*ny, -1*nx*nz, 0],
        [-1*nx*ny, 1-ny*ny, -1*ny*nz, 0],
        [-1*nx*nz, -1*ny*nz, 1-nz*nz, 0],
        [0,0,0,1]
    ])
```

La coordonnée homogène permet d'ajouter de la profondeur dans les futurs exercices. On prend tout simplement la version 3x3 et on ajoute la composante 1 en coordonnée (4,4) et les supplémentaires à 0

3) On souhaite maintenant implémenter translation_matrix, on utilise la matrice suivante :

$$\begin{bmatrix} 1 & 0 & 0 & \Delta x \\ 0 & 1 & 0 & \Delta y \\ 0 & 0 & 1 & \Delta z \\ 0 & 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} x \\ y \\ z \\ 1 \end{bmatrix} = \begin{bmatrix} x + \Delta x \\ y + \Delta y \\ z + \Delta z \\ 1 \end{bmatrix}$$

Ce qui donne la fonction suivante :

```python
def translation_matrix(tx, ty, tz):
    """Génère une matrice homogène de translation (4x4)."""
    return np.array([
        [1,0,0,tx],
        [0,1,0,ty],
        [0,0,1,tz],
        [0,0,0,1]
    ])
```

4) On souhaite implémenter une matrice de projection en perspective, on utilise :

$$\begin{bmatrix} x \\ y \\ z \\ z/d \end{bmatrix}$$

Ce qui donne la fonction suivante :

```python
def perspective_projection_matrix( d):
    """Génère une matrice homogène de projection en perspective avec une distance focale d."""
    # TODO :
    return np.array([
        [1,0,0,0],
        [0,1,0,0],
        [0,0,1,0],
        [0,0,1/d,0]
    ])
```

5) On ajoute la coordonnée homogène pour permettre les translations ainsi que les projections. Ensuite, on applique les transformations en utilisant des matrices 4x4 homogènes. Enfin, on retourne aux coordonnées cartésiennes en divisant par w après la projection.

Exercice 2)

1) Voir code

   Résumé des modifications :

```python
central_translation = translation_matrix(translate_x_ptr[0], translate_y_ptr[0], translate_z_ptr[0])
rotation_axis = Vector3(axis_x_ptr[0], axis_y_ptr[0], axis_z_ptr[0])
central_rotation = rotation_matrix_homogeneous(rotation_axis, np.radians(rotation_angle_ptr[0]))
projection_matrix = np.eye(4)
```

```python
central_transform = central_translation @ central_rotation @ projection_matrix
```

```python
orbit_x = np.cos(angle) * distance_from_orbit  * orbit_radius_ptr[0]
orbit_y = np.sin(orbit['inclination']) * distance_from_orbit * orbit_radius_ptr[0]
orbit_z = np.sin(angle) * distance_from_orbit * orbit_radius_ptr[0]
```

Explication du premier TODO :

On crée un dictionnaire, avec des clés/valeurs.

transform: Matrice identité 4x4 pour stocker la transformation homogène.

angle_offset : Angle initial aléatoire de l'objet sur son orbit.

inclinaison: inclinaison de l'orbite, basculement du plan orbital.

rotation_axis : Axe de rotation aléatoire autour duquel ça tourne.

clockwise: sens de rotation aléatoire (true = horaire, false = anti-horaire)

distance_from_orbit : distance des cubes par rapport au cube central

scale : multiplicateur de la taille des cubes

2) Ordre des transformations :

```python
central_transform = central_translation @ central_rotation @ projection_matrix
```

On choisit cet ordre (hors projection implémenté dans l'exercice suivant), car si la rotation est appliquée avant la translation, le cube tournerait autour de l'origine au lieu de tourner sur lui-même. Evidemment, la projection arrive à la fin car sinon, aucune des transformations antérieure n'aurait d'effet.

3) Implémentation de scale et de distance from orbit dans le dictionnaire :

```
"scale": np.random.uniform(0.5,2),
"distance_from_orbit": np.random.uniform(1,10)
```

En ce qui concerne l'angle, on doit l'appliqué ici :

