Transformations & Projections

Gitopoulos Giorgos, 9344

Computer Graphics @ ECE AUTh, April 2021

Abstract

This project deals with **transformations** of the view of an object in the **3D space**, as itself or the virtual camera moves or rotates and its **projection** in the **2D space** of a photography.

1. Introduction

The code of the project was written in *MATLAB* (can be found **here**) . Firstly, we implemented the *transformation_matrix* class, that contains an **Affine Transformation Matrix** that will be described in [2.1]. The class methods *rotate* and *translate* will be presented in [2.2] and [2.3] respectively. The transformation functions that used for object or system movement or rotation are described in [3], while the observer **perspective** and the object **projection** in the **2D space** are explained in [4]. Some display functions are presented in [5] and the way the program works in [6]. The output images of the implementation can be found in [7].

2. Affine Transformation

In order to represent the movement or the rotation of an object in the 3D space, we use the **Affine Transformation**, which is expressed by the formula

$$c_q = Tc_p \quad (1) \ ,$$

where $c_p = [p_x \ p_y \ p_z \ 1]^T$ is the augmented coordinate vector of the initial point p, $c_q = [q_x \ q_y \ q_z \ 1]^T$ the augmented coordinate vector of the transformed point q and T the transformation matrix.

2.1. Transformation Matrix

The transformation matrix that mentioned previously, was implemented as a class with a 4×4 matrix T as a variable. It contains two methods, *rotate* and *translate*. The first one, constructs the matrix T in order to rotate the vector c_p by a given angle θ around a given vector u, while the second one constructs the matrix T, so that it moves the vector c_p by a given vector t.

2.2. Rotation

In our implementation, we use the **Rodrigues' Rotation Formula** to rotate the vector c_p by θ (rads) around u. This formula is expressed by the matrix

$$R = (1 - cos\theta) \ \hat{u} \ \hat{u}^T + cos\theta \ I_3 + sin\theta \ [\hat{u}]_\times \quad (2) \ ,$$

where $\hat{u}$ is the 3×1 unit vector in the direction of u (normalized), I_3 is the 3×3 identity matrix and $[\hat{u}]_\times$ is the skew-symmetric matrix of vector $\hat{u}$. We use R matrix to define the transformation matrix as

$$T = \begin{bmatrix} R^T & 0_{3\times1} \\ 0_{1\times3} & 1 \end{bmatrix} \quad (3)$$

and from (1) and (3) we take the rotated product vector c_q. So, *rotate* method constructs the transformation matrix of (3).

2.3. Translation

To translate the vector c_p by $t = [t_x \ t_y \ t_z \ 0]^T$, we just need to add the two vectors:

$$c_q = \begin{bmatrix} p_x \\ p_y \\ p_z \\ 1 \end{bmatrix} + \begin{bmatrix} t_x \\ t_y \\ t_z \\ 0 \end{bmatrix} \quad (4)$$

We perform this operation using (1), after constructing the transformation matrix as follows:

$$T = \begin{bmatrix} 1 & 0 & 0 & t_x \\ 0 & 1 & 0 & t_y \\ 0 & 0 & 1 & t_z \\ 0 & 0 & 0 & 1 \end{bmatrix} \quad (5)$$

Method *translate* constructs the matrix T of (5).

3. Transformation Functions

3.1. Affine Transformation

The function *affine_transform* implements operation (1), but can be executed for a given set of points c_p and returns the transformation product (with a given matrix T) set of points c_q in non-augmented vectors ($3 \times num\_of\_points$ matrix).

3.2. Coordinate System Transformation

The function *system_transform* computes and returns the new coordinate vector d_p (non-augmented form) of a given point (with initial coordinate vector c_p), after rotating the coordinate system axes by a transformation matrix T. The new coordinates are computed by the formula

$$d_p = T^{-1} c_p \quad (6) \ .$$

4. Camera Perspective

In order to determine the 2D projection of an object in the camera system, we implemented the functions *project_cam* and *project_cam_ku*.

The first one, uses as arguments the unit coordinate vectors of the camera system in *WCS (World Coordinate System)* c_x, c_y, c_z, the camera coordinate vector c_v in *WCS*, the set of points p we want to project and the distance w between the camera lens and shutter. First of all, we have to compute the 3D coordinates of the given points in the camera system. The transformation matrix is defined as

$$T = \begin{bmatrix} R^T & -R^T c_v \\ 0_{1\times3} & 1 \end{bmatrix} \quad (7) \ ,$$

where

$$R = [c_x \; c_y \; c_z] \;\; (8) \; .$$

By applying (1), we find the set coordinates in the camera system. Assuming that x_p, y_p, z_p are the coordinates in the camera system of a point of the given set, we compute its perspective projection coordinates x_q, y_q, z_q using the following formulas:

$$x_q = \frac{wx_p}{z_p} \; , \;\; y_q = \frac{wy_p}{z_p} \; , \;\; z_q = 0 \;\; (8) \; ,$$

that come out of the geometry of the system. The 2D coordinates of all set points x_q and y_q are returned, so does the values of z_p, that express the depths of the points.
The function *project_cam_ku*, executes the exact same procedure with the previous function, but needs as arguments two direction vectors, that define the position of the camera, instead of c_x, c_y and c_z. Using them, we compute c_x, c_y and c_z vectors and call the *project_cam* function, which returns the projected coordinates and the depths of the given points p.

