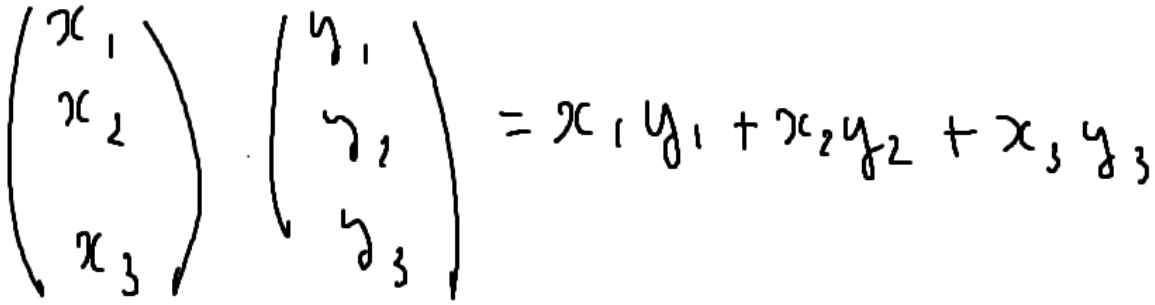
Exercice 1)

1) L'objectif de cette question est de développer la fonction dot_product, qui prend deux vecteurs en paramètre et retourne le produit scalaire entre les deux vecteurs.
   On utilise la formule suivante :



   Voici la fonction implémentée dans le code :

```python
def dot_product(A, B):
    """
    Retourne le produit scalaire entre A et B
    """
    return A.x * B.x + A.y * B.y + A.z * B.z
```

   On prend chaque composante du vecteur A et du vecteur B et on les fait multiplier entre-elles. A et B sont des pr.Vector3, donc ils ont des composantes .x, .y et .z

2) L'objectif de cette question est de développer la fonction vector_length, qui prend en paramètre un vecteur et qui retourne la norme du vecteur donné.
On utilise la formule suivante :

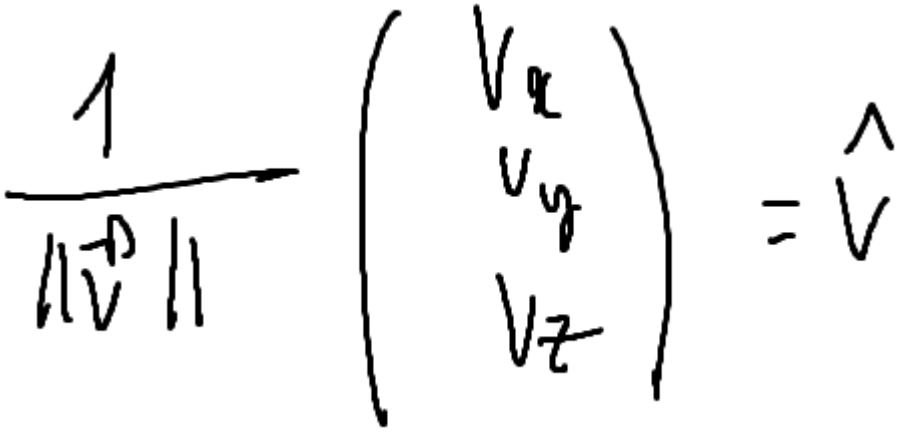
$$\text{Soit } \vec{v} = \begin{pmatrix} x \\ y \\ z \end{pmatrix}, \quad \|\vec{v}\| = \sqrt{x^2 + y^2 + z^2}$$

Voici le code de la solution implémentée :

```
def vector_length(vector):
    """
    Retourne la norme de vector
    """
    return math.sqrt(
        dot_product(vector, vector)
        )
```

On remarque que l'argument sous la racine n'est autre que le produit scalaire de vector avec lui-même, il suffit donc de prendre la racine de la fonction déjà développée ci-dessus.

