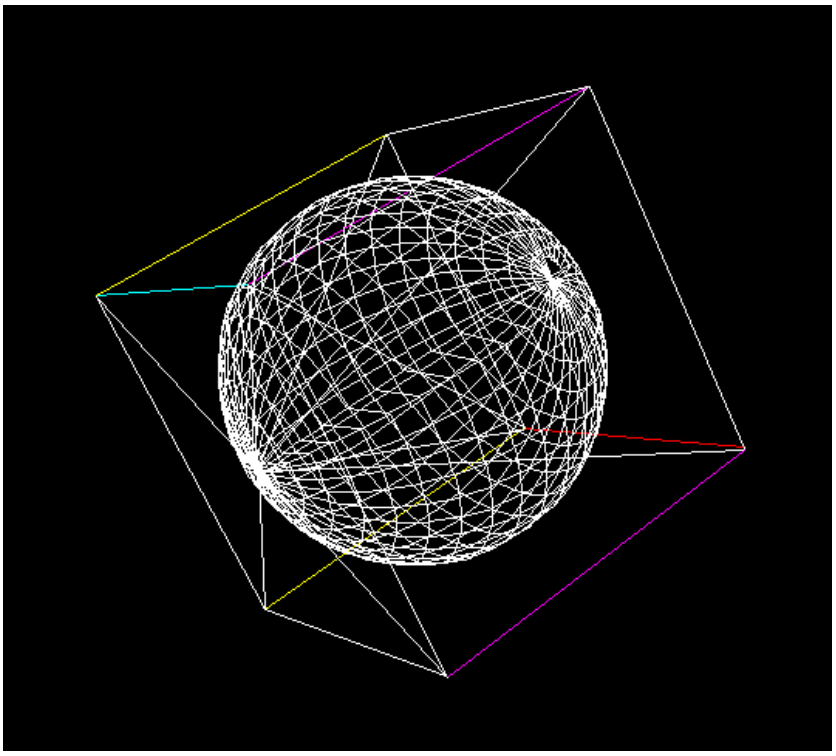
UNIVERSITÉ PARIS 8 VINCENNES-SAINT-DENIS

TD n°2 :

Transformations et Projections

par NAFFIEN Lucie



23 novembre 2021

Supervisé par M. M. Ammi

Table des matières

I. Introduction

Pouvoir animer une image 3D est essentiel en programmation graphique. Dans ce TD, réalisé dans le module “Programmation Graphique” du la 3ème annee de Licence ISEI (Informatique des Systèmes Embarqués et Interactifs) de l’Université Paris 8, nous allons effectuer la simulation d’un cube et d’une sphère tournant sur eux-même.

II. Transformation

Pour modifier la position d’un objet, il est nécessaire de calculer la nouvelle position de ses sommets. On peut utiliser pour cela des matrices de transformations respectant l’architecture suivante :

$$\begin{pmatrix} R_{11} & R_{12} & R_{13} & T_x \\ R_{21} & R_{22} & R_{23} & T_y \\ R_{31} & R_{32} & R_{33} & T_z \\ 0 & 0 & 0 & 1 \end{pmatrix}$$

Où **R** représente la matrice de rotation et **T** la matrice de translation.

Matrice de transformation

Le calcul de cette matrice a été réalisé dans la fonction `mat4 transformation_matrix(double pitch, double yaw, double roll, vec3 translation).`

A. Matrices de rotation

Afin de calculer la matrice de transformation de l’objet, il faut établir sa matrice de rotation, comme représentée ci-dessous :

$$R = R_x.R_y.R_z = \begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & \cos\theta_x & -\sin\theta_x & 0 \\ 0 & \sin\theta_x & \cos\theta_x & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix} \cdot \begin{pmatrix} \cos\theta_y & 0 & \sin\theta_y & 0 \\ 0 & 1 & 0 & 0 \\ -\sin\theta_y & 0 & \cos\theta_y & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix} \cdot \begin{pmatrix} \cos\theta_z & -\sin\theta_z & 0 & 0 \\ \sin\theta_z & \cos\theta_z & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix}$$

Matrice de rotation

Où **θ** représente la coordonnée x de l'objet pour la première matrice, y pour la seconde et z pour la dernière.