```
orbit_x = np.cos(angle) * distance_from_orbit  * orbit_radius_ptr[0]
orbit_y = np.sin(orbit['inclination']) * distance_from_orbit * orbit_radius_ptr[0]
orbit_z = np.sin(angle) * distance_from_orbit * orbit_radius_ptr[0]
```

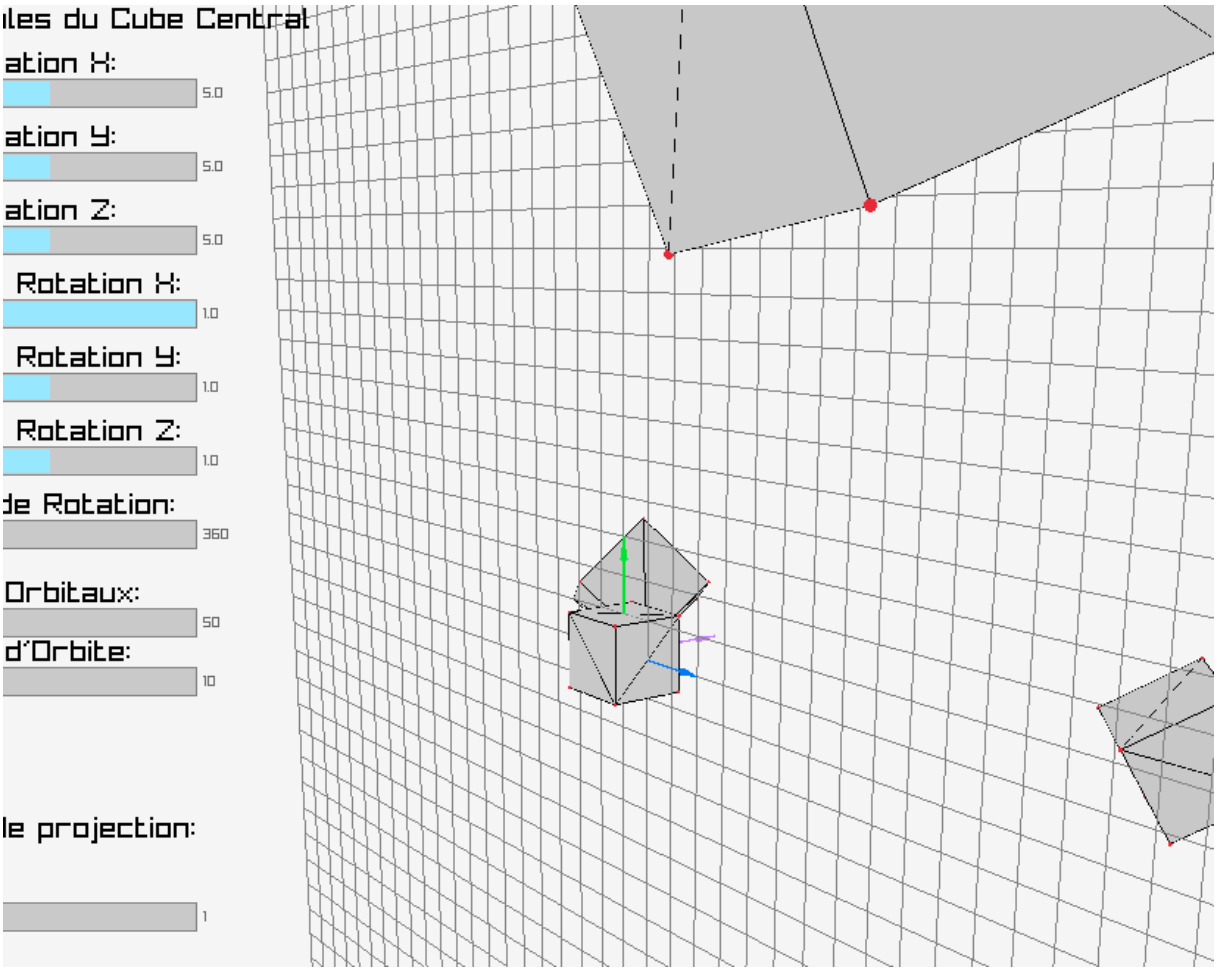
Pour le scaling, il faut réaliser une matrice de mise à l'échelle uniforme :

```
scale_matrix = np.array([
    [scale, 0, 0, 0],
    [0, scale, 0, 0],
    [0,0,scale,0],
    [0,0,0,1]
])


orbit_transform = central_transform @ orbit_translation @ orbit_rotation @ scale_matrix @ projection_matrix
```

Ci-dessous, l'ordre des transformations, on translate, puis on fait tourner, et enfin on met à l'échelle, pour éventuellement, projeté sur le plan.

Screenshot :



4) Pour ajouter le plan, on trace un plan :