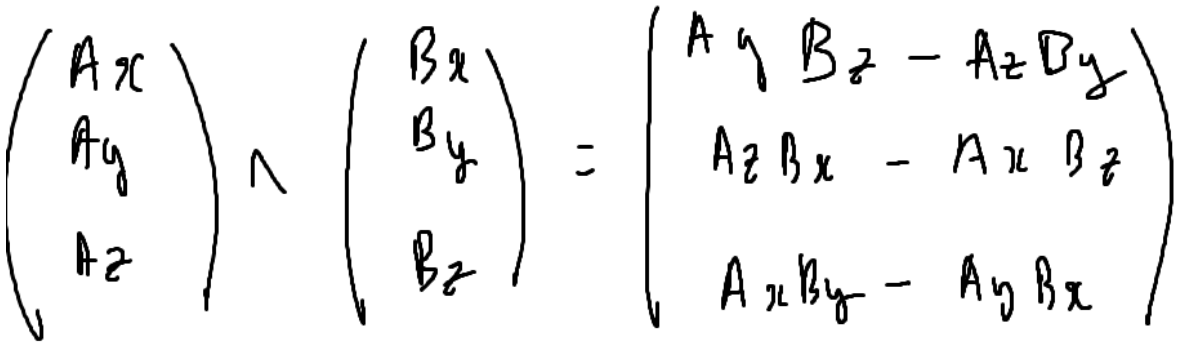
3) L'objectif de cette question est de développé la fonction vector_normalize, qui retourne le vecteur donné en paramètre, mais avec une norme de 1 (unitaire).
On utilise la formule suivante :



Voici le code implémenté pour la solution :

```
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
                   )
```

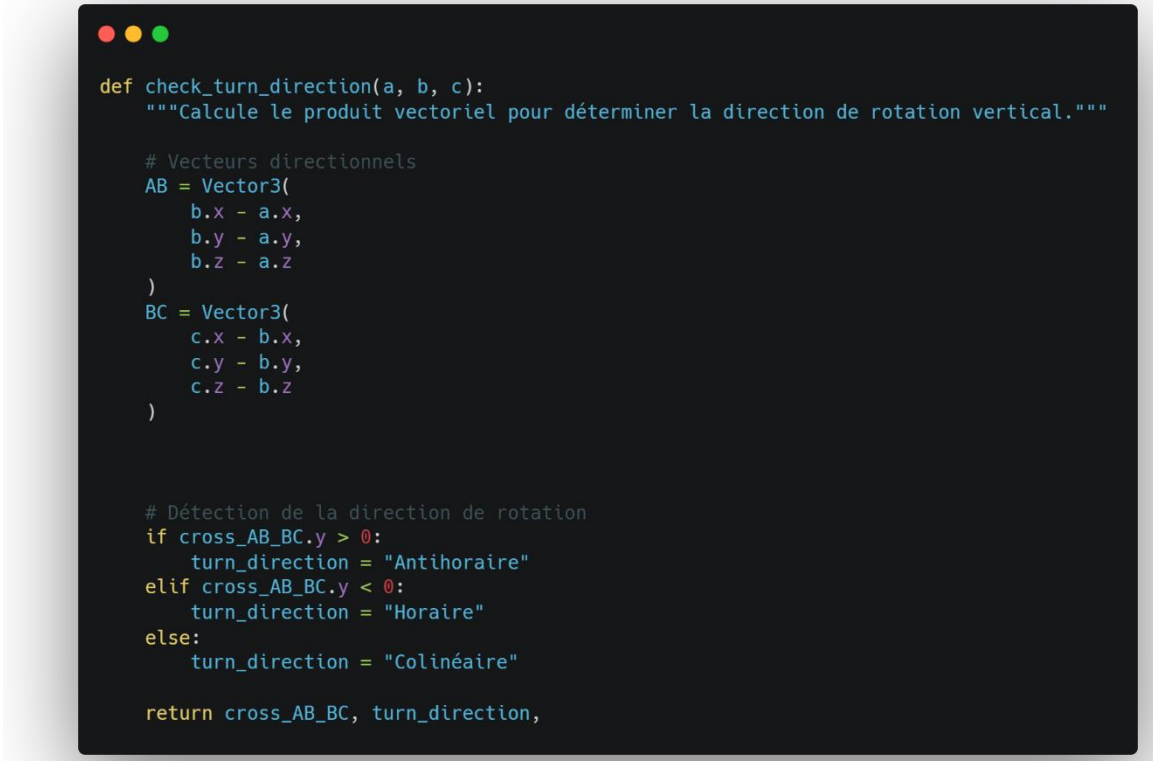
4) L'objectif de cette question est de développé la fonction cross_product, qui retourne le produit vectoriel entre deux vecteurs.
   On utilise la formule suivante :



Voici le code de la solution implémentée :

```python
def cross_product(A, B):
    """
    Retourne le produit vectoriel entre A et B
    """
    return Vector3(
        A.y * B.z - A.z * B.y,
        A.z * B.x - A.x * B.z,
        A.x * B.y - A.y * B.x
    )
```