Elle a été implémenté dans le code de la façon suivante :

```
mat4 transformation_matrix(double pitch, double yaw, double roll, vec3
translation)
{
      // Calcul de la matrice de rotation
    double rad_pitch = pitch * (M_PI / 180);
    double rad_yaw = yaw * (M_PI / 180);
    double rad_roll = roll * (M_PI / 180);

      mat4 m_rotation_x;
      mat4 m_rotation_y;
      mat4 m_rotation_z;

      m_rotation_x[1][1] = cos(rad_pitch);
      m_rotation_x[2][1] = -sin(rad_pitch);
      m_rotation_x[1][2] = sin(rad_pitch);
      m_rotation_x[2][2] = cos(rad_pitch);

      m_rotation_y[0][0] = cos(rad_yaw);
      m_rotation_y[2][0] = sin(rad_yaw);
      m_rotation_y[0][2] = -sin(rad_yaw);
      m_rotation_y[2][2] = cos(rad_yaw);

      m_rotation_z[0][0] = cos(rad_roll);
      m_rotation_z[1][0] = -sin(rad_roll);
      m_rotation_z[0][1] = sin(rad_roll);
      m_rotation_z[1][1] = cos(rad_roll);


      mat4 m_transformation = m_rotation_x*m_rotation_y*m_rotation_z;
```

Les paramètres pitch, yaw et roll, représentant l'angle de rotation des coordonnées x, y et z, sont en radians. Or les matrices de rotations nécessitent des degrés. La première étape de cette fonction est donc de convertir ces radians en degrés à l'aide de la formule suivante :

$$degrés = radians * (\pi / 180)$$

La seconde étape consiste à initialiser les matrices de rotation pour chaque angle de rotation. Par définition, la structure mat4 remplit les matrices avec la matrice de référence. Remplacer les données aux coordonnées voulues est donc suffisant, et plus rapide lors de l'exécution que de redéfinir chaque coordonnées.

En troisième étape, ces 3 matrices sont multipliées pour obtenir la matrice de rotation finale.

B. Matrices de translation

Après avoir calculé la matrice de rotation, il faut rajouter les valeurs de celle de translation.

```
      // Ajout de la matrice de translation
      m_transformation[3][0]= translation.x;
      m_transformation[3][1]= translation.y;
      m_transformation[3][2]= translation.z;

    return m_transformation;
}
```

La matrice de transformation est ainsi obtenue et peut être retournée par la fonction.

III. Projection

L'objet que nous voulons dessiner est en 3 dimensions. Cependant, l'écran d'un ordinateur n'en possède que 2 : sa largeur et sa hauteur. Il est donc nécessaire de transposer l'objet sur un plan 2D avec une matrice de projection, représentée ci-dessous :

$$\begin{pmatrix} f & 0 & u_0 & 0 \\ 1 & f & v_0 & 0 \\ 1 & 1 & 1 & 1 \end{pmatrix}$$

Matrice de projection

Où **f** représente la **focale**, soit la distance entre le point de vue de l'utilisateur et le centre plan où l'objet doit être représentée, $\mathbf{u}_0$ la distance entre l'axe des ordonnées et le centre du plan et $\mathbf{v}_0$ la distance entre l'axe des abscisses et le centre du plan.

Celle-ci est calculée par la fonction *mat4x3 projection_matrix(double focal)*.

```
/* TD2, Exercice 2
 * projection_matrix : projecte l'objet 3D sur un plan 2D
 * Parametre : focal : double : focale de la matrice de projection
 * Retour : mat4x3 : matrice permettant la projection des sommets
 */
mat4x3 projection_matrix(double focal)
{
    mat4x3 m_projection;
    m_projection[0][0] = focal;
    m_projection[2][0] = window.get_width()*window.get_sample() / 2;
    m_projection[1][1] = focal;
    m_projection[2][1] = window.get_height()*window.get_sample() / 2;

    return m_projection;
}
```

La focale est donnée en paramètre et le centre de l'écran, soit sa taille divisée par 2, est utilisé pour positionner l'objet.

