TP3)

Exercice 1)

1) Implémenter la fonction compute_normal
   On utilise la méthode best-fit :

$$n_x = \sum_{i=1}^{n} (z_i + z_{i+1})(y_i - y_{i+1})$$

$$n_y = \sum_{i=1}^{n} (x_i + x_{i+1})(z_i - z_{i+1})$$

$$n_z = \sum_{i=1}^{n} (y_i + y_{i+1})(x_i - x_{i+1})$$

   Ce qui donne le code suivant :

```
def compute_normal(points):
    """
    TODO
    """
    n = Vector3(0,0,0)
    for i in range(len(points)-1):
        pi = points[i]
        pip1 = points[i+1]
        n.x += (pi.z + pip1.z) * (pi.y - pip1.y)
        n.y += (pi.x + pip1.x) * (pi.z - pip1.z)
        n.z += (pi.y + pip1.y) * (pi.x - pip1.x)

    return vector_normalize(n)
```

   On a ici un vecteur normal au plan crée par les points.

2) Créer les plans rouge et vert (estimés et référence)

```
def draw_plane(normal, point_on_plane, color, size=10):
    """
    TODO
    """
    normal = vector_normalize(normal)
    # Générer deux vecteurs orthogonaux dans le plan
    if abs(normal.x) > 1e-3 or abs(normal.z) > 1e-3:
        v1 = vector_normalize(Vector3(-normal.z, 0, normal.x))
    else:
        v1 = vector_normalize(Vector3(0, -normal.z, normal.y))
    v2 = vector_normalize(cross_product(normal, v1))

    for i in range(-size, size + 1):

        start1 = Vector3(point_on_plane.x + v1.x * -size + v2.x * i,
                         point_on_plane.y + v1.y * -size + v2.y * i,
                         point_on_plane.z + v1.z * -size + v2.z * i)
        end1 = Vector3(point_on_plane.x + v1.x * size + v2.x * i,
                       point_on_plane.y + v1.y * size + v2.y * i,
                       point_on_plane.z + v1.z * size + v2.z * i)
        start2 = Vector3(point_on_plane.x + v2.x * -size + v1.x * i,
                         point_on_plane.y + v2.y * -size + v1.y * i,
                         point_on_plane.z +v2.z * -size + v1.z * i)
        end2 = Vector3(point_on_plane.x + v2.x * size + v1.x * i,
                       point_on_plane.y + v2.y * size + v1.y * i,
                       point_on_plane.z +v2.z * size + v1.z * i)

        pr.draw_line_3d(start1, end1, color)
        pr.draw_line_3d(start2, end2, color)
```

Explication du code :

On calcul notre vecteur normal approximé. On constate que les vecteurs (-n.z, 0, n.x) et (0,-n.z,n.y) sont orthogonaux a n car produit scalaire nul. Ce code a été reprit des tp précédent. Ensuite, on réalise un produit vectoriel afin de trouver un vecteur orthogonal aux deux autres vecteurs pour former un trièdre direct (un plan 3D).

V1 et v2 agissent comme des vecteurs unitaires de la base composée par les points

Ensuite, il suffit de tracer la grille.

Pour tracer le plan de référence et le plan estimé, on a besoin d'un vecteur normal approximé et d'un vecteur normal de référence, les voici :

```
# TODO: Calculer le vecteur normal du plan à partir des points
normal = compute_normal(points)
normal_ref = vector_normalize(Vector3(1,1,1))
```

Ensuite, on a besoin de calculer le centroid du plan pour ensuite, pouvoir placer les points à partir du centroid :

```
# TODO: La normale doit être affichée au centroïde du plan
moy_x = 0
moy_y = 0
moy_z = 0
length = len(points)
for i in range(length):
    moy_x += points[i].x
    moy_y += points[i].y
    moy_z += points[i].z

moy_x /= length
moy_y /= length
moy_z /= length
centroid = Vector3(
        moy_x,
        moy_y,
        moy_z
)
```

Puis, on a besoin de deux points, l'un sur le plan estimé, et l'autre sur le plan de référence, comme suit :

```
)
point_on_plane = Vector3(centroid.x, centroid.y, centroid.z)
point_on_plane_ref = Vector3(0,0,0)

while not pr.window_should_close():
```