5) L'objectif de cet exercice est d'implémenter le contrôle des rotations (horaires, anti-horaires et colinéaires) dans la fonction check_turn_direction. Pour ce faire, on utilise la composante y du produit vectoriel entre deux vecteurs construits à partir de trois points consécutifs, présents dans le np.array points.
Voici le code implémenté :



```python
def check_turn_direction(a, b, c):
    """Calcule le produit vectoriel pour déterminer la direction de rotation vertical."""

    # Vecteurs directionnels
    AB = Vector3(
        b.x - a.x,
        b.y - a.y,
        b.z - a.z
    )
    BC = Vector3(
        c.x - b.x,
        c.y - b.y,
        c.z - b.z
    )



    # Détection de la direction de rotation
    if cross_AB_BC.y > 0:
        turn_direction = "Antihoraire"
    elif cross_AB_BC.y < 0:
        turn_direction = "Horaire"
    else:
        turn_direction = "Colinéaire"

    return cross_AB_BC, turn_direction,
```

On remarque que on est dans un système de coordonnée droitier, il suffit donc de réaliser la règle de la main droite pour constater que si on a un résultat négatif pour le produit vectoriel, on tourne dans un sens horaire.

Lorsqu'on lance le code maintenant, on voit écrit le résultat de check_turn_direction, plus précisément la toute première.

Nous allons maintenant essayer de déterminer si on va vers le haut, vers le bas, et faire en sorte que sur les segments reliant les points, il soit écrit si on tourne dans le sens horaire, anti-horaire ou colinéairement.

Exercice 2)

1) L'objectif de cette question est d'utiliser la fonction generate_maze_path pour généré un chemin type labyrinthe, randomisé. On veut analyser une liste de points pour indiquer les virages à gauche et à droite qui se produisent.
   Pour ce faire, on doit écrire dans la méthode main(), plus précisément dans la boucle qui dessine le maze.
   Voici le code implémenté :

```
while not pr.window_should_close():
        update_camera_position(camera, movement_speed)
        draw_scene(camera, grid_size, points, direction, turn)
        for i in range(len(points)-2):
            cross_product_res, turn_direction_res, vertical_movement_res =
check_turn_direction(points[i], points[i+1], points[i+2])
            text_turn_direction_position = Vector3(
                points[i+1].x,
                points[i+1].y,
                points[i+1].z
            )

            draw_text_if_visible_3(camera, turn_direction_res, text_turn_direction_position, 10,
pr.BLACK)
```

On prend trois point consécutifs et on écrit la direction au sommet du premier vecteur.

