Introduction to Computer Graphics & Polygon Filling

Petridis Konstantinos

Computer Graphics @ ECE AUTh, April 2022

Abstract

The major topic of this paper, is the the implementation and illustration of **Trasnsformations**, such as **translations** and **rotations** of a 3D object. The projection of the object from the 3D space, into the 2 dimensions, using rules of **Perspective Projection** is a crucial part of the process and is discussed in detail as well.

1. Introduction

In our previous, **Graphics** related project, we implemented a **triangle filling** algorithm, which assumes that the coordinates of every triangle, as well as the colors and coordinates of every 2D point are known, in order to assign color to every pixel, visualizing the object on an image. This however, is the final part of a **Graphics System**. Now we take one step back, to understand how we obtained the 2D coordinates of the object and how it was depicted in the way (angle and general position) we see it on the final image. In order to establish a complete understanding of the process followed by a **Graphics System**, we have to first define the problem we are trying to solve. After that, we will decompose it into smaller, easier to comprehend sub-problems and discuss each one separately.

2. Problem Definition

Given the matrices below:

- **verts3d**: $L \times 3$ matrix containing the 3D coordinates of every vertex. L vertices in total.
- **vcolors**: $L \times 3$ matrix containing the RGB colors of every vertex. L vertices in total.
- **faces**: $K \times 3$ matrix containing the indices from the vertices of every triangle. K triangles in total.

and the **coordinates** and **pointing axis** of the **camera**, depict the object as seen through the camera. After that, **translate** and **rotate** the object, each time visualizing the effect of the current transformation.

3. Problem Decomposition

3.1. Coordinate System Transformation

Initially, we have the 3D coordinates of the object, relative to **WCS** (World Coordinate System), which is basically the system with $\{\hat{x}, \hat{y}, \hat{z}\}$ axes and $(0, 0, 0)$ as the start. The first thing that needs to be done, is transfer this coordinate system, to match the coordinate system formed by the camera. This can be obtained through an **Affine Transformation**, since if we rotate and translate WCS appropriately, it will eventually match the camera coordinate system. In order to achieve this, we compute the camera axes, since we know its position, pointing direction and the up-vector. Let L_h be the **transformation**, which, if applied to the initial coordinate system, results in the **camera system**. We know that this matrix includes a rotation and a translation. The rotation matrix is the Rodrigues matrix and is computed as follows:

$$R = \begin{bmatrix} \hat{x}_c & \hat{y}_c & \hat{z}_c \end{bmatrix}$$

$\hat{x}_c, \hat{y}_c, \hat{z}_c$ being the axes of the camera system. What happens however, to the coordinates of the object itself? If we visualize the object, as immovable points in 3D space, rotating the axis so that it matches the camera system, is equivalent to rotating every point in the opposite direction, keeping the axes static. It's easily comprehensible that in this case, the new coordinates of the object can be obtained by

$$p_c = R^{-1}p$$

where p are the initial 3D coordinates, relative to WCS. After rotating the coordinate system, we have to translate it, so that its new start of axes is identical to the camera's position. This means that the total transformation applied on the object's coordinates is given by:

$$p_c = R^{-1}p + c_v$$

c_v being the coordinates of the camera. In order to take advantage of the optimized Numpy multiplication operations which use vectorization, we include the above transformations in a single, homogeneous matrix L_h, so that the transformation is expressed as

$$p_c = L_h p$$

where

$$L_h = \left[\begin{array}{c|c} & c_{v,x} \\ \mathrm{R}^{-1} & c_{v,y} \\ & c_{v,z} \\ \hline 0_{1\times 3} & 1 \end{array}\right]$$

The above process can be summarized in the following subroutine.

Algorithm 1 Coordinate System Transformation

```
1: procedure SYSTEMTRANSFORM
2:     input : $p$                                    ▷ 3D coordinates
3:     $\hat{x}_c\hat{y}_c\hat{z}_c \leftarrow$ compute camera axes
4:     $L \leftarrow$ compute $L$ as above
5:     $p = Lp$
6:     return $p$
```

3.2. Projection from 3D to 2D

At this point, we have the coordinates of our object, relative to the coordinate system of the camera. The next step, is to apply the rules of **Perspective Projection** and obtain the projected, 2D coordinates from the 3D object.