Enfin, il suffit de mettre bout a bout les fonctions développées jusqu'ici :

```
draw_points(points)
draw_plane(normal, point_on_plane, pr.RED)

draw_vector_3(point_on_plane, Vector3(centroid.x + normal.x * 5, centroid.y + normal.y * 5, centroid.z + normal.z * 5), pr.RED)
draw_plane(normal_ref, point_on_plane_ref, pr.GREEN)

draw_vector_3(point_on_plane_ref, Vector3(point_on_plane_ref.x + normal_ref.x * 5, point_on_plane_ref.y + normal_ref.y * 5, point_on_plane_ref.z + n
```

On place les points, on trace le plan estimé, ensuite on trace sa normale placée au centroid, et de même pour le plan de référence.

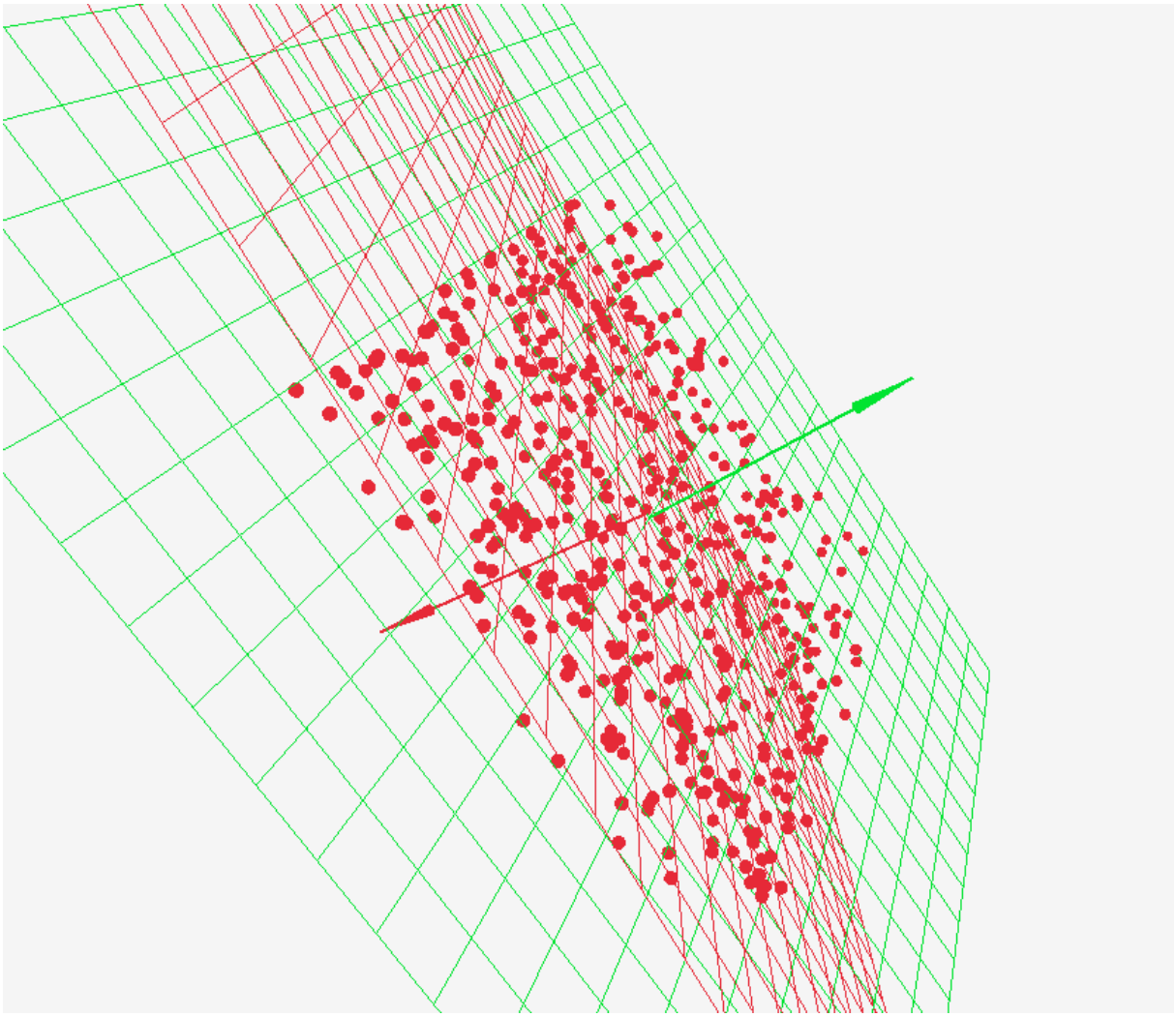
3) On souhaite permettre une déviation aléatoire du plan :