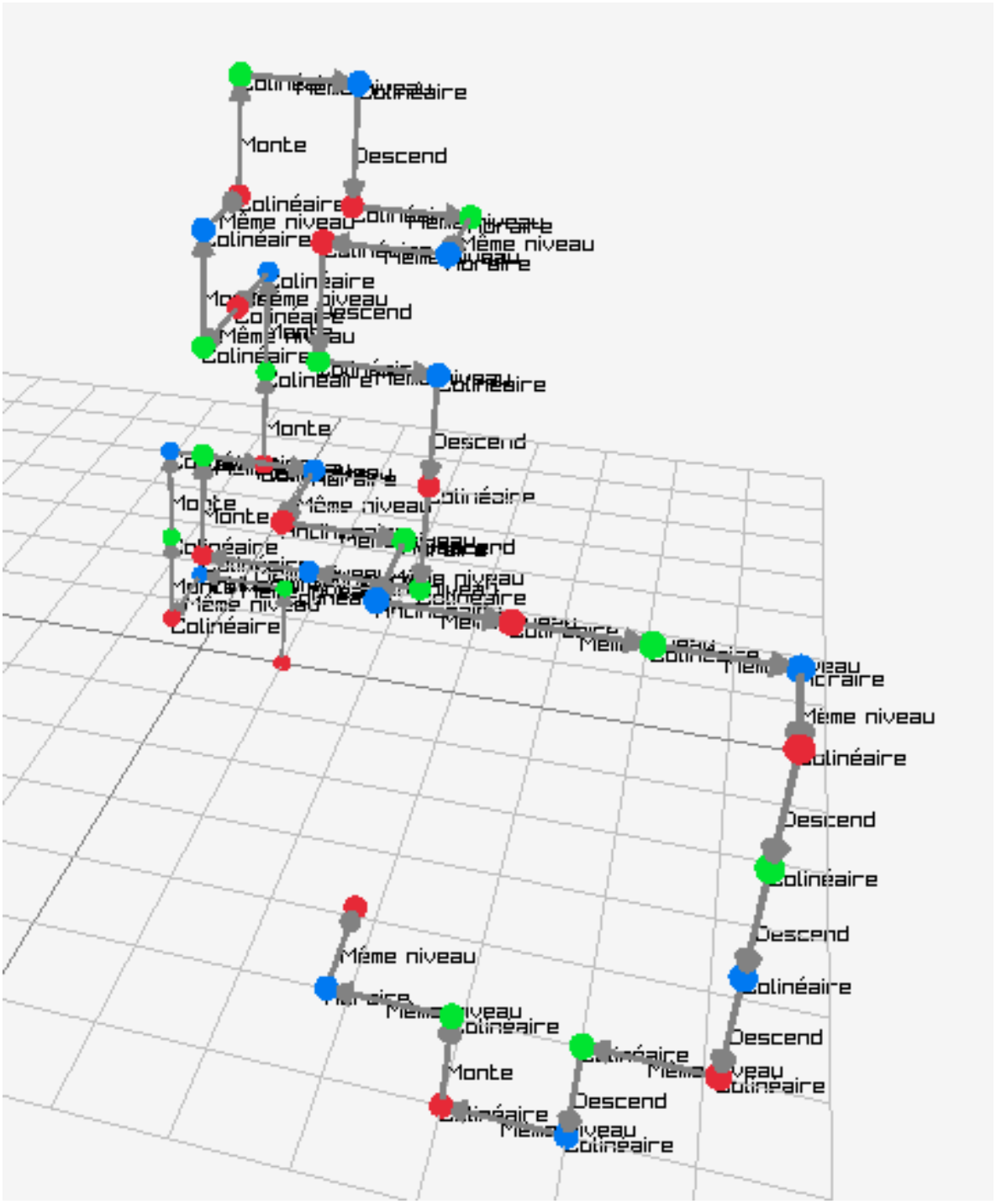
2) Déjà réalisé au-dessus. Maintenant, intéressons-nous à la question bonus. On souhaite vérifier si on va vers le haut ou vers le bas. Pour ce faire, on met l'argument False de la méthode generate_maze_path à True, afin de générer un maze qui possède de la profondeur sur l'axe y.
Voici le code implémenté :

```
ef check_turn_direction(a, b, c):
    """Calcule le produit vectoriel pour déterminer la direction de rotation et le mouvement vertical."""

    # Vecteurs directionnels
    AB = Vector3(
        b.x - a.x,
        b.y - a.y,
        b.z - a.z
    )
    BC = Vector3(
        c.x - b.x,
        c.y - b.y,
        c.z - b.z
    )

    # Produit vectoriel pour la direction de rotation
    cross_AB_BC = cross_product(AB, BC)

    # Détection du mouvement vertical
    if c.y > b.y:
        vertical_movement = "Monte"
    elif c.y < b.y:
        vertical_movement = "Descend"
    else:
        vertical_movement = "Même niveau"

    # Détection de la direction de rotation
    if cross_AB_BC.y > 0:
        turn_direction = "Antihoraire"
    elif cross_AB_BC.y < 0:
        turn_direction = "Horaire"
    else:
        turn_direction = "Colinéaire"

    return cross_AB_BC, turn_direction, vertical_movement
```

Tout d'abord, on doit modifier la fonction check_turn_direction afin de vérifier à partir de la position relative des points entre eux, si on se déplace vers le haut ou vers le bas. Pour ce faire, on compare la position y du point i+1 au voisinage du point i.
Ensuite, on doit modifier la boucle de rendu graphique :

```
while not pr.window_should_close():
        update_camera_position(camera, movement_speed)
        draw_scene(camera, grid_size, points, direction, turn)
        for i in range(len(points)-2):
            cross_product_res, turn_direction_res, vertical_movement_res =
check_turn_direction(points[i], points[i+1], points[i+2])
            text_turn_direction_position = Vector3(
                points[i+1].x,
                points[i+1].y,
                points[i+1].z
            )
            text_vertical_movement_position = Vector3(
                (points[i+1].x + points[i+2].x) / 2,
                (points[i+1].y + points[i+2].y) / 2,
                (points[i+1].z + points [i+2].z) / 2
            )
            draw_text_if_visible_3(camera, turn_direction_res, text_turn_direction_position, 10,
pr.BLACK)
            draw_text_if_visible_3(camera, vertical_movement_res, text_vertical_movement_position, 10,
pr.BLACK)
    pr.close_window()
```

