Réalisation du rendu 3D pour les jeux vidéo

Tanana Mehdi

2023/2024 — 21135

Mise en situation

Figure: The Legend of Zelda BOTW - 2017

Figure: KCorp - RLCS 2023

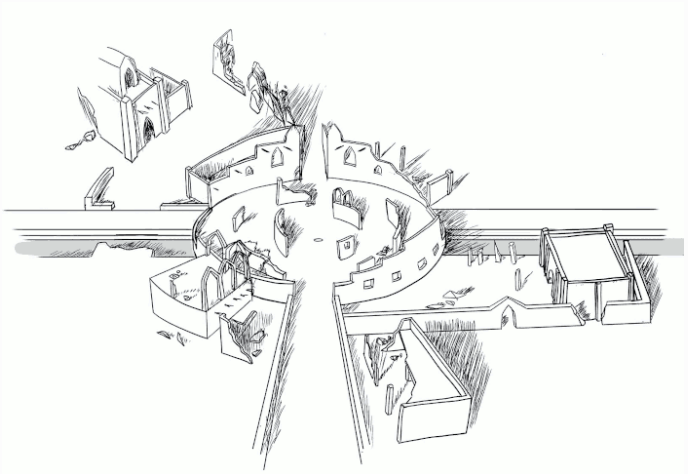


Figure: Prototype d'une scène

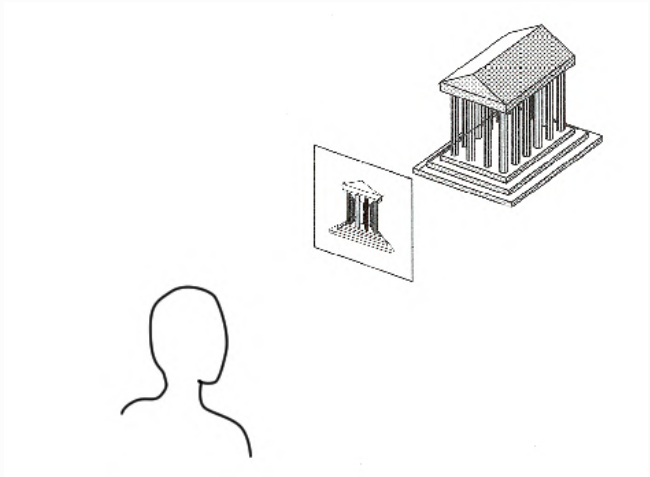


Figure: Vision du joueur

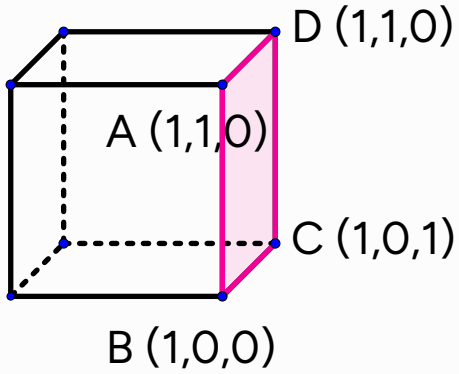
Problématique

Comment créer et afficher une scène 3D dans un jeu vidéo ?

Plan

- Modélisation du solide
- Modélisation du joueur
- Projections des solides à travers différents espaces
- Premiers essais
- Corrections des modèles choisis
- Résultats actuels

Modèle du solide

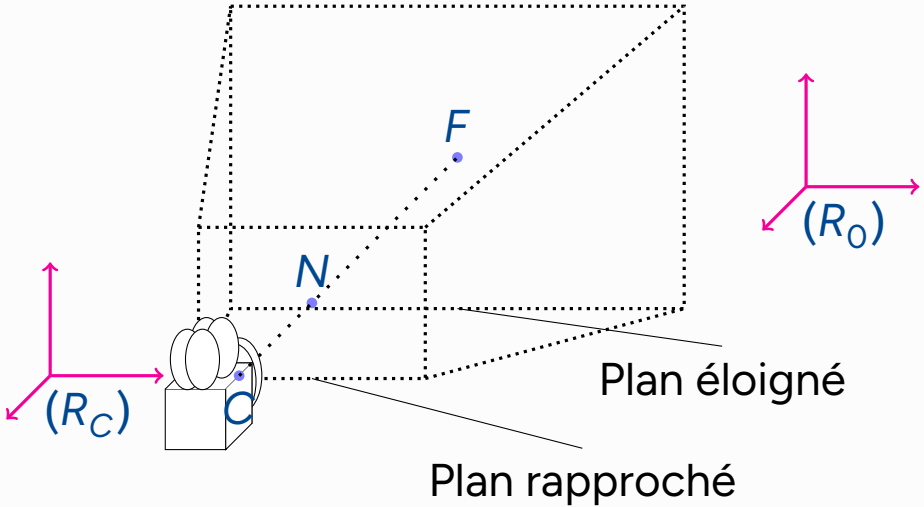


Sommet $(s_i) = (x, y, z, 1)$
Face $(f_i) = (s_1, \cdots, s_k)$
Solide $= ((s_1, \cdots, s_n), (f_1, \cdots, f_p))$

