TSI : Synthèse d'images,
Projet OpenGL
CPE

Le projet est à rendre sur github pour le **28/06/2024 à 23h59**.

1 Objets 3D

1.1 Plan

Dans un environnement 3D, les objets sont souvent représentés par des maillages, qui sont des collections de sommets, d'arêtes et de faces. On stocke pour chaque sommet sa position dans l'espace, ainsi que sa normale, sa couleur et ses coordonnées de texture.
Pour un plan sur l'axe Z, par exemple, on peut définir les sommets suivants :

```
p0 = (0, 0, 0)  # Coin inférieur gauche
p1 = (1, 0, 0)  # Coin inférieur droit
p2 = (1, 1, 0)  # Coin supérieur droit
p3 = (0, 1, 0)  # Coin supérieur gauche
```

On peut également ajouter des normales pour chaque sommet, qui sont des vecteurs perpendiculaires à la surface du maillage.

```
n0 = (0, 0, 1)  # Normale pour le plan sur l'axe Z
n1 = (0, 0, 1)  # Normale pour le plan sur l'axe Z
n2 = (0, 0, 1)  # Normale pour le plan sur l'axe Z
n3 = (0, 0, 1)  # Normale pour le plan sur l'axe Z
```

On peut aussi ajouter des couleurs pour chaque sommet, par exemple en utilisant des valeurs RGB :

```
c0 = (1, 0, 0)  # Rouge
c1 = (0, 1, 0)  # Vert
c2 = (0, 0, 1)  # Bleu
c3 = (1, 1, 0)  # Jaune
```

Finalement, on peut ajouter des coordonnées de texture pour chaque sommet, qui sont utilisées pour mapper une texture sur le maillage :

```
uv0 = (0, 0)  # Coordonnées de texture pour le coin inférieur gauche
uv1 = (1, 0)  # Coordonnées de texture pour le coin inférieur droit
uv2 = (1, 1)  # Coordonnées de texture pour le coin supérieur droit
uv3 = (0, 1)  # Coordonnées de texture pour le coin supérieur gauche
```

On peut concatener les coordonnées des sommets et des normales dans une liste pour les envoyer au GPU :

```
sommets = np.concatenate((p0, n0, c0, uv0, p1, n1, c1, uv1, p2, n2, c2, uv2, p2,
```

On peut ensuite définir les faces du plan en utilisant les indices des sommets :

```
f0 = (0, 1, 2)  # Triangle inférieur droit
f1 = (0, 2, 3)  # Triangle supérieur gauche
```

Question 1 *Créez un maillage représentant un plan sur l'axe Z en utilisant les vertex définis ci-dessus.*

1.2 Cube

Un cube est constitué de six faces carrées. Pour obtenir des normales et des coordonnées de texture correctes, chaque face doit disposer de ses **propres sommets**, ce qui donne 24 sommets au total (4 sommets $\times$ 6 faces).
Voici les sommets définis pour chaque face. Le texte suivant est généré automatiquement via un llm, il est important de le comprendre mais dans une cas réel, on utiliserait un outil de modélisation 3D pour créer le maillage.
Face avant (z = 1), couleur rouge :

```
p0 = (0, 0, 1)
p1 = (1, 0, 1)
p2 = (1, 1, 1)
p3 = (0, 1, 1)

n0 = (0, 0, 1)
n1 = (0, 0, 1)
n2 = (0, 0, 1)
n3 = (0, 0, 1)

c0 = (1, 0, 0)  # Rouge
c1 = (1, 0, 0)
c2 = (1, 0, 0)
c3 = (1, 0, 0)

uv0 = (0, 0)
uv1 = (1, 0)
uv2 = (1, 1)
uv3 = (0, 1)
```

Face arrière (z = 0), couleur vert :

```
p4 = (1, 0, 0)
p5 = (0, 0, 0)
p6 = (0, 1, 0)
p7 = (1, 1, 0)

n4 = (0, 0, -1)
n5 = (0, 0, -1)
n6 = (0, 0, -1)
n7 = (0, 0, -1)

c4 = (0, 1, 0)  # Vert
c5 = (0, 1, 0)
c6 = (0, 1, 0)
c7 = (0, 1, 0)

uv4 = (0, 0)
uv5 = (1, 0)
uv6 = (1, 1)
uv7 = (0, 1)
```

Face droite (x = 1), couleur bleu :

```
p8  = (1, 0, 1)
p9  = (1, 0, 0)
p10 = (1, 1, 0)
p11 = (1, 1, 1)

n8  = (1, 0, 0)
n9  = (1, 0, 0)
n10 = (1, 0, 0)
n11 = (1, 0, 0)

c8  = (0, 0, 1)  # Bleu
c9  = (0, 0, 1)
c10 = (0, 0, 1)
c11 = (0, 0, 1)

uv8  = (0, 0)
uv9  = (1, 0)
uv10 = (1, 1)
uv11 = (0, 1)
```