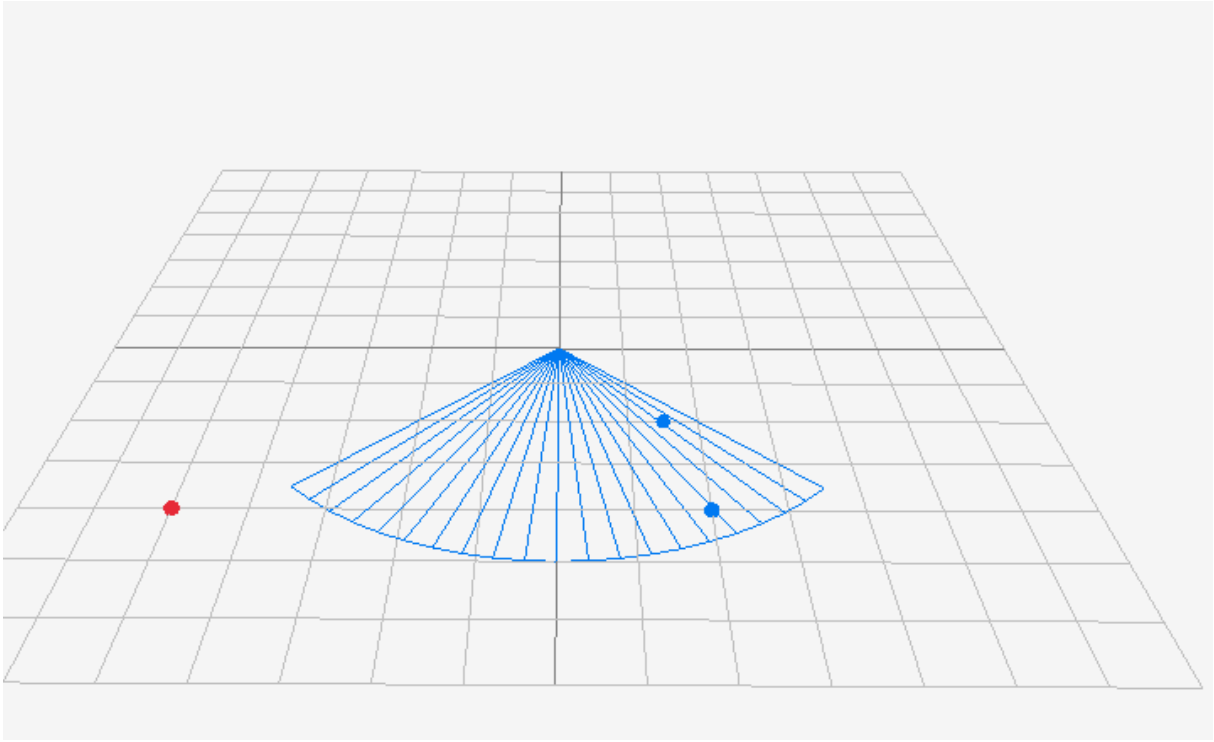
Ici, on a tout simplement ajouté un nouveau résultat à traîté : vertical_movement_res, et comme pour le cas horaire/antihoraire, on souhaite le placer sur le moteur graphique. Pour ce faire, on le place au milieu en calculant le milieu du segment.
Screenshot du rendu final :