Coordonnées homogènes

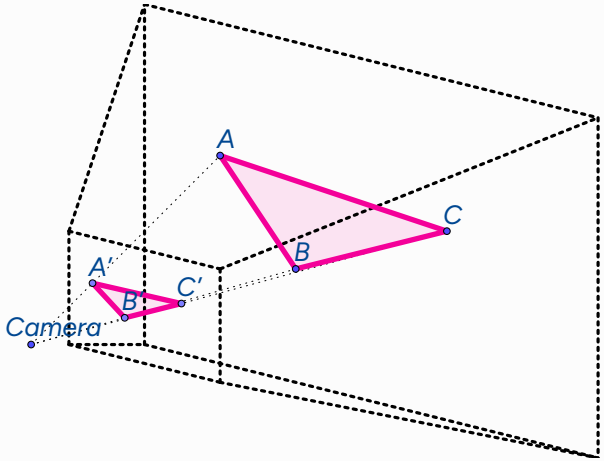
$\forall \alpha > 0$ on confond les points $(\alpha x, \alpha y, \alpha z, \alpha)$ et (x, y, z)

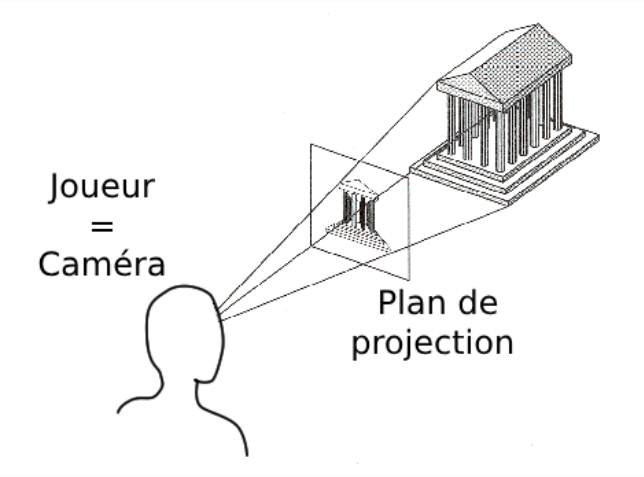
Modèle du joueur



- (R_C) : référentiel lié à la caméra
- (R_0) : référentiel ”absolu” lié à la scène

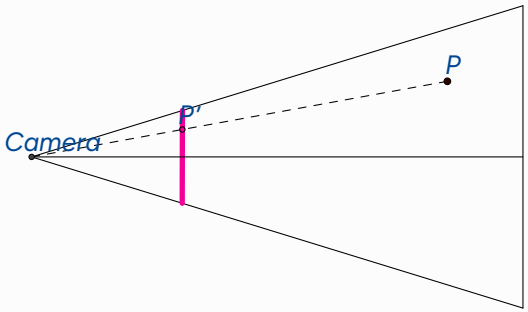
Projections





On chercher à déterminer la projection d'un objet de l'espace sur l'écran du joueur

Projections



Par argument de similitude de triangles :

$$\begin{cases} x' = z' \frac{x}{z} \\ y' = z' \frac{y}{z} \\ z' = z' \frac{z}{z} \end{cases}$$

Sous forme matricielle :

$$\begin{pmatrix} x' \\ y' \\ z' \\ 1 \end{pmatrix} \equiv \begin{pmatrix} x \\ y \\ z \\ \frac{z}{z'} \end{pmatrix} = \begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & \frac{1}{z'} & 0 \end{pmatrix} \begin{pmatrix} x \\ y \\ z \\ 1 \end{pmatrix}$$

Normalisation des coordonnées

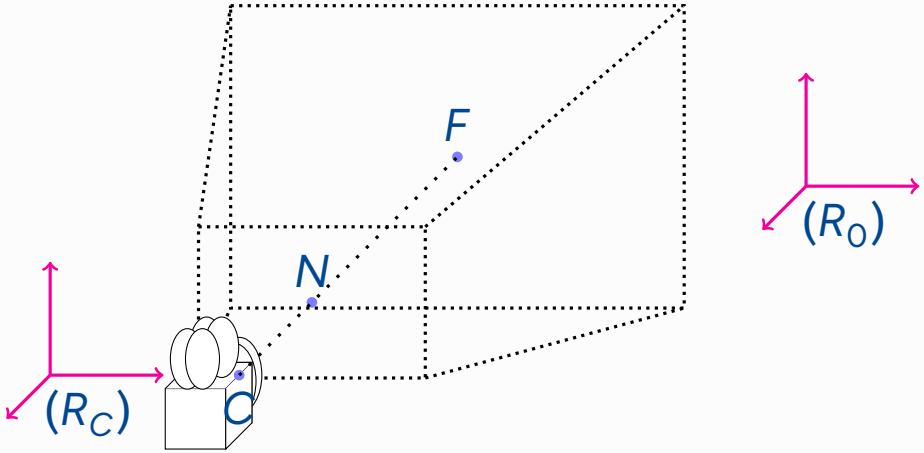
On ramène les coordonnées à l'intervalle $[-1, 1]$ en appliquant :

$$\begin{cases} [-X_m, X_m] \longrightarrow [-1, 1], x \longmapsto x/X_m \\ [-Y_m, Y_m] \longrightarrow [-1, 1], y \longmapsto y/Y_m \\ [N, F] \longrightarrow [-N, F], z \longmapsto \frac{F+N}{F-N}z - 2\frac{FN}{F-N} \end{cases}$$