```
draw_plane(axis, 20)
```

axis correspond aux axes de transformation.

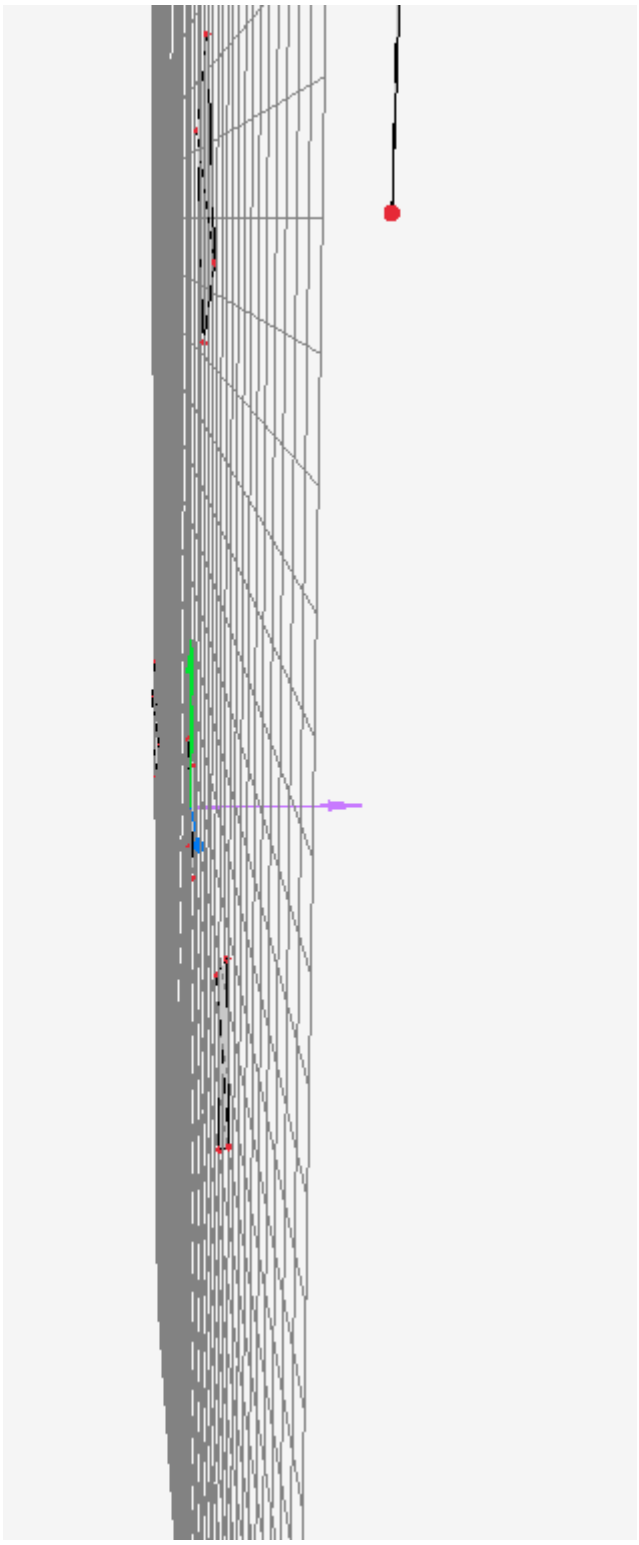
On ajoute aussi les axes de transformation et de référence :

draw_coordinate_axes(Vector3(0, 0, 0), scale=3)

draw_transformation_axis(Vector3(0, 0, 0), axis, scale=3)

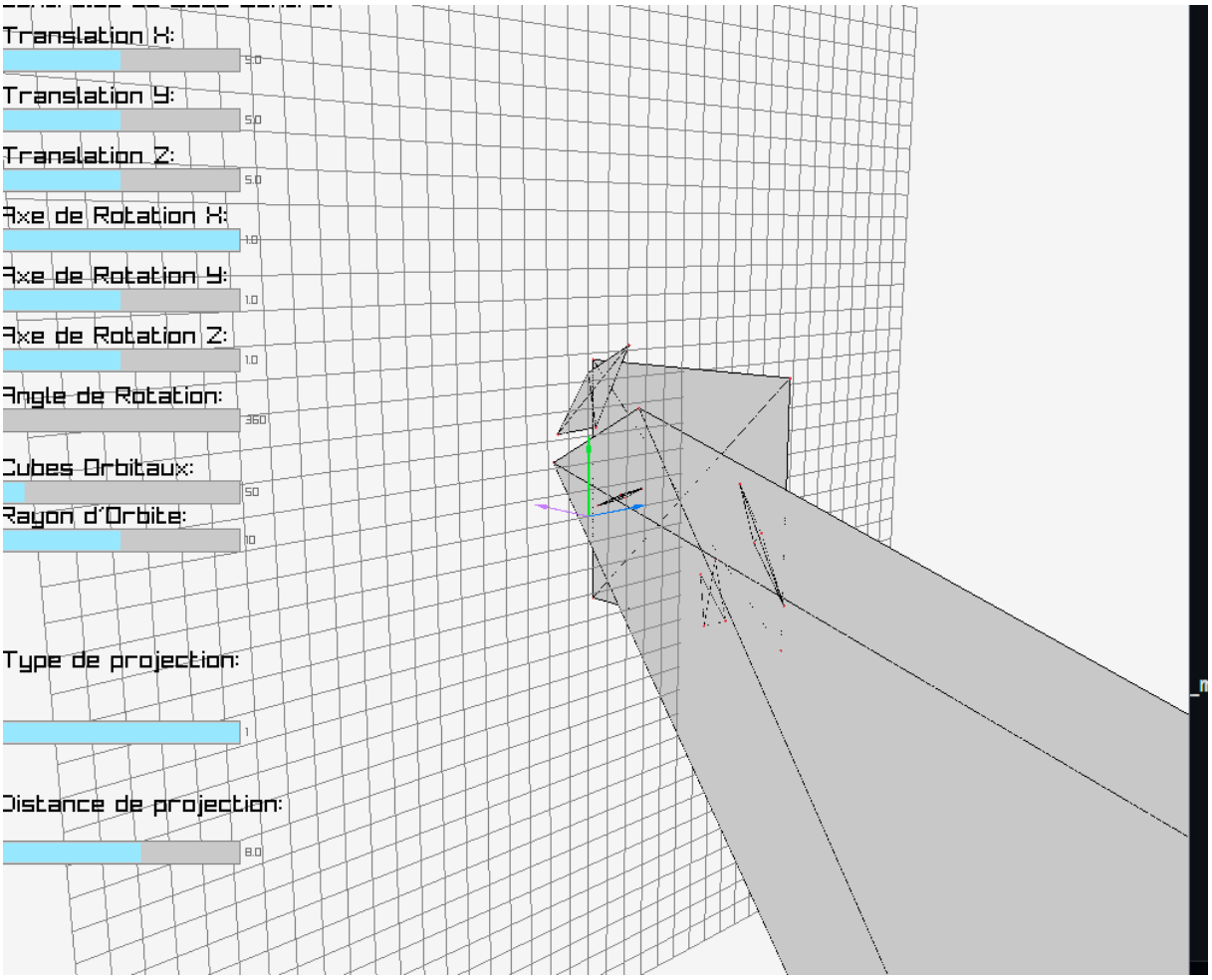
Enfin, il faut réaliser la transformation orthographique. Donc on initialise un ptr pour stocker la valeur, on fait une barre jaugée qu'on peut faire glisser pour varier entre projection orthographique et perspective. On applique la projection orthographique, et voila ce que cela nous donne :

Screenshot :



On constate que les points sont bien confondus avec le plan

5) Pour la projection en perspective, voici le screenshot :



Exercice 3)

1) Pour paramétrer le nombre de cubes par tour, il faut paramétrer num_turns_ptr dans la GUI :