Face gauche (x = 0), couleur jaune :

```
p12 = (0, 0, 0)
p13 = (0, 0, 1)
p14 = (0, 1, 1)
p15 = (0, 1, 0)

n12 = (-1, 0, 0)
n13 = (-1, 0, 0)
n14 = (-1, 0, 0)
n15 = (-1, 0, 0)

c12 = (1, 1, 0)  # Jaune
c13 = (1, 1, 0)
c14 = (1, 1, 0)
c15 = (1, 1, 0)

uv12 = (0, 0)
uv13 = (1, 0)
uv14 = (1, 1)
uv15 = (0, 1)
```

Face supérieure (y = 1), couleur cyan :

```
p16 = (0, 1, 1)
p17 = (1, 1, 1)
p18 = (1, 1, 0)
p19 = (0, 1, 0)

n16 = (0, 1, 0)
n17 = (0, 1, 0)
n18 = (0, 1, 0)
n19 = (0, 1, 0)
```

```
c16 = (0, 1, 1)  # Cyan
c17 = (0, 1, 1)
c18 = (0, 1, 1)
c19 = (0, 1, 1)

uv16 = (0, 0)
uv17 = (1, 0)
uv18 = (1, 1)
uv19 = (0, 1)
```

Face inférieure (y = 0), couleur magenta :

```
p20 = (0, 0, 0)
p21 = (1, 0, 0)
p22 = (1, 0, 1)
p23 = (0, 0, 1)

n20 = (0, -1, 0)
n21 = (0, -1, 0)
n22 = (0, -1, 0)
n23 = (0, -1, 0)

c20 = (1, 0, 1)  # Magenta
c21 = (1, 0, 1)
c22 = (1, 0, 1)
c23 = (1, 0, 1)

uv20 = (0, 0)
uv21 = (1, 0)
uv22 = (1, 1)
uv23 = (0, 1)
```

Ensuite, on définit les triangles de chaque face à l'aide des indices des sommets :

```
# Face avant
f0 = (0, 1, 2)
f1 = (0, 2, 3)

# Face arrière
f2 = (4, 5, 6)
f3 = (4, 6, 7)

# Face droite
f4 = (8, 9, 10)
f5 = (8, 10, 11)

# Face gauche
f6 = (12, 13, 14)
f7 = (12, 14, 15)

# Face supérieure
f8 = (16, 17, 18)
```

```
f9 = (16, 18, 19)

# Face inférieure
f10 = (20, 21, 22)
f11 = (20, 22, 23)
```

Question 2 *Créez un maillage complet représentant un cube à l'aide des 24 sommets définis, en assignant à chaque sommet sa position, sa normale, sa couleur et ses coordonnées de texture.*

1.3 Sphère

Pour la sphère, il suffit de générer des sommets en utilisant des coordonnées sphériques. on pourra utilser le code suivant pour générer une sphère en Python.

```
import numpy as np

def generate_sphere(radius=1.0, lat_segments=16, lon_segments=32):
    vertices = []

    indices = []

    for i in range(lat_segments + 1):
        theta = np.pi * i / lat_segments
        sin_theta = np.sin(theta)
        cos_theta = np.cos(theta)

        for j in range(lon_segments + 1):
            phi = 2 * np.pi * j / lon_segments
            sin_phi = np.sin(phi)
            cos_phi = np.cos(phi)

            # Position (x, y, z)
            x = radius * sin_theta * cos_phi
            y = radius * cos_theta
            z = radius * sin_theta * sin_phi

            # Normale (normée car sphère unitaire)
            nx = sin_theta * cos_phi
            ny = cos_theta
            nz = sin_theta * sin_phi

            # Texture coordinates (u, v)
            u = j / lon_segments
            v = i / lat_segments

            r, g, b = 1.0, 1.0, 1.0

            # Interlaced vertex: position, normal, color, uv
            vertex = [x, y, z, nx, ny, nz, r, g, b, u, v]
            vertices.append(vertex)

    # Faces
```

```
    for i in range(lat_segments):
        for j in range(lon_segments):
            a = i * (lon_segments + 1) + j
            b = a + lon_segments + 1

            indices.append((a, b, a + 1))
            indices.append((a + 1, b, b + 1))

    return {
        'interlaced': np.array(vertices, dtype=np.float32),  # [pos, norm, color
        'faces': np.array(indices, dtype=np.int32)           # (i0, i1, i2)
    }
```

Question 3 *Créez un maillage représentant une sphère en utilisant la fonction* `generate_sphere()` *ci-dessus. Vous pouvez ajuster les paramètres pour obtenir la résolution souhaitée.*