Donc la projection devient :

$$\begin{pmatrix} x' \\ y' \\ z' \\ 1 \end{pmatrix} \equiv \begin{pmatrix} \frac{x}{X_m} \\ \frac{y}{Y_m} \\ \frac{F+N}{F-N}z - 2\frac{FN}{F-N} \\ z \end{pmatrix} = \begin{pmatrix} \frac{1}{X_m} & 0 & 0 & 0 \\ 0 & \frac{1}{Y_m} & 0 & 0 \\ 0 & 0 & \frac{F+N}{F-N} & \frac{-2FN}{F-N} \\ 0 & 0 & 1 & 0 \end{pmatrix} \begin{pmatrix} x \\ y \\ z \\ 1 \end{pmatrix}$$

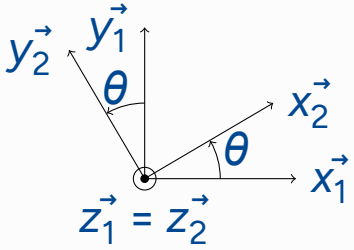
Changement de référentiels



On connaît les coordonnées dans le monde et non par rapport à la caméra

Matrice de rotation

- Dimension 2 :



$$R(\theta)=\begin{pmatrix}\cos\theta & -\sin\theta\\ \sin\theta & \cos\theta\end{pmatrix}$$

- Dimension 4 ($M = (x, y, z, 1)$):

$$R_x(\theta)=\begin{pmatrix}1 & 0 & 0 & 0\\ 0 & \cos\theta & -\sin\theta & 0\\ 0 & \sin\theta & \cos\theta & 0\\ 0 & 0 & 0 & 1\end{pmatrix} \quad R_y(\theta)=\begin{pmatrix}\cos\theta & 0 & \sin\theta & 0\\ 0 & 1 & 0 & 0\\ -\sin\theta & 0 & \cos\theta & 0\\ 0 & 0 & 0 & 1\end{pmatrix} \quad R_z(\theta)=\begin{pmatrix}\cos\theta & -\sin\theta & 0 & 0\\ \sin\theta & \cos\theta & 0 & 0\\ 0 & 0 & 1 & 0\\ 0 & 0 & 0 & 1\end{pmatrix}$$

Matrice de translation

La translation de vecteur $\vec{t} = (t_x, t_y, t_z)$:

$$T_{\vec{t}}(x, y, z) = (x + t_x, y + t_y, z + t_z) \equiv (x + t_x, y + t_y, z + t_z, 1)$$

L'intérêt des coordonnées homogènes :

$$\tilde{T}_{\vec{t}}(x, y, z) = \begin{pmatrix} x + t_x \\ y + t_y \\ z + t_z \\ 1 \end{pmatrix} = \begin{pmatrix} 1 & 0 & 0 & t_x \\ 0 & 1 & 0 & t_y \\ 0 & 0 & 1 & t_z \\ 0 & 0 & 0 & 1 \end{pmatrix} \begin{pmatrix} x \\ y \\ z \\ 1 \end{pmatrix}$$

Changement de référentiels

Position de la caméra : (x_c, y_c, z_c)
Orientation de la caméra : (ψ, θ, φ) (Angles d'Euler)
Passage :

1. Translation de $(-x_c, -y_c, -z_c)$ pour ramener la caméra à l'origine
2. Rotations de $-\psi$,$-\theta$ et $-\varphi$ pour ajuster l'orientation

Sous forme matricielle :

$$\begin{pmatrix} \cos(\psi) & \sin(\psi) & 0 & 0 \\ -\sin(\psi) & \cos(\psi) & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix} \begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & \cos(\varphi) & \sin(\varphi) & 0 \\ 0 & -\sin(\varphi) & \cos(\varphi) & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix} \begin{pmatrix} \cos(\theta) & 0 & -\sin(\theta) & 0 \\ 0 & 1 & 0 & 0 \\ \sin(\theta) & 0 & \cos(\theta) & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix} \begin{pmatrix} 1 & 0 & 0 & -x_c \\ 0 & 1 & 0 & -y_c \\ 0 & 0 & 1 & -z_c \\ 0 & 0 & 0 & 1 \end{pmatrix}$$