Pour ce faire, dans la métode generate_random_points_on_plane, il suffit d'ajouter ce bout de code :

```
random_deviation = np.random.uniform(-deviation, deviation)
p = Vector3(
    point.x + r1 * u.x + r2 * v.x + normal.x * random_deviation,
    point.y + r1 * u.y + r2 * v.y + normal.y * random_deviation,
    point.z + r1 * u.z + r2 * v.z + normal.z * random_deviation
)
```

Qui va crée une deviation variant de -deviation à + deviation (+/-1 par défaut), sur la normal du plan considéré

Screenshot :



On constate que ça ne colle pas parfaitement, et c'est normal car n estimé est calculé avec la méthode best-fit.

Exercice 4)

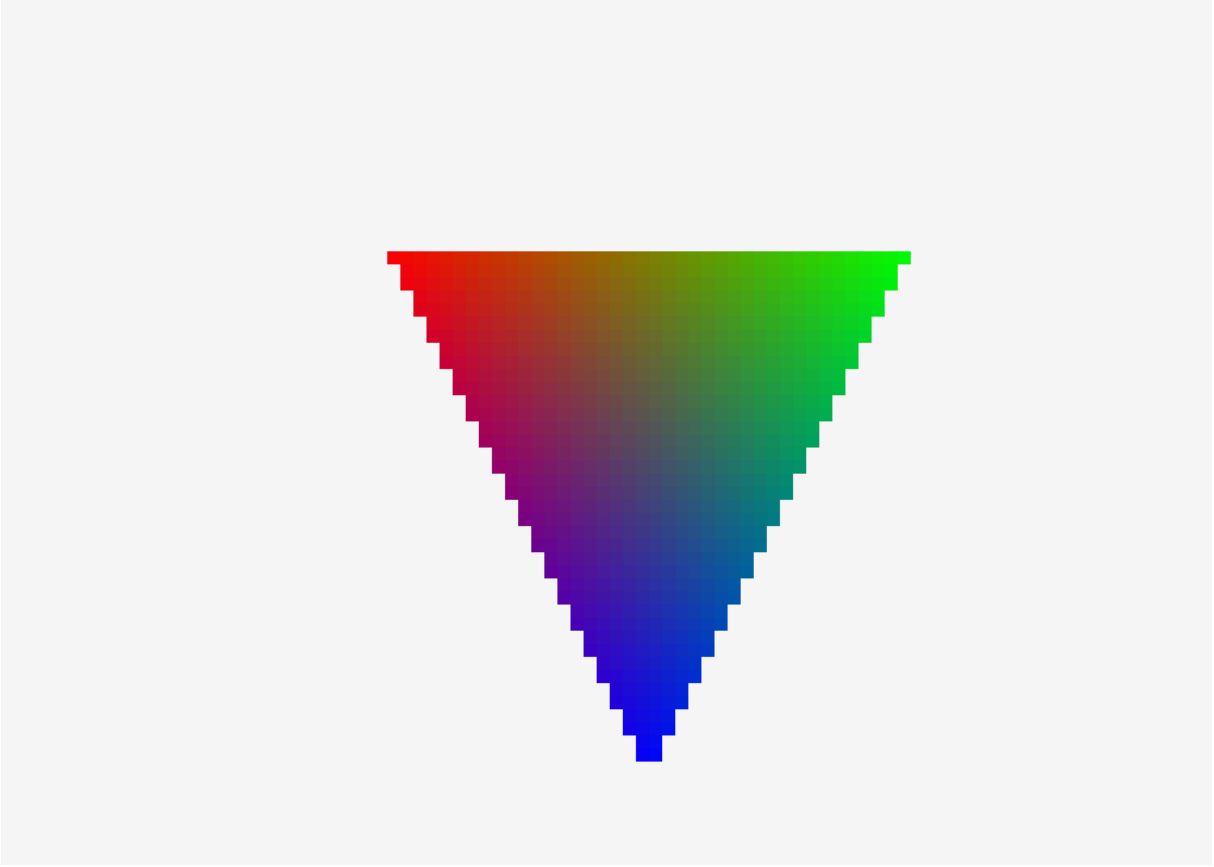
Il suffit d'utiliser la formule du cours pour calculer les coordonnées barycentriques et les retournées, comme suit :

```python
def barycentric_coordinates(p, a, b, c):
    """TODO : Calculer les coordonnées barycentriques du point p dans le triangle abc."""
    denominator = (a.y - c.y) * (b.x - c.x) + (b.y - c.y) * (c.x - a.x)
    b1 = (p.y - c.y) * (b.x - c.x) + (b.y - c.y) * (c.x - p.x)
    b1/=denominator
    b2 = (p.y - a.y) * (c.x - a.x) + (c.y - a.y) * (a.x - p.x)
    b2/=denominator
    b3 = (p.y - b.y) * (a.x - b.x) + (a.y - b.y) * (b.x - p.x)
    b3/=denominator

    u=b1
    v=b2
    w=b3


    return u, v, w
```