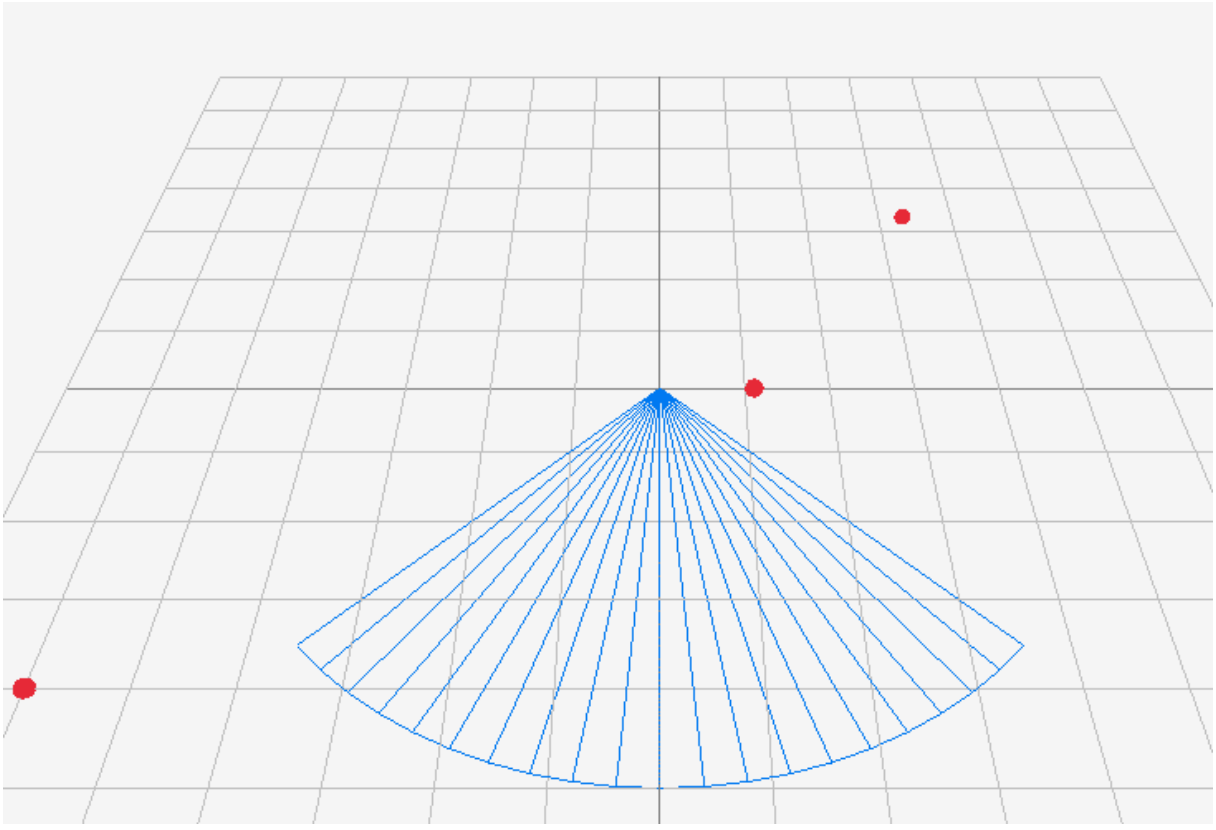
Exercice 3)
Dans cet exercice, on doit implémennter la fonction is_point_in_fov en suivant les directives du code. Des points sont placés, avec un fov de dessiné, si le point est dans le fov, cette fonction doit retourné True, sinon False.
Voici le code implémenté :

```python
def is_point_in_fov(fov_position, fov_direction, fov_distance, fov_angle, point):
    """
    Vérifie si un point est dans le champ de vision défini par une position, une direction, une distance
et un angle.

    :param fov_position: Position du point de départ de FOV.
    :param fov_direction: Direction centrale du FOV.
    :param fov_distance: Distance maximale de portée de la FOV.
    :param fov_angle: Angle du FOV en degrés.
    :param point: Point à vérifier.
    :return: True si le point est dans le champ de vision, sinon False.
    """
    #  : Vecteur du FOV vers le point

    to_point = Vector3(
        point.x - fov_position.x,
        point.y - fov_position.y,
        point.z - fov_position.z
    )

    #  : Calcule la distance au point et vérifie qu'elle est dans la distance FOV

    distance_to_point = vector_length(to_point)
    if distance_to_point > fov_distance:
        return False

    # Normalise la direction du FOV et le vecteur vers le point
    norm_fov_direction = vector_normalize(fov_direction)
    norm_to_point = vector_normalize(to_point)

    # : Calcule le produit scalaire

    dot_res = dot_product(norm_fov_direction, norm_to_point)

    #  Calcule le cosinus de l'angle demi du FOV

    cos_res = math.cos(math.radians(fov_angle) / 2)

    #  Vérifie si le produit scalaire satisfait la condition du FOV
    return dot_res >= cos_res
```

Dans un premier temps, nous devons construire to_point, to_point est le vecteur qui relie point et fov_position. Ensuite, on doit calculer distance_to_point, qui est simplement la norme du vecteur to_point. Si cette distance est supérieure à la distance fov, on est en dehors du fov en termes de distance peu importe l'angle, on peut donc arrêter directement la fonction. Sinon, on normalise fov_direction et to_point, pour réaliser un produit scalaire entre les deux, l'objectif de ce produit scalaire est d'avoir l'angle entre ces deux vecteurs. Enfin, on calcul l'angle demi en convertissant en radiant fov_angle, si dot_res est plus grand que cos_res, c'est qu'on est en dehors du fov peu importe la distance.
Enfin, en changeant les couleurs et en ajoutant quelques conditions pour rendre le rendu plus attractif, on tombe sur le résultat ci-dessous.
Screenshot du rendu final :



En bleu les points dans le fov, en rouge les points qui sont en dehors.
Un autre screenshot :