Premiers résultats

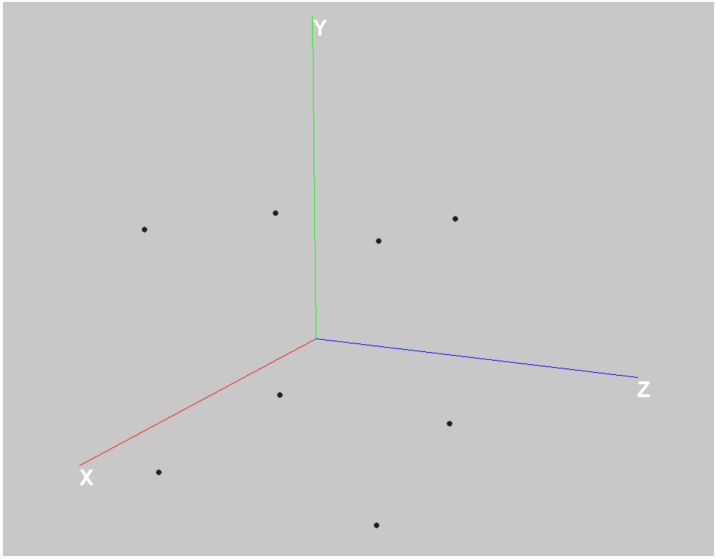


Figure: Sommets d'un cube

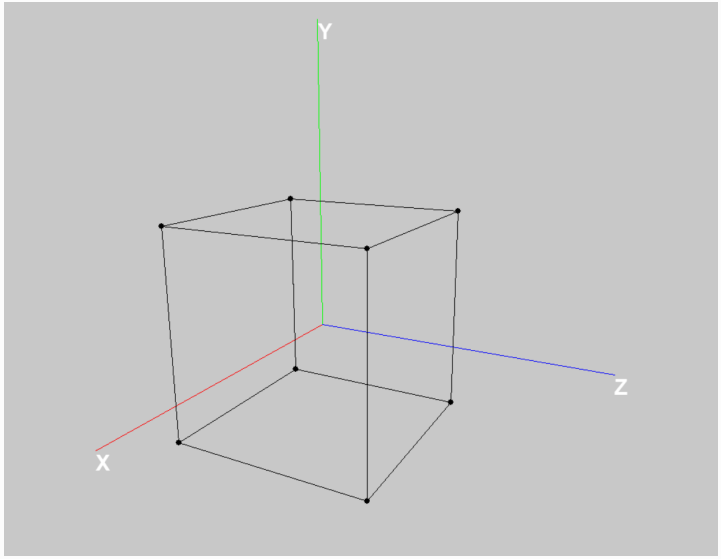


Figure: Modèle filaire du cube

Compatibilité format .OBJ (Structure du fichier)

```
mtllib Car.mtl
o Car_Cube
v -0.942557 0.399997 0.885192
v -0.742557 0.399997 0.885192
...
f 1316//2 1314//2 1410//2 1411//2
f 1329//3 1290//3 1502//3 1503//3
f 1295//4 1291//4 1498//4 1500//4
...
```

Compatibilité format .OBJ (Implémentation)