Screenshot :



Exercice 5)

On sait que X(M) = V – E + F

Et que : 2(1-g) = X(M) ⇔ g = (2-X(M)) / 2

D’où les programmes suivants :

En lisant la documentation trimesh, on constate que V, E et F possèdent des implémentations dans les fichiers mesh :

```python
def euler_characteristic(mesh):
    """Calcule la caractéristique d'Euler-Poincaré pour un maillage.
    V = len(mesh.vertices)
    E = len(mesh.edges_unique)
    F = len(mesh.faces)
```

Résultats pour cube.ply et earth.ply :

```
X(M) = 2
Genus = 0.0
PS C:\Users\Aman\Desktop\maths_3D\TP3> python exo5.py
RAYLIB STATIC 5.5.0.2 LOADED
X(M) = 2
Genus = 0.0
```

Dolphin.ply contient des zones de discontinuités donc il est inutilisable.

Exercice 2)

1) La fonction generate_random_points_on_plane génère des points aléatoires sur un plan défini par un point et un vecteur normal. On peut ajuster les paramètres num_points, spread, et deviation pour contrôler le nombre de points, leur dispersion et leur déviation par rapport au plan.
2) La fonction calc_pmin_pmax calcule les points minimum (Pmin) et maximum (Pmax) d'un ensemble de points 3D. Elle parcourt tous les points et détermine les valeurs minimales et maximales pour chaque axe (x, y, z).
3) La fonction draw_aabb dessine déjà la boîte englobante axiale (AABB) en utilisant les points Pmin et Pmax. Pour marquer le centroïde, on peut ajouter une fonction qui calcule et dessine le point central de l'AABB.
4) La fonction apply_transformation applique une transformation linéaire (via une matrice 4x4) à un ensemble de points. La fonction transform_aabb transforme les coins de l'AABB et recalcule Pmin et Pmax pour la nouvelle AABB.