We apply the following formula, to every 3D point of the object:

$$\begin{bmatrix} x_{proj} \\ y_{proj} \\ 1 \end{bmatrix} = f \begin{bmatrix} x/z \\ y/z \\ z/z \end{bmatrix}$$

where f is given. The z value before the projection, is kept, as it expresses the depth of a point, which is needed when coloring the triangles in later stages.

Now we have obtained the 2D coordinates, as projected onto the camera. However, we need to transfer those coordinates to correspond to the frame of the image, which is significantly larger than the camera's curtain. So before running the coloring routine from the previous project, we apply rasterization and we finally have the 2D object, ready to be filled with color.

Algorithm 2 Projection from 3D to 2D

```
1: procedure PROJECTCAMERA
2:     input : p                    ▷ 3D coordinates, relative to camera system
3:     $D = [\,]$                   ▷ Depth of points
4:     for point in p do
5:         $x = f\ x/z$
6:         $y = f\ y/z$
7:         $D.append(z)$
8:     $P_{rast} \leftarrow rasterizeCoordinates()$
9:     return $P_{rast}$, $D$
```

3.3. Rendering

Now, we have computed everything we need to render the object and display it. We call the **render** function from the previous project, using the 'gouraud' configuration and the image is successfully displayed, as shown below.

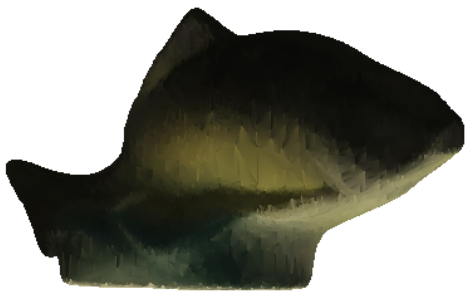

Figure 1: *Image, using gouraud shading*

The entire process is described by *Algorithm 3*

Algorithm 3 Change coordinates, project and render

```
1: procedure RENDEROBJECT
2:     input : verts3d                          ▷ 3D coordinates relative to WCS
3:     $p \leftarrow$ SYSTEMTRANSFORM()          ▷ relative to camera
4:     $P_{rast},\ D \leftarrow$ PROJECTCAMERA()  ▷ 3D to 2D
5:     $Img \leftarrow$ RENDER()                 ▷ Color the image
6:     return $Img$
```

3.4. Translation

The first transformation to be applied, is a translation, which is algebraically interpreted as simply adding to the coordinates of the object, an arbitrary value, depending on how much we want the object to moved across each axis. The linear formulation of the transformation is

$$p_{transl} = \begin{bmatrix} x \\ y \\ z \end{bmatrix} + \begin{bmatrix} t_x \\ t_y \\ t_z \end{bmatrix}$$

or, in homogeneous form

$$p_{transl} = \left[\begin{array}{c|c} \mathrm{I}_3 & \begin{matrix} t_x \\ t_y \\ t_z \end{matrix} \\ \hline 0_{1\times 3} & 1 \end{array}\right] \begin{bmatrix} x \\ y \\ z \end{bmatrix}$$

If we apply this transformation to every point of the object, we should observe that the new object has moved along some axes.

3.5. Rotation

The second transformation, is a rotation, around a specified rotation axis. Rotating an object, translates to multiplying its coordinates, with a type **SO(3)** matrix, which is called the **Rodrigues Rotation Matrix**

$$R = R(u, \theta)$$

u being the rotation axis and θ the angle of rotation. The **Rodriguez Matrix** is computed by the formula:

$$R = (1 - \cos\theta)uu^T + \cos\theta I_3 + \sin\theta \begin{bmatrix} 0 & -u_z & u_y \\ u_z & 0 & -u_x \\ -u_y & u_x & 0 \end{bmatrix}$$

A rotation transformation can be expressed as either

$$p_{rot} = Rp$$

or, in homogeneous form

$$p_{rot} = \left[\begin{array}{c|c} \mathrm{R} & \begin{matrix} 0 \\ 0 \\ 0 \end{matrix} \\ \hline 0_{1\times 3} & 1 \end{array}\right] \begin{bmatrix} x \\ y \\ z \end{bmatrix}$$

3.6. Applying the transformations

The process to accomplish every type of transformation is demonstrated in the following algorithm.