2 Géométrie 3D

La rotation et la translation d'un objet dans un espace 3D peuvent être représentées à l'aide d'une matrice 4x4 homogène. Cette matrice permet de combiner les transformations de rotation et de translation en une seule opération, simplifiant ainsi les calculs.
Lorsqu'on applique une transformation à une normale, il est important de noter que la quatrième composante du vecteur doit être égale à 0, car les normales ne subissent pas de translation, uniquement des rotations.
Avec la bibliothèque `Pyrr` en Python, il est possible de créer une matrice combinant rotation et translation en utilisant des fonctions telles que `pyrr.matrix44.create_from_translation()` et `pyrr.matrix44.create_from_z_rotation()` pour générer respectivement des matrices de translation et de rotation. Ces matrices peuvent ensuite être multipliées pour obtenir une matrice de modèle complète. On appliquera toujours la rotation avant la translation pour conserver le centre de rotation au centre du repère.

Question 4 *Modifiez vos shaders pour qu'ils utilisent une matrice de modèle unique au lieu de séparer les transformations en une translation et une rotation distinctes.*

Question 5 *Déplacer un objet dans l'espace 3D en utilisant la matrice de modèle avec les touches directionnelles (gauche et droite faisant une rotation), faites en sorte que l'objet se déplace suivant la direction dans laquelle il est orienté.*

3 Caméra

La caméra dans un environnement 3D est souvent modélisée comme un point de vue dans l'espace, avec une direction vers laquelle elle regarde. Pour représenter la caméra, on utilise généralement un système de coordonnées où l'axe Z pointe vers l'avant, l'axe Y vers le haut et l'axe X vers la droite. La matrice de vue est utilisée pour transformer les coordonnées des objets du monde en coordonnées de la caméra. Cette matrice est souvent construite à partir de la position de la caméra, de la direction vers laquelle elle regarde et d'un vecteur "up" qui définit l'orientation verticale.

Question 6 *Implémentez une caméra dans votre scène 3D. Vous pouvez utiliser la méthode create_look_at de la bibliothèque* `Pyrr` *pour créer la matrice de vue. Utilisez cette matrice dans votre shader pour transformer les coordonnées des objets du monde en coordonnées de la caméra avec la projection.*

On pourra attacher la caméra à un objet, par exemple un personnage, pour que la caméra suive les mouvements de cet objet. Il suffirat de positionner la caméra à la position de l'objet avec un petit décallage pour être derrière lui.

Question 7 *Attachez la caméra à un objet (par exemple, un personnage) de votre scène. Faites en sorte que la caméra suive les mouvements de cet objet tout en gardant une certaine distance.*

4 Physique

La physique dans un environnement 3D peut être simulée en utilisant des concepts tels que la gravité, les forces et les collisions. Pour simuler la gravité, on peut appliquer une force constante vers le bas sur les objets, ce qui les fait tomber. Pour gérer les collisions, on peut utiliser des algorithmes de détection de collision qui vérifient si deux objets se chevauchent ou entrent en contact.

4.1 Gravité et schéma d'Euler explicite

Le principe fondamental de la dynamique, connu sous le nom de deuxième loi de Newton, stipule que la somme des forces appliquées à un objet est égale à la masse de cet objet multipliée par son accélération :

$$\vec{F} = m \cdot \vec{a}$$

où :

- $\vec{F}$ est la force résultante appliquée à l'objet,
- m est la masse de l'objet,
- $\vec{a}$ est l'accélération de l'objet.

Pour simuler la gravité, on applique une force constante $\vec{F}_g = m \cdot \vec{g}$, où $\vec{g}$ est le vecteur gravité (par exemple, $\vec{g} = (0, -9.81, 0)$ en m/s²).
Le schéma d'Euler explicite est une méthode numérique simple pour résoudre les équations différentielles. Il repose sur l'approximation suivante :

$$\vec{v}(t + \Delta t) = \vec{v}(t) + \vec{a}(t) \cdot \Delta t$$

$$\vec{p}(t + \Delta t) = \vec{p}(t) + \vec{v}(t) \cdot \Delta t$$

où :

- $\vec{v}$ est la vitesse,
- $\vec{p}$ est la position,
- Δt est le pas de temps.