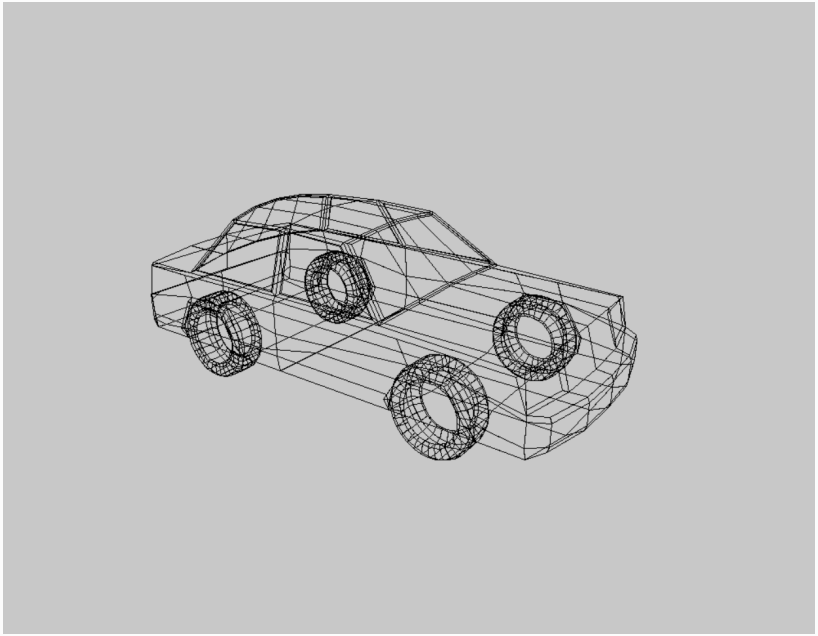


Figure: Voiture

Figure: Table de Ping Pong

Constat - Visibilité des faces cachées

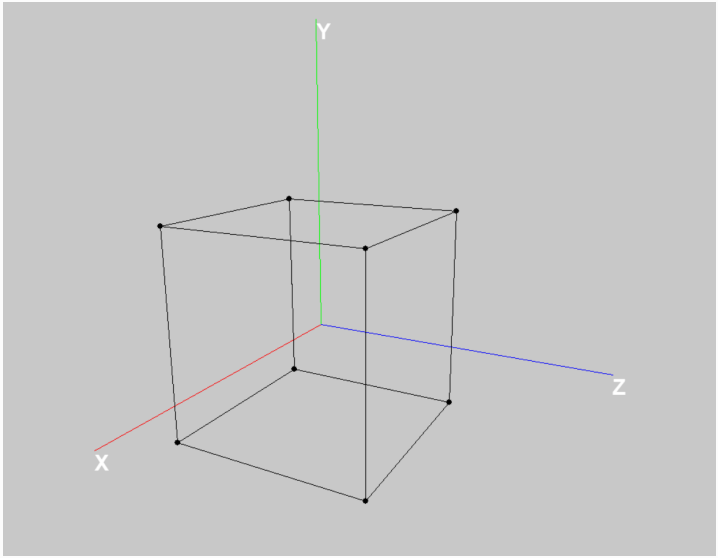


Figure: Faces arrières visibles

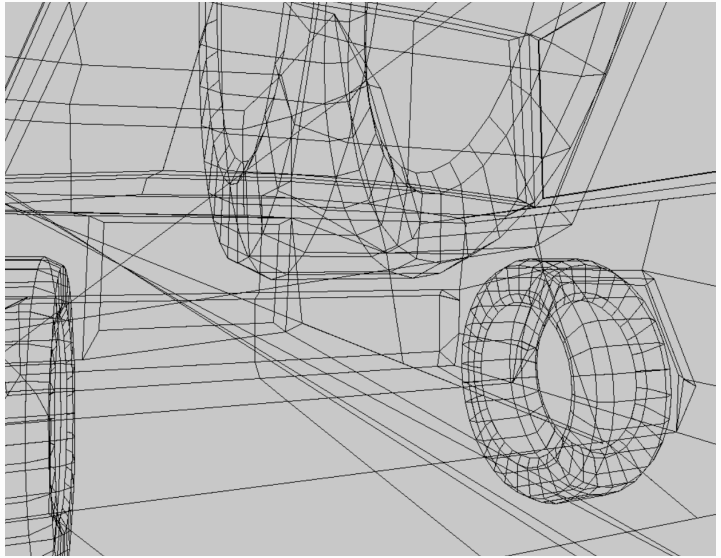
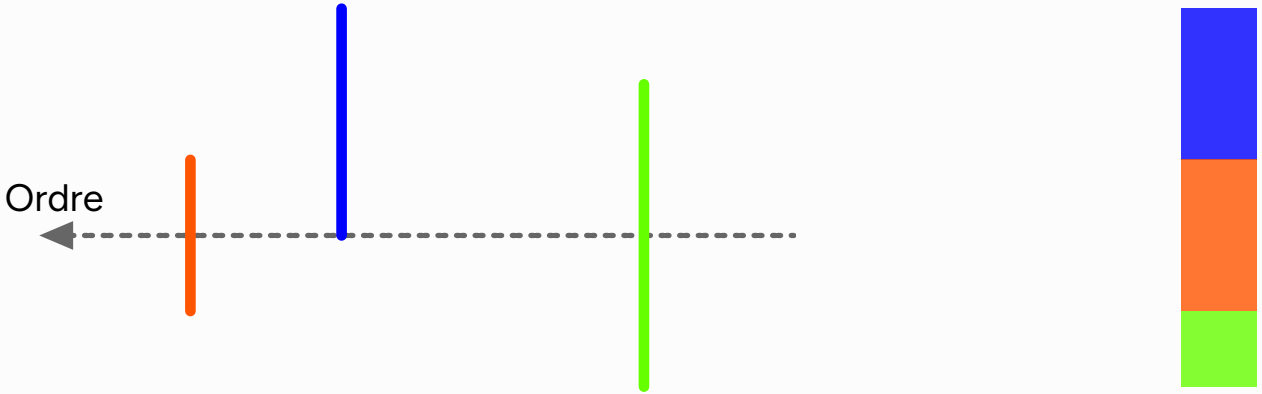