Exercice 4)
Dans cet exercice, on se propose de crée un parallélogramme afin d'en sortir la surface. Pour ce faire, on se base sur les exercices précédents. Nous avons

besoin de pouvoir tracer des vecteurs, les relié entre eux à partir de points donnés, et enfin afficher la surface.
Voici le code implémenté :

```python
def draw_parallelogram(vec_1, vec_2):
    """Dessine un parallélogramme défini par deux vecteurs en utilisant des vecteurs au lieu de lignes."""
    color = pr.BLUE

    trans = Vector3(vec_1.x + vec_2.x, vec_1.y + vec_2.y, vec_1.z + vec_2.z)

    # Dessiner les vecteurs
    draw_vector_3(Vector3(0, 0, 0), vec_1, color)  # Vecteur du point d'origine vers le premier vecteur
    draw_vector_3(Vector3(0, 0, 0), vec_2, color)  # Vecteur du point d'origine vers le deuxième vecteur
    draw_vector_3(vec_1, trans, color)             # Vecteur du premier sommet vers le sommet opposé
    draw_vector_3(vec_2, trans, color)             # Vecteur du deuxième sommet vers le sommet opposé
```

Pour le premier point, il se situe à l'origine. On veut d'abord tracer une ligne, puis, réaliser une translation pour relier les deux bouts.
Enfin, pour calculer la surface :

```python
def parallelogram_area(vec_1, vec_2):
    return vector_length(cross_product(vec_1,vec_2))
```

On utilise la formule suivante :

$$A = \|v_1 \times v_2\|$$

On doit maintenant afficher le résultat du calcul de surface et tracer les vecteurs dans un plan, pour se faire, on utilise les fonctions des exercices précédents :

```
while not pr.window_should_close():
        update_camera_position(camera, movement_speed)

        pr.begin_drawing()
        pr.clear_background(pr.RAYWHITE)
        pr.begin_mode_3d(camera)

        pr.draw_grid(grid_size, 1)
        draw_parallelogram(vec_1, vec_2)

        pr.end_mode_3d()

        draw_text_if_visible_3(camera, f"Surface: {area:.2f}", Vector3(2, 2, 2), 20, pr.BLACK)

        pr.end_drawing()
    pr.close_window()
```

Exercice 5)
Dans cet exercice, on nous fournit un fichier .ply que l'on exploite à l'aide du module trimesh.

1) Dans cette question, on doit calculé le vecteur n, à partir de la formule suivante : $\mathbf{n} = \mathbf{a} \times \mathbf{b} / ||\mathbf{a} \times \mathbf{b}||$

Pour ce faire, on implémente la fonction suivante, qui n'est autre qu'un calcul direct à partir de la formule :

a et b sont des edge du mesh, on doit donc les calculer en premier lieu. Ensuite, on réalise le produit vectoriel entre les deux, et en normalize le vecteur ainsi obtenu. Ce qui donne la fonction suivante pour calculer les normales :

```python
def compute_face_normals(mesh):
    """Calcule les normales pour chaque face du mesh et retourne une liste de tuples (centre,
normale)."""
    normals = []
    for face in mesh.faces:
        v0 = Vector3(*mesh.vertices[face[0]])
        v1 = Vector3(*mesh.vertices[face[1]])
        v2 = Vector3(*mesh.vertices[face[2]])

        # TODO : Calculer le vecteur normal de la face
        edge1 = Vector3(v1.x - v0.x, v1.y - v0.y, v1.z - v0.z) # TODO
        edge2 = Vector3(v2.x - v1.x, v2.y - v1.y, v2.z - v1.z) # TODO
        normal = cross_product(edge1, edge2) # TODO
        normal = vector_normalize(normal)

        # Calculer le centre de la face
        center = compute_face_center(v0, v1, v2)

        normals.append((center, normal))
    return normals
```