```
pr.draw_text("Cubes par tours:", 10, 580, 20, pr.BLACK)
pr.gui_slider_bar(pr.Rectangle(10, 620, 200, 20), "0.5", "5.0", num_turns_ptr, 0, 15.0)
```

2) Pour faire tourner les cubes en synchronisation :

```
time_angle = pr.get_time() + i*0.2
cube_rotation = rotation_matrix_homogeneous(Vector3(0, 1, 0), time_angle)
```

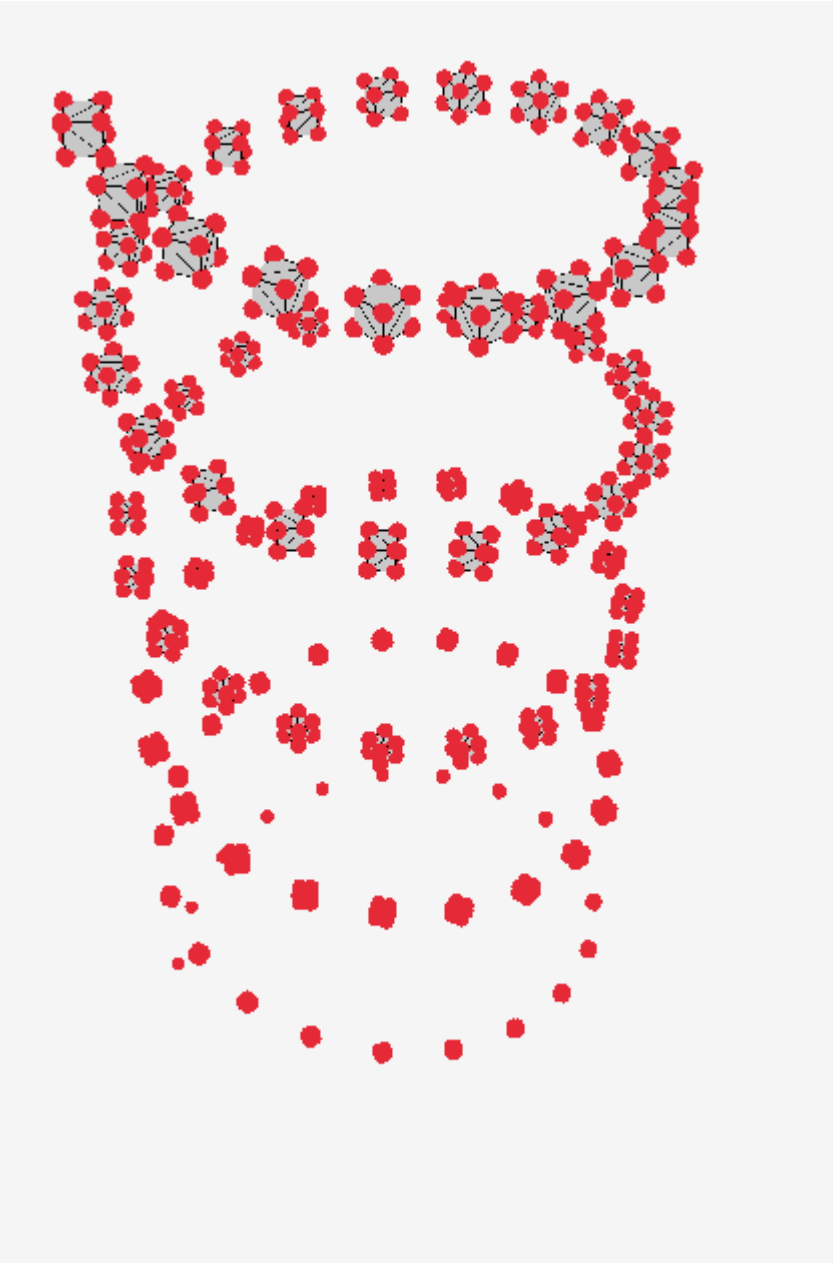
La variable i augmentant tour par tour, cela assure que chaque cube est un peu plus tourné que son précédent.

3) Idem pour le scaling :

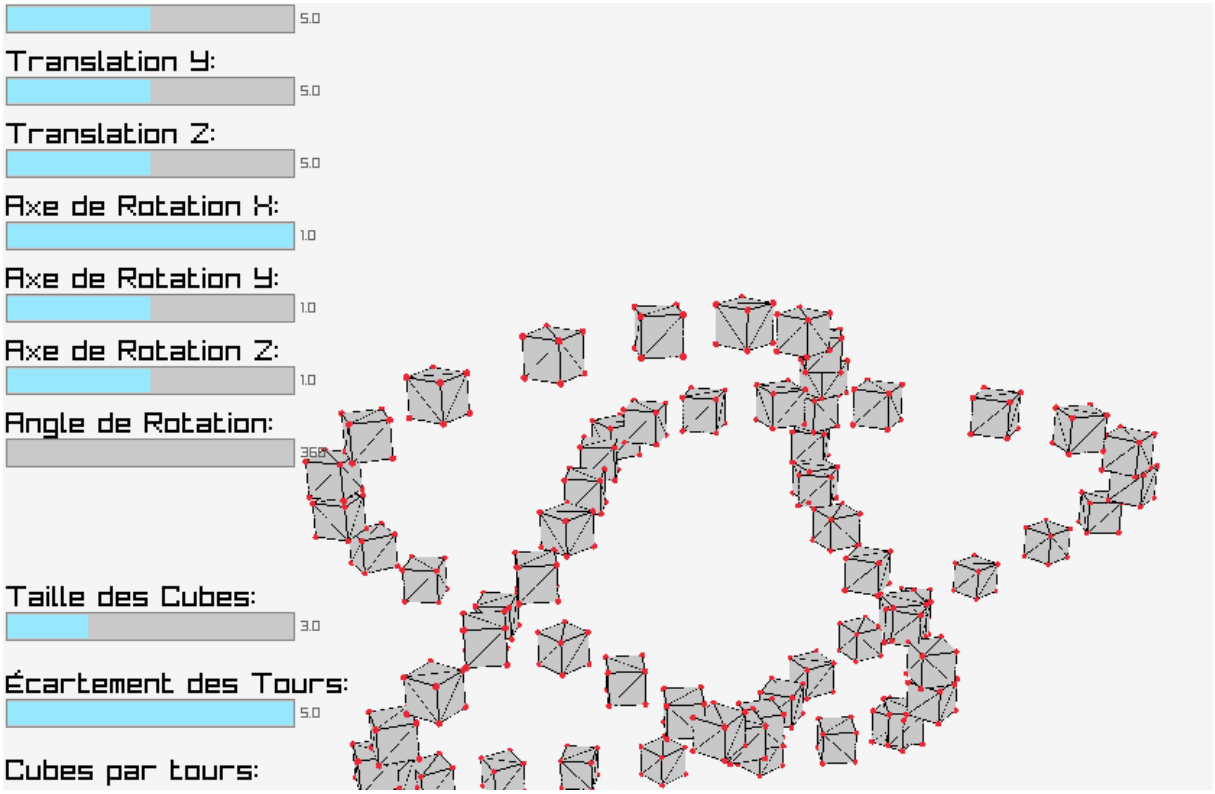
```
time_scale = 0.5 + (i / num_cubes)