5. Display Functions

5.1. Rasterize

We assume that the 2D projection of an object in the camera, is stored as its points coordinates in a level of size $H \times W$ (inches) and $O(0,0)$ is exactly in its center. We need to transport the object coordinates to a $M \times N$ pixel matrix with $O(1,1)$ being the lower left pixel, in order to process it as an image. This procedure is implemented for every point p of a given set by *rasterize* function and is described by the following algorithm:

Algorithm 1 Rasterize Point

$c_{vertical} \leftarrow M/H$
$c_{horizontal} \leftarrow N/W$
$p_{rasterized}(1) \leftarrow round((p(1) + H/2) \; c_{vertical} + 0.5)$
$p_{rasterized}(2) \leftarrow round((p(2) + W/2) \; c_{horizontal} + 0.5)$

where p is the coordinate vector of a point in the $H \times W$ level (inches) and $p_{rasterized}$ is the coordinate vector of the same point in the $M \times N$ pixel matrix.

5.2. Render Object

The object is displayed using *render_object* function. This function uses as arguments a set of points p, the indices of the points that constitute triangles F, the colors of the points C, the pixel matrix dimensions M and N, the camera level dimensions H and W, the distance between camera lens and shutter w, the camera coordinate vector in *WCS* c_v and the camera position vectors c_{lookat} and c_{up}. Firstly, it calls *project_cam_ku* function that returns the 2D projection of the points P and their depths D . Then, the *rasterize* function is called using P in order to take the $M \times N$ pixel matrix P_{rast}, which is given with F, C and D in the *render* function of the **previous project**. The *render* function returns the painted image matrix I.

6. Code Functionality

In the *demo* file, we create an object of the class *transformation_matrix* and load the given workspace *hw2.mat*. The workspace contains the vectors $t_1 = [-15 \;\; 15 \;\; 3]^T$, $t_2 = [-10 \;\; 10 \;\; 10]^T$ and the angle $\theta = 1.5708 \; rads$. After taking a photo of the object in its **initial state** using *render_object* and saving it as $0.jpg$, we execute the following steps:

1. **Translate the object by** t_1, take a photograph of it using *render_object* and save it as $1.jpg$.
2. **Rotate the object by** θ, take a photograph of it using *render_object* and save it as $2.jpg$.
3. **Translate the object by** t_2, take a photograph of it using *render_object* and save it as $3.jpg$.

In each step, we update the matrix T of the *transformation_matrix* object (using *translate* or *rotate* respectively) and we use *affine_transformation* to transform the given set of points V, before calling *render_object*.

7. Results

The output images of the *demo* file are presented in Figure 1.

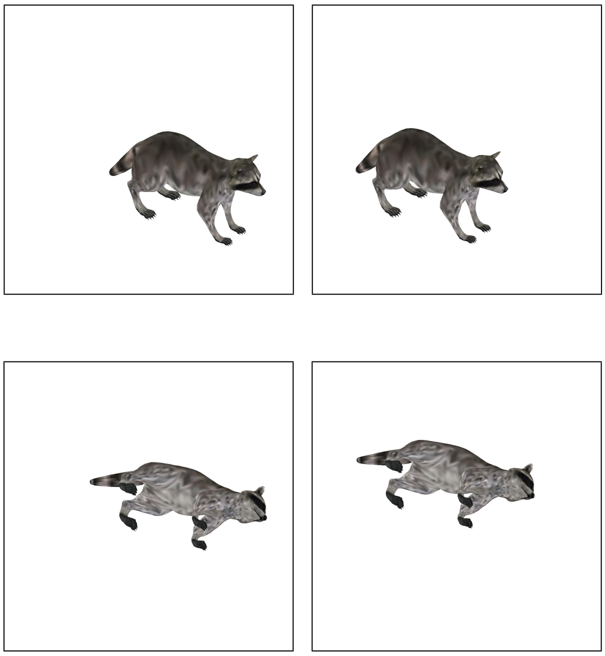

Figure 1: *Output images: initial state (up left), after step 1 (up right), after step 2 (bottom left), after step 3 (bottom right)*