IV. Mise à jour des coordonnées de l'objet

```
/* TD2, Exercice 3
 * update_transformation : met a jour les coordonnes des vertices transformees
 * Parametre : m : mat4 : matrice de transformation
 */
void Object::update_transformation(mat4 m)
{
vertices_transformed.clear();
      //boucle pour chaque vertives
      for(unsigned int i = 0; i < vertices.size(); i++)
      {
      vec4 v_res = m * vertices[i];
      vertices_transformed.push_back(v_res);
      }
}
```

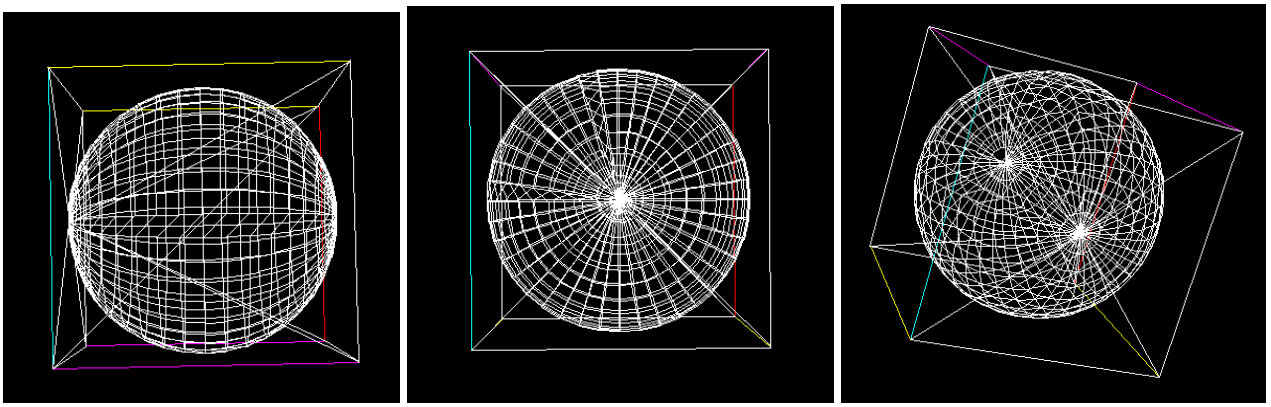
```
/* TD2, Exercice 4
 * update_projection : met a jour les coordonnes des vertices projetees
 * Parametre : m : mat4x3 : matrice de projection
 */
void Object::update_projection(mat4x3 m)
{
    vec3 v_out;
    vertices_projected.clear();

    for(unsigned int i = 0; i < vertices_transformed.size(); i++)
    {
      v_out = m * vertices_transformed[i];
      vertices_projected.push_back(vec2(v_out / v_out.z));
    }
}
```

V. Mise à jour de l'affichage

Avec les coordonnées de l'objets mises à jour, on peut finalement dessiner l'objet. Ici, on utilise le mode file de fer.

```
/* TD2, Exercice 5
 * draw : dessine l'objet
 * Parametres - window : Window & : fenetre devant afficher l'objet
 *            - light : vec4 :
 */
void Object::draw(Window & window, vec4 light)
{
      // Selection de la methode de dessin
    for(unsigned int i=0; i<faces.size(); i++)
    {
       switch(draw_method)
       {
            // mode fil de fer
             case DRAW_WIRE :
                   for(unsigned int x = 0; x < 4; x++)
                  {
                  vec2 p1 = vertices_projected[faces[i].vertex_index[x]];
                  vec2 p2 = vertices_projected[faces[i].vertex_index[(x + 1) %
4]];
                  window.draw_line(p1, p2, faces[i].color);
                  }
                   break;
             case DRAW_FILL :
                   // TODO => TP03 //
                   break;
             case DRAW_LAMBERT :
                   // TODO => TP04 //
                   break;
             case DRAW_GOURAUD :
                   // TODO => TP04 //
                   break;
             default :
                   break;
       }
    }
}
```

On obtient un cube et une sphère tournant sur eux-mêmes :



Résultat du programme