cube_scaling = uniform_scaling_matrix_homogeneous(cube_scale_ptr[0] * time_scale)
```

Screenshot des autres courbes :

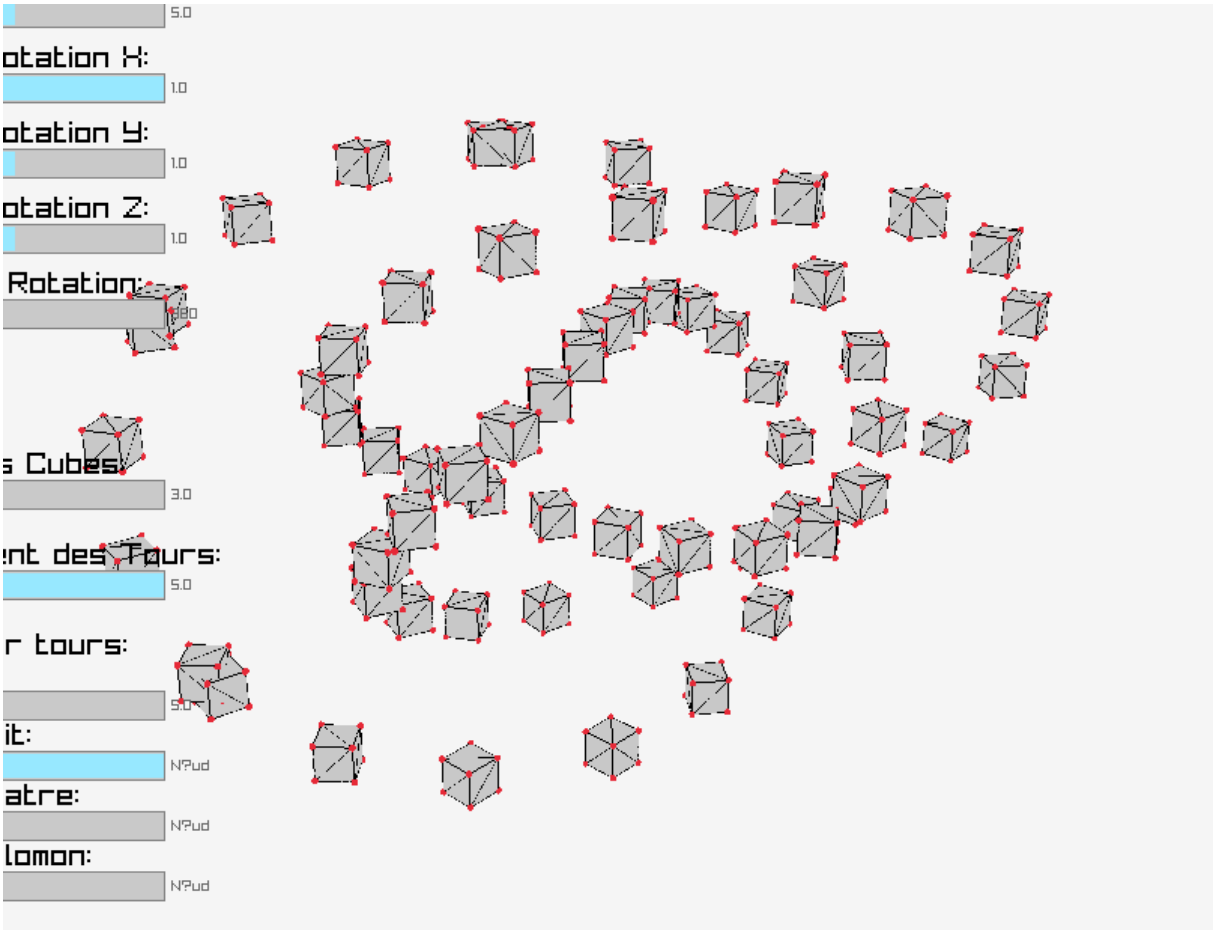
Helix :



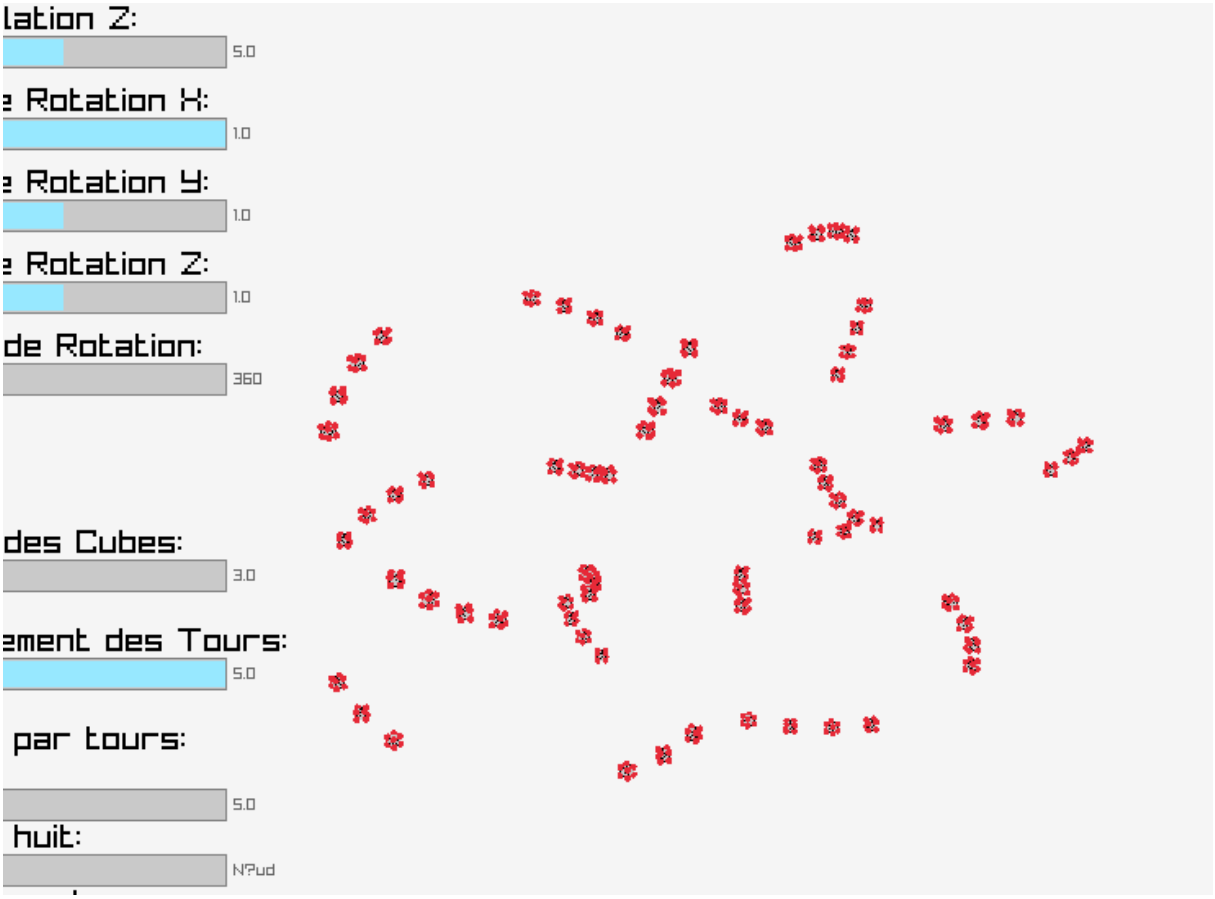
Nœud 4 :



Nœud 8 :



Nœud solomon :



Exercice 4)

1) Angles d'euler :