Note : l'aabb verte est l'aabb de la aabb transformée

L'aabb bleu est l'aabb des points transformés

L'aabb rouge est l'aabb d'origine

Voici comment on les calcul :

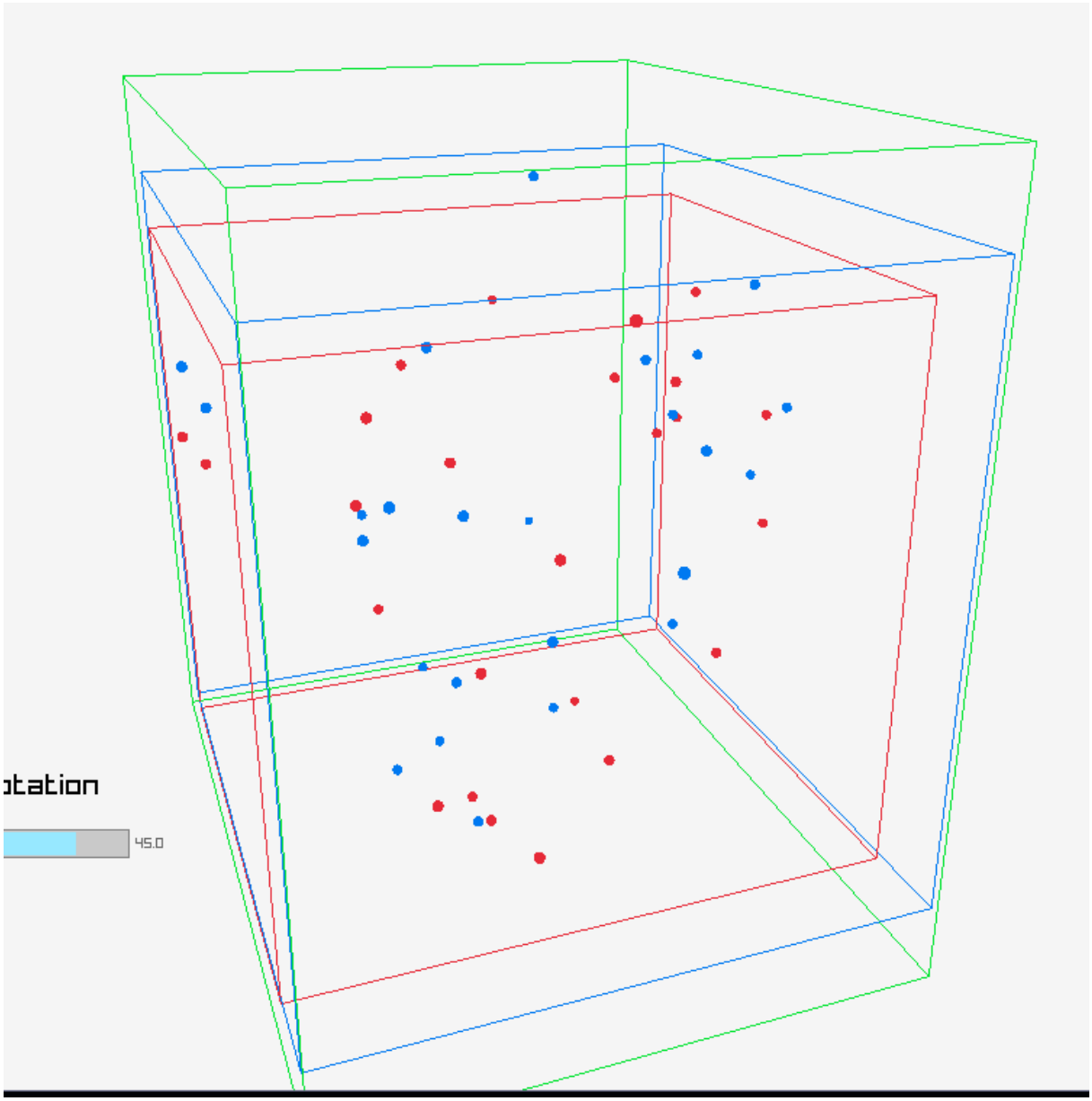
```
# Appliquer la transformation à chaque itération
rotation_matrix = rotation_matrix_x(angle_radians)
transformed_points = apply_transformation(points, rotation_matrix)

# Transformer l'AABB
pmin, pmax = calc_pmin_pmax(points)
pmin_trans, pmax_trans = calc_pmin_pmax(transformed_points)
pmin_matrix, pmax_matrix = transform_aabb(pmin, pmax, rotation_matrix)
pr.begin_drawing()
pr.clear_background(pr.RAYWHITE)

pr.begin_mode_3d(camera)

# Dessiner les points et AABB avant et après transformation
draw_aabb(pmin, pmax, pr.RED)
draw_points(points, pr.RED)
draw_aabb(pmin_trans, pmax_trans, pr.BLUE)
draw_points(transformed_points, pr.BLUE)
draw_aabb(pmin_matrix, pmax_matrix, pr.GREEN)
```