Algorithm 4 Transform object

```
1: procedure TRANSFORM
2:     input : p                    ▷ 3D coordinates relative to WCS
3:     $L \leftarrow$ Compute homogeneous transformation matrix
4:     $p \leftarrow Lp$            ▷ Transform the coordinates appropriately
5:     $Img \leftarrow$ RENDEROBJECT()
6:     return $Img$
```

4. Results

Now that the analysis is complete and the algorithmic procedure has been established, it's time to exhibit the results from the testing.

4.1. Translation

For the first translation, we used the vector:

$$t_1 = \begin{bmatrix} 0 \\ 0 \\ -5000 \end{bmatrix}$$

This means that the object is only moved towards the z axis. This is why it appears to be significantly smaller in *Figure 2*, compared to *Figure 1*, as it was dragged further from our point of view (camera), in the direction of z.

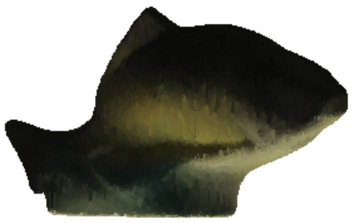

Figure 2: *Object after translation across z axis*

4.2. Rotation

After the first translation, we are ready to apply a rotation. The rotation axis used, is

$$u = \begin{bmatrix} 0 \\ 1 \\ 0 \end{bmatrix}$$

which is basically the y axis. Therefore, we expect our object to rotate, following a horizontal arc around the vertical axis of the image. As depicted in *Figure 3*, the object rotated as expected and eventually came closer to us, thus appears larger.

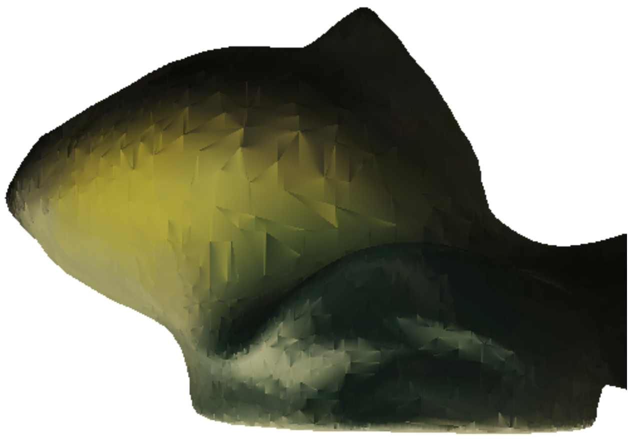

Figure 3: *Object after rotaion around y axis*

4.3. Translation

The final transformation is a translation using

$$t_1 = \begin{bmatrix} 0 \\ 500 \\ -10000 \end{bmatrix}$$

The object is expected to be shifted upwards, across y axis, and appear smaller, as once again we moved it towards the z axis. This is verified by *Figure 4*.

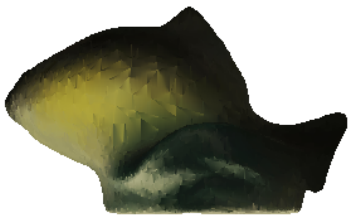

Figure 4: *Object after translation across y and z*

5. Code Instructions

There is one **demo** file called, named **demo.py**, under the 'src' directory. All functions used for reading, coloring, coordinate system change, rasterization, projection as well as the implementations for the Affine transformations (rotations and translations), are located inside 'inc' directory. Finally, the file containing all the necessary data is placed under the 'data' directory. Everything is set to run automatically and it's not a necessity for the user to interfere. Be sure to check out the **README** for direct instructions as well. The main library used is **Numpy** for the computations and data structures that it provides.