```python
def rotation_matrix_yaw_pitch_roll(yaw, pitch, roll):
    """Generate a rotation matrix from yaw, pitch, and roll.
    # Convert degrees to radians
    yaw = math.radians(yaw)  # Rotation around Z-axis
    pitch = math.radians(pitch)  # Rotation around Y-axis
    roll = math.radians(roll)  # Rotation around X-axis

    # Precompute sin and cos values
    cos_yaw, sin_yaw = math.cos(yaw), math.sin(yaw)
    cos_pitch, sin_pitch = math.cos(pitch), math.sin(pitch)
    cos_roll, sin_roll = math.cos(roll), math.sin(roll)

    # Rotation around Z-axis (Yaw)
    Rz = rotation_matrix_z(cos_yaw, sin_yaw)

    # Rotation around Y-axis (Pitch)
    Ry = rotation_matrix_y(cos_pitch, sin_pitch)

    # Rotation around X-axis (Roll)
    Rx = rotation_matrix_x(cos_roll, sin_roll)

    # Combine the rotations in order: Rz * Ry * Rx
    return Rz @ Ry @ Rx
```

Cette fonction crée une matrice de rotation en combinant les rotations autour des trois axes.

L'idée est de convertir les angles d'Euler (donnés en degrés) en radians, puis de créer les matrices de rotation individuelles pour chaque axe. Ensuite, on multiplie ces matrices dans le bon ordre : R=Rz×Ry×RxR = Rz @ Ry @ RxR=Rz×Ry×Rx

Cela permet d'obtenir une seule matrice qui applique toutes les rotations d'un coup.

2) Ici, on veut faire tourner un segment autour d'un point précis (ex : l'épaule ou le coude).

Problème : Les rotations se font toujours autour de l'origine (0,0,0). Solution :Déplacer le point de référence à l'origine ; Appliquer la rotation ; Revenir à la position initiale

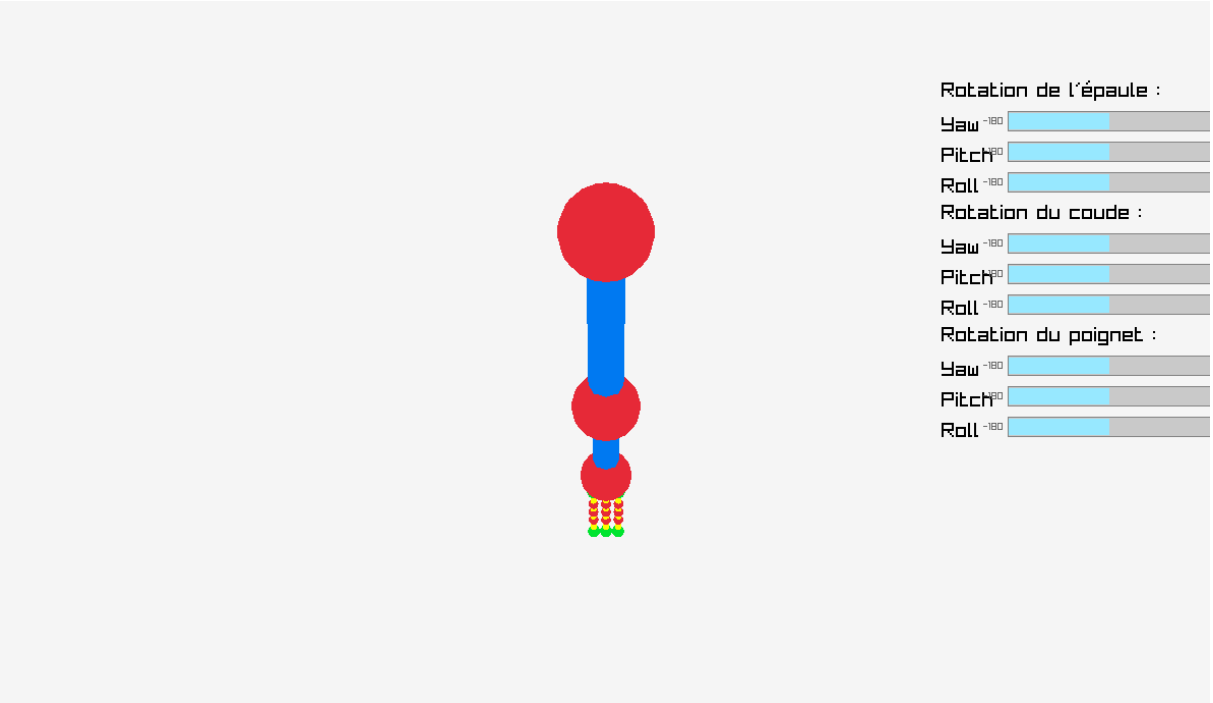
On utilise pour cela des matrices de translation qui déplacent temporairement le segment avant et après la rotation.

3) La bonne combinaison de transformations est : déplacement du point, application de la rotation, retour à la position initiale, prise en compte de la transformation du segment parent.

4) Les doigts sont initialement définis dans un repère local (par rapport au poignet). Pour qu'ils suivent les mouvements du bras, il faut appliquer la rotation du poignet à leurs positions locales.

On utilise apply_rotation pour transformer les positions locales des doigts en positions globales, en tenant compte de l'orientation du poignet.

5) Screenshot :



6) Afin de rendre cela plus réaliste, j'ai mit des phalanges sur les doigts.