Figure: Rendu incohérent

Algorithme du peintre



Modifications

Ancien modèle

Sommet $(s_i) = (x, y, z, 1)$

Face $(f_i) = (s_1, \cdots, s_k)$

Solide $= ((s_1, \cdots, s_n), (f_1, \cdots, f_p))$

Modèle modifié

Sommet $(s_i) = (x, y, z, 1)$

Face $(f_i) = (s_1, \cdots, s_k)$

Couleur $(c_i) = (c_1, \cdots, c_k)$

Solide $= ((s_1, \cdots, s_n), (f_1, \cdots, f_p), (c_1, \cdots, c_p))$

Résultat

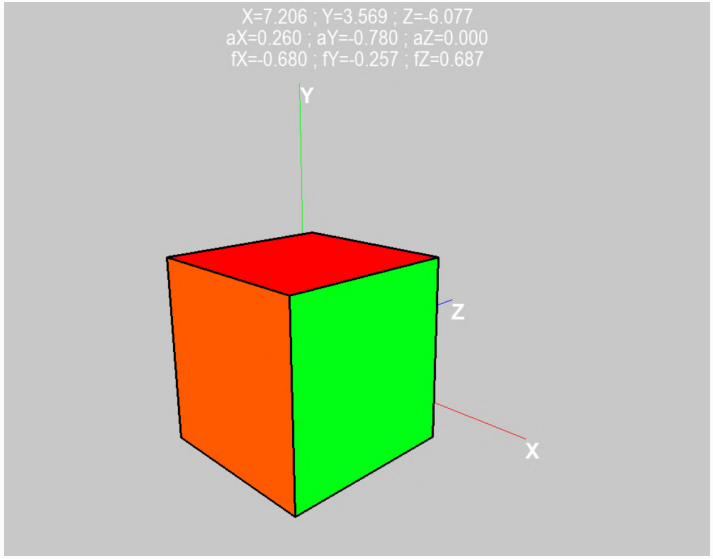


Figure: Couleurs du Rubik's cube

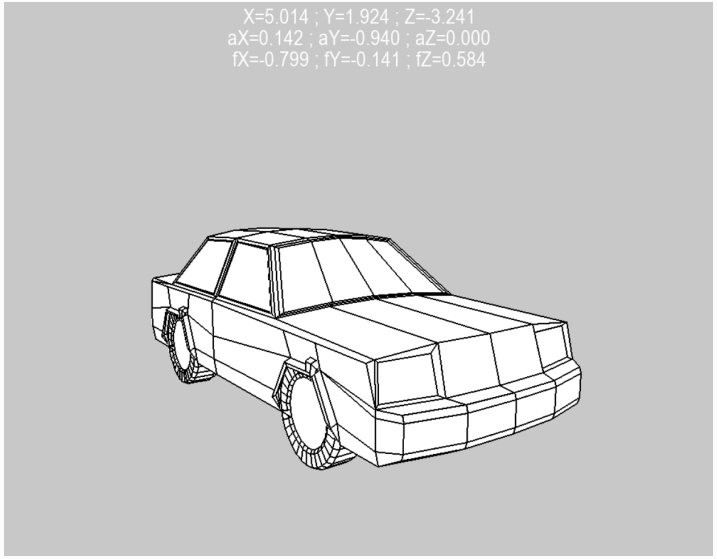
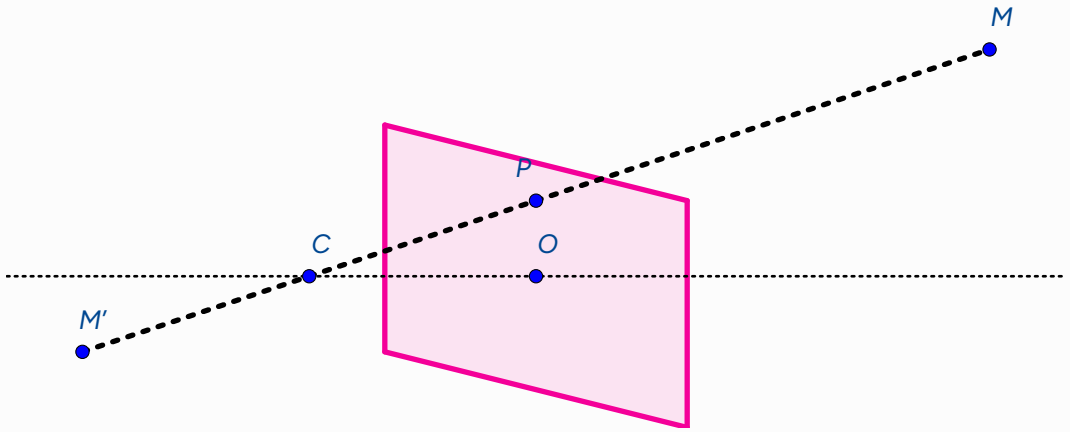