Voici un pseudocode pour simuler la gravité en utilisant le schéma d'Euler explicite :

```
# Initialisation
position = [x0, y0, z0]  # Position initiale
velocity = [vx0, vy0, vz0]  # Vitesse initiale
gravity = [0, -9.81, 0]  # Accélération due à la gravité
mass = m  # Masse de l'objet
time_step = dt  # Pas de temps
```

```
# Boucle de simulation
while simulation_running:
    # Calcul de l'accélération
    acceleration = gravity  # Pas d'autres forces dans cet exemple
    # On note ici que l'accélération ne dépend pas de la masse et qu'elle est co

    # Mise à jour de la vitesse
    velocity = velocity + acceleration * time_step

    # Mise à jour de la position
    position = position + velocity * time_step
```

Question 8 *Implémentez la simulation de la gravité en utilisant le schéma d'Euler explicite. Vous pouvez utiliser un pas de temps fixe, ou alors utiliser la différence de temps entre deux frames.*

4.2 Collisions

La détection de collisions dans un environnement 3D peut être réalisée en utilisant des algorithmes géométriques. Pour des objets simples comme des sphères ou des cubes, on peut utiliser des tests de collision basés sur la distance entre les objets. Pour un objet et un plan, on peut vérifier si la distance entre le point le plus proche de l'objet et le plan est inférieure à un certain seuil.

```
# Pseudo-algorithme pour la collision avec un plan en z=0

# Initialisation
position = [x, y, z]  # Position actuelle de l'objet
velocity = [vx, vy, vz]  # Vitesse actuelle de l'objet
radius = r  # Rayon de l'objet (si c'est une sphère, par exemple)

# Détection de collision
if position[2] - radius <= 0:  # Vérifie si l'objet touche ou passe sous le plan
# Résolution de la collision
position[2] = radius  # Replace l'objet juste au-dessus du plan
velocity[2] = 0 # Annule la composante verticale de la vitesse
```

Pour une sphère, la collision peut être détectée en vérifiant si la distance entre les centres des deux sphères est inférieure à la somme de leurs rayons.

```
# Pseudo-algorithme pour la collision entre deux sphères

# Initialisation
position1 = [x1, y1, z1]  # Position de la première sphère
position2 = [x2, y2, z2]  # Position de la deuxième sphère
velocity1 = [vx1, vy1, vz1]  # Vitesse de la première sphère
velocity2 = [vx2, vy2, vz2]  # Vitesse de la deuxième sphère
radius1 = r1  # Rayon de la première sphère
radius2 = r2  # Rayon de la deuxième sphère

# Détection de collision
distance = sqrt((x2 - x1)**2 + (y2 - y1)**2 + (z2 - z1)**2)
if distance <= radius1 + radius2:
  # Résolution de la collision
```

Pour un cube alignés sur les axes, on peut vérifier si les coordonnées des sommets de l'un des cubes se trouvent à l'intérieur des limites de l'autre cube.

```
# Pseudo-algorithme pour la collision entre deux cubes alignés sur les axes
# Initialisation
position1 = [x1, y1, z1]  # Position du premier cube
position2 = [x2, y2, z2]  # Position du deuxième cube
size1 = [sx1, sy1, sz1]  # Taille du premier cube
size2 = [sx2, sy2, sz2]  # Taille du deuxième cube
# Détection de collision
if (position1[0] < position2[0] + size2[0] and  # axe X
    position1[0] + size1[0] > position2[0] and
    position1[1] < position2[1] + size2[1] and  # axe Y
    position1[1] + size1[1] > position2[1] and
    position1[2] < position2[2] + size2[2] and  # axe Z
    position1[2] + size1[2] > position2[2]):
  # Résolution de la collision
```

Question 9 *Implémentez un algorithme de détection de collision pour des objets simples tels que un plan, des sphères et des cubes.*

5 Jeu

Question 10 *Réalisez un jeu 3D minimaliste avec un personne vue à la 3e personnes navigue dans un monde 3D.*

Le sujet de votre jeu est libre mais doit contenir au moins deux programmes graphiques différents, des interactions utilisateurs et aussi la réutilisation intélligente d'objets (affiché plusieurs fois).

Remarques On rappelle qu'une rotation s'applique toujours autour de l'origine, il est donc nécessaire de translater l'objet si on souhaite faire tourner celui-ci autour d'un autre point tel que son centre par exemple

- Faites des choses simples dans un premier temps.
- Prêtez une attention particulière aux déplacements de vos objets ainsi qu'au déplacement possible de votre caméra. Définissez vos repères et faites des schémas.
- Dans le cas de la gestion des collisions réfléchissez comment détecter de manière approximative celles-ci.

Rendu Rendez votre travail sous forme d'un commit sur github contenant:

- L'ensemble de vos fichiers sources.
- L'ensemble de vos données (textures, fichiers de maillages, etc).
- Un rapport concis expliquant ce que vous avez réalisé (5 à 10 pages max). Vous prendrez soin d'illustrer votre rapport avec des captures d'écran de votre jeu associé aux explications des méthodes que vous avez mis en place. Pour des approches géométriques, préférez des explications sous forme de schémas plutôt que de longs paragraphes.