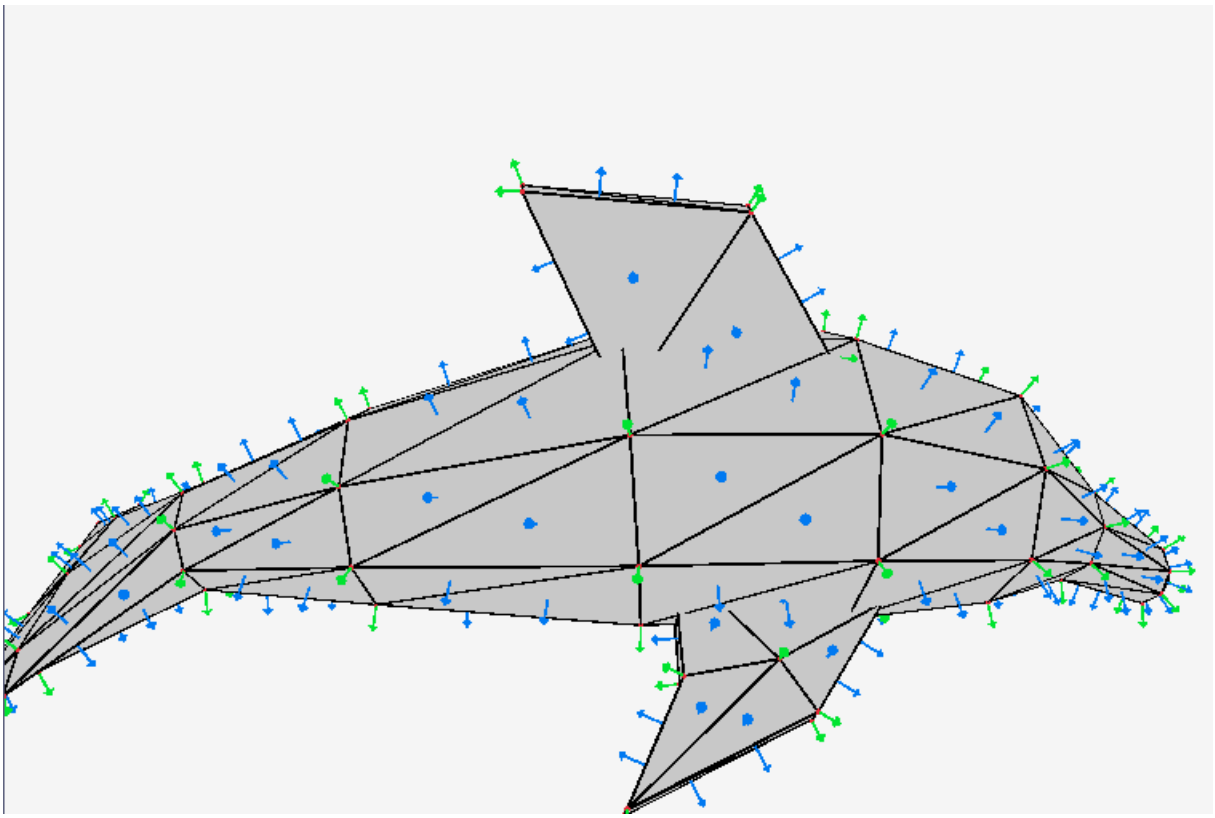
2) Dans cette question, on doit déterminer le centre de gravité de chaque face.
   Pour ce faire, on fait la moyenne des coordonnées des sommets qui définissent la face considérée, et ainsi pour chaque face.
   Dolphin.ply étant un maillage polygonal composé de triangles, on peut facilement trouver le centre de gravité à l'aide de cette fonction :

```python
def compute_face_center(v0, v1, v2):
    """Calcule le centre d'une face triangulaire."""
    return Vector3(
        (v0.x + v1.x + v2.x) / 3,
        (v0.y + v1.y + v2.y) / 3,
        (v0.z + v1.z + v2.z) / 3
    )
```

   Aussi, la fonction suivante a été complétée :

```python
def compute_vertex_normals(mesh, face_normals):
    """
    Calcule les normales pour chaque sommet en moyennant les normales des faces adjacentes.
    Retourne un dictionnaire avec chaque sommet et son vecteur normal associé.
    """

    # TODO : mettre en œuvre le calcul des normales pour chaque sommet en moyennant les normales des
faces adjacentes
    vertex_normals = {i: Vector3(0, 0, 0) for i in range(len(mesh.vertices))}
    vertex_count = {i: 0 for i in range(len(mesh.vertices))}

    for face, (_, normal) in zip(mesh.faces, face_normals):
        for vertex_index in face:
            vertex_normals[vertex_index].x += normal.x
            vertex_normals[vertex_index].y += normal.y
            vertex_normals[vertex_index].z += normal.z
            vertex_count[vertex_index] += 1

    # TODO : Moyenne les normales (n'oubliez pas de normaliser)
    for i in vertex_normals:

        if vertex_count[i] > 0:
            vertex_normals[i].x /= vertex_count[i]
            vertex_normals[i].y /= vertex_count[i]
            vertex_normals[i].z /= vertex_count[i]

            vertex_normals[i] = vector_normalize(vertex_normals[i])


    return vertex_normals
```

Screenshot du résultat



On remarque qu’aux extrémités, on ades vecteurs normaux en vert, qui sont en adéquation avec ce qui a été vu en cours, ces endroits représentent des discontinuitées.