Figure: Couleur blanche par défaut

Détection des faces non visibles



Deux points symétriques par rapport à l'origine ont même projection sur l'écran

Rendu corrigé

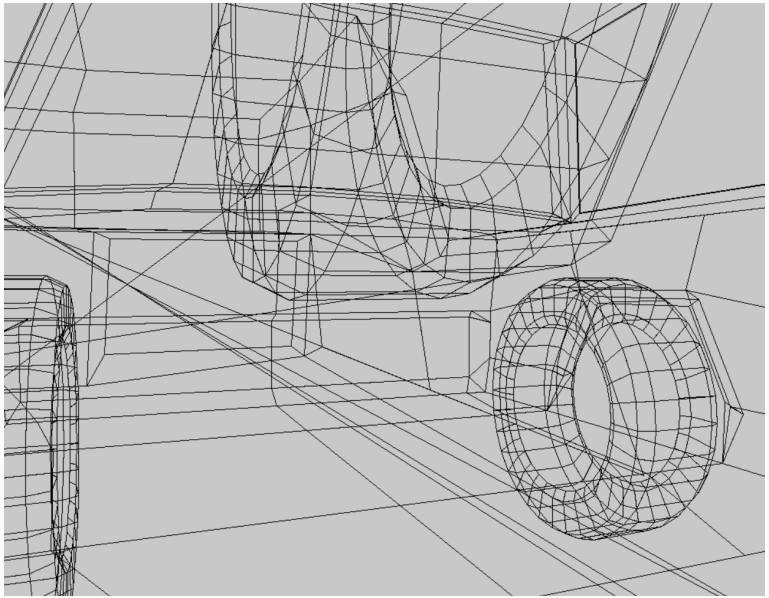


Figure: Avant correction

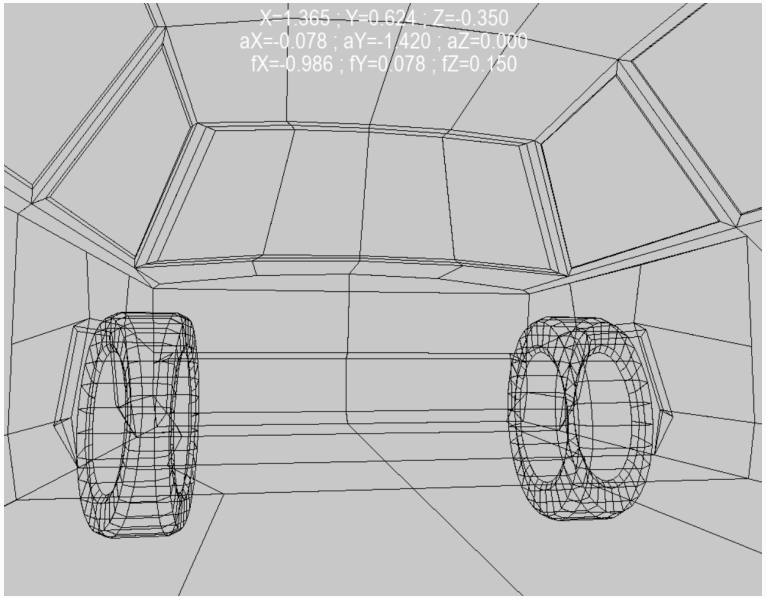


Figure: Après correction

Résultats actuels

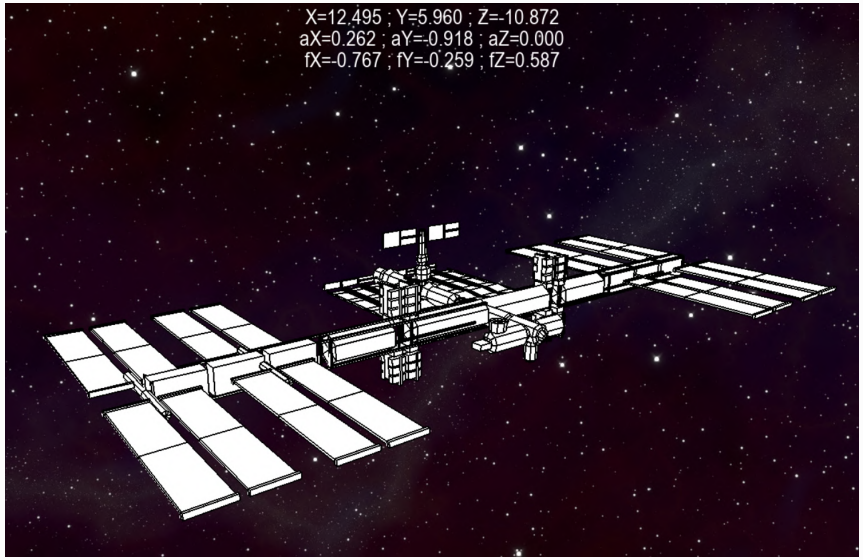


Figure: ISS by Poly by Google [CC-BY]

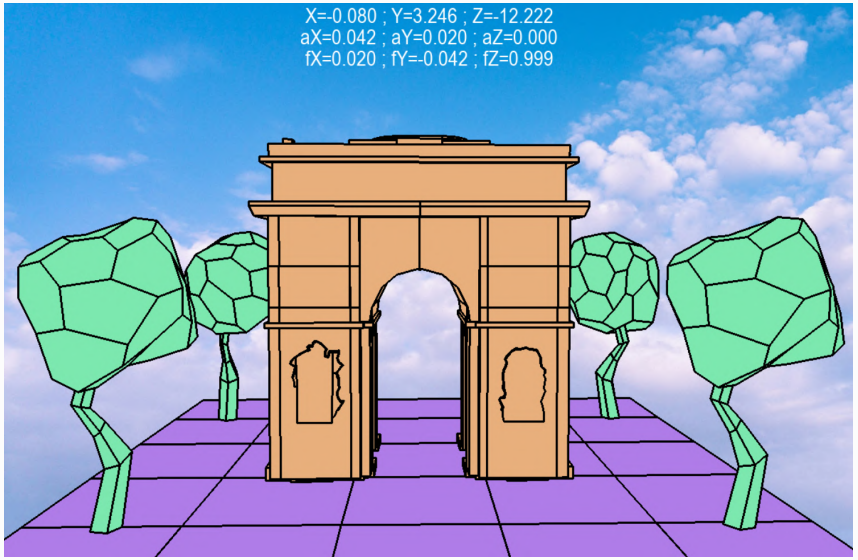


Figure: Scène avec plusieurs objets

Aspects à étudier

- Performance discutables

Rendu	Cube	Voiture	Arc de triomphe	ISS
Performances	≥60 IPS	32 IPS	30 IPS	≤20 IPS

- Solides déformables ?
- Texture des objets ?

Merci pour votre attention

Structure de programme

```
D:.
| camera.py              # Gestion du modèle caméra/joueur
| helper_functions.py    # Tri fusion, Extraction de modèle depuis un .OBJ, ...
| main.py
| matrix_functions.py    # Rotations, Translation, Homothétie
| object_3d.py           # Gestion du modèle du solide
| projection.py          # Matrices de projections caméra-ecran
| scene.py               # Assemblage des différents composants
|—— backups
|—— resources
        | modele.obj
        | background.jpg
```

main.py [1/2] (Images enregistrées avec CodeSnap sur VSCode)

```python
import pygame as pg
import numpy as np
from matrix_functions import *
```

```python
class Camera:
    def __init__(self, scene, position):
        self.scene = scene
        self.position = np.array([*position,1.0])
        self.anglePitch = self.angleYaw = self.angleRoll = 0

        self.forward = np.array([0,0,1,1])
        self.up = np.array([0,1,0,1])
        self.abs_up = np.array([0,1,0,1])
        self.right = np.array([1,0,0,1])

        self.h_fov = np.pi / 3
        self.near_plane = 0.01
        self.far_plane = 100

        self.moving_speed = 0.08
        self.rotation_speed = 0.02
```