Screenshot :



Exercice 3)

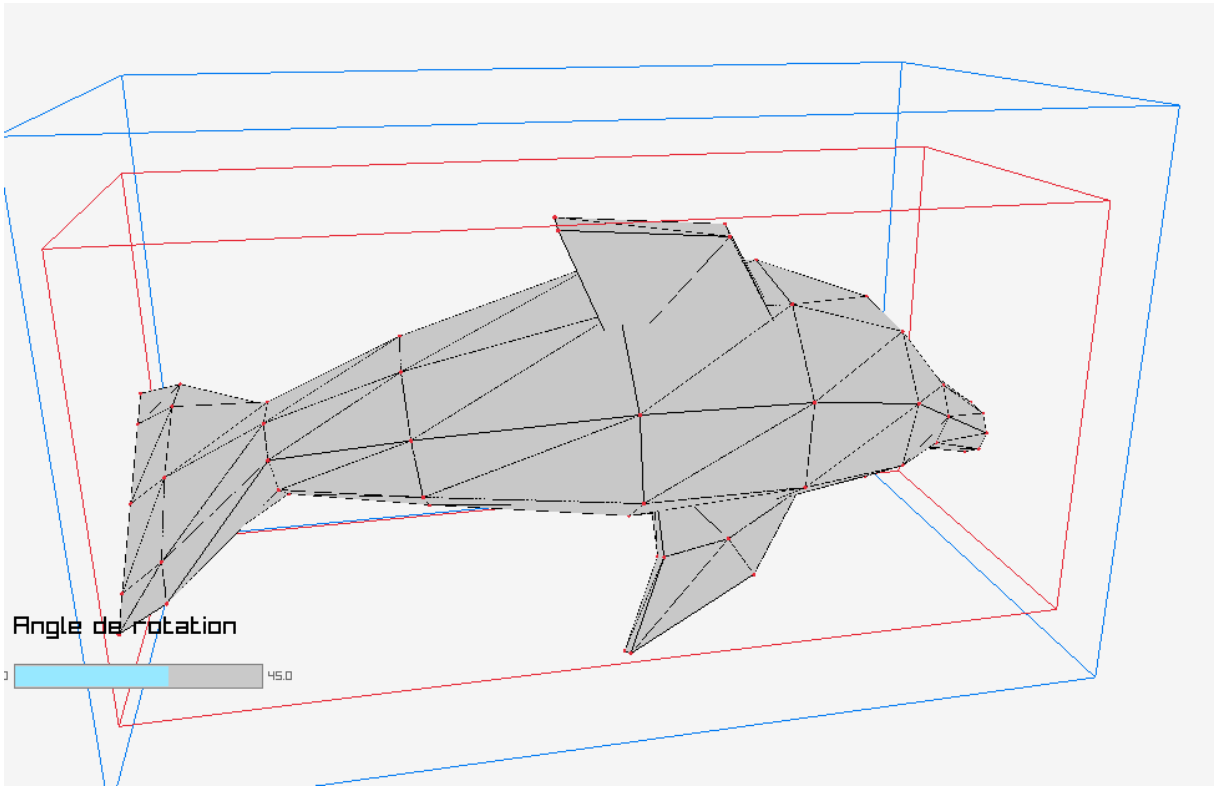
Dans cet exercice, on doit adapté l'exercice d'avant pour qu'il puisse charger un fichier .ply.

Pour ce faire, on doit importer les méthodes draw_mesh, load_ply_file, initialize_mesh_for_transforming

Vector3(*v) sort les composantes de v en v.x, v.y, v.z, réalise une action de « unpacking components ».

Apply_transformation est modifié pour appliqué la matrice a l'ensemble des coins que composent le mesh.

Screenshot :


Angle de rotation
45.0