main.py [2/2]

```python
def camera_yaw(self, angle):
    self.angleYaw += angle

def camera_pitch(self, angle):
    self.anglePitch += angle

def axiiIndentity(self):
    self.forward = np.array([0, 0, 1, 1])
    self.up = np.array([0, 1, 0, 1])
    self.right = np.array([1, 0, 0, 1])

def camera_update_axii(self):
    rotate = rotate_x(self.anglePitch) @ rotate_y(self.angleYaw)
    self.axiiIndentity()
    self.forward = self.forward @ rotate
    self.right = self.right @ rotate
    self.up = self.up @ rotate
    self.forward /= np.linalg.norm(self.forward[:3])
    self.right /= np.linalg.norm(self.right[:3])
    self.up /= np.linalg.norm(self.up[:3])
```

```python
def translation_matrix(self):
    x, y, z, w = self.position
    return translate([-x,-y,-z])

def rotation_matrix(self):
    rx = rotate_x(-self.anglePitch)
    ry = rotate_y(-self.angleYaw)
    rz = rotate_z(-self.angleRoll)
    return ry @ rx @ rz

def camera_matrix(self):
    self.camera_update_axii()
    return self.translation_matrix() @ self.rotation_matrix()
```

helper_functions[1/3]

```python
def merge(L, a, b, c):
    temp = L[a:b+1]
    i,j = 0, b+1
    for k in range(a,c+1):
        if j>c or (i < len(temp) and temp[i] < L[j]): L[k], i = temp[i], i+1
        else: L[k], j = L[j], j+1

def merge_aux(L,g,d):
    if d > g:
        m = (g+d)//2
        merge_aux(L,g,m)
        merge_aux(L,m+1,d)
        merge(L, g, m, d)

def merge_sort(L):
    merge_aux(L,0,len(L)-1)
```

Figure: Tri fusion

helper_functions[2/3]

```python
def vertexes_faces_from_obj(filename):
    f = open(filename, mode='r')
    vertexes = []
    faces = []
    for line in f:
        if line.startswith('v '):
            coords = line.split()[1:]
            vertexes.append([float(c) for c in coords] + [1.0])
        elif line.startswith('f '):
            vertices = line.split()[1:]
            faces.append([int(v.split('/')[0]) - 1 for v in vertices])
    f.close()
    return np.array(vertexes), faces
```

Figure: Exploitation des fichiers .OBJ

helper_functions[3/3]

```python
def distance(point1, point2):
    vec = [point1[i]-point2[i] for i in range(len(point1))]
    s = 0
    for composante in vec:
        s += composante**2
    return s**0.5

def distance_from_camera(point, camera):
    return distance(point, camera.position[:-1])

def calculate_depth(object, face):
    center = np.array([0,0,0], float)
    for index in face:
        center += object.vertexes[index][:-1]
    center /= len(face)
    return distance_from_camera(center, object.scene.camera)
```

Figure: Algorithme du peintre (Calcul des distances)

object_3d.py [1/3]

```python
class Object3D:
    def __init__(self, scene, vertexes='', faces='', colors=''):
        self.scene = scene

        self.vertexes = vertexes
        self.faces = faces
        self.colors = [pg.Color('white') for _ in self.faces] if colors=='' else [pg.Color(r,g,b) for r,g,b in colors]
        self.thickness = 1 if self.scene.render.wireframemode else 2

        # OBJECT SETTINGS
        self.movement_flag = False
        self.draw_vertexes = True if self.scene.render.wireframemode else False
        self.draw_edges = True
        self.backface_cull = self.scene.render.backface_cull

        self.font = pg.font.SysFont('Arial', 30, bold=True)
        self.label = ''
```

object_3d.py [2/3]

```
30 def z_sort_faces(self):
31     depths = [[0,i] for i in range(len(self.faces))]
32     for i in range(len(self.faces)):
33         depths[i][0] = calculate_depth(self, self.faces[i])
34     merge_sort(depths)
35     self.faces = [self.faces[i] for _,i in depths]
36     self.colors = [self.colors[i] for _,i in depths]
37
38 def draw(self):
39     self.z_sort_faces()
40     self.screen_projection()
```

object_3d.py [3/3]

```python
class Object3D:
    def __init__(self, scene, vertexes='', faces='', colors=''):
        self.scene = scene

        self.vertexes = vertexes
        self.faces = faces
        self.colors = [pg.Color('white') for _ in self.faces] if colors=='' else [pg.Color(r,g,b) for r,g,b in colors]
        self.thickness = 1 if self.scene.render.wireframemode else 2

        # OBJECT SETTINGS
        self.movement_flag = False
        self.draw_vertexes = True if self.scene.render.wireframemode else False
        self.draw_edges = True
        self.backface_cull = self.scene.render.backface_cull

        self.font = pg.font.SysFont('Arial', 30, bold=True)
        self.label = ''
```

scene.py [1/2]

```
5  class Scene:
6      def __init__(self, render, camera, objects = ''):
7          self.render = render
8          self.camera = camera
9          self.projection = Projection(self)
10         self.objects = [] if objects == '' else objects
11
12     def draw(self):
13         if not self.render.draw_background:
14             if self.render.darkmode: self.render.screen.fill(pg.Color(60, 60, 60))
15             else: self.render.screen.fill(pg.Color(200, 200, 200))
16         else:
17             self.render.screen.blit(self.render.background,(0,0))
18         for object in self.objects:
19             object.draw()
```

scene.py [2/2]

```
21 class DefaultScene(Scene):
22     def __init__(self, render):
23         camera = Camera(self, [3,3,-10])
24         super().__init__(render, camera, [])
25         if self.render.debugmode:
26             axes = Axes(self)
27             axes.translate([0.0001, 0.0001, 0.0001])
28             axes.scale(4)
29             self.objects.append(axes)
```