Computer Graphics Labs

Abel J. P. Gomes

LAB. 2

Department of Computer Science and Engineering
University of Beira Interior
Portugal
2020

LAB. 2

GEOMETRIC TRANSFORMATIONS

1. Learning goals
2. Getting started
3. Euclidean transformations
4. Affine transformations
5. 3D transformations in OpenGL
6. Example
7. Programming exercises
8. Final remarks

Lab. 2

GEOMETRIC TRANSFORMATIONS

In this lecture, we are going to deal with geometric transformations in 2D as their generalization in 3D is straightforward. These geometric transformations are also called affine transformations.

1. Learning Goals

At the end of this chapter **you should be able to**:

1. Explain what transformations are and why we use them in computer graphics.
2. List the three main transformations we use in computer graphics and describe what each one does.
3. Understand how to rotate a point around an arbitrary point.
4. Understand what homogenous coordinates are and why we use them in computer graphics.
5. Understand the importance of the order of operations in a matrix multiplication expression.
6. Understand what a CTM (Combined Transformation Matrix) is and understand what order the transformations must be in to achieve the desired CTM.
7. Be aware of the default facilities of OpenGL; for example, the default 2D domain is OpenGL is $[-1,1]\times[-1,1]$.

2. Getting Started

Geometric transformations are used to fulfill two main requirements in computer graphics:

1. To **model** and construct scenes.
2. To **navigate** our way around 2- and 3- dimensional space.

For example, when a street building has *n* identical windows, we proceed as follows:

1. To construct a single window by means of graphics primitives;
2. To replicate *n* times the window.
3. To put each window at a desirable location using translations and rotations.

This shows that transformations such as translations and rotations can be used as **scene modeling** operations.

These transformations can be also used to move a bot or an avatar in the virtual environment of a First-Person Shooter (FPS) game.

3. Euclidean Transformations

There are two Euclidean transformations:

1. Translation
2. Rotation

3.1. Translation

Translation can be thought of as moving something. In translation, a point is moved a distance in a direction.

For example, when the point $A(x, y)$ is translated dx units in the x direction and dy units in the y direction, it becomes:

$$A'(x + dx, y + dy)$$

or, equivalently,

$$\begin{cases} x' = x + dx \\ y' = y + dy \end{cases}$$

Representing points as column matrices, we obtain

$$A = \begin{bmatrix} x \\ y \end{bmatrix}, \qquad A' = \begin{bmatrix} x' \\ y' \end{bmatrix}, \qquad \text{and} \qquad T = \begin{bmatrix} dx \\ dy \end{bmatrix},$$

so that the translation can be expressed as follows:

$$A' = A + T$$

In general, translating an object means to translate its vertices (i.e. corners or endpoints) in such a manner that lines or polygons can then be drawn using the transformed vertices.

3.2. Rotation about the origin

Using polar coordinates (r, ϕ), a given point in the plane is given by the following equations:

$$\begin{cases} x = r\cos\phi \\ y = r\sin\phi \end{cases}$$

By default, rotating an object by the angle θ means rotating it around the origin by θ. After rotating the previous point by the angle θ around the origin, the get the following transformed point:

$$\begin{cases} x' = r\cos(\phi + \theta) \\ y = r\sin(\phi + \theta) \end{cases}$$

or

$$\begin{cases} x' = r\cos\phi\cos\theta - r\sin\phi\sin\theta) \\ y' = r\cos\phi\sin\theta + r\sin\phi\cos\theta) \end{cases}$$

that is

$$\begin{cases} x' = x\cos\theta - y\sin\theta \\ y' = x\sin\theta + y\cos\theta \end{cases}$$

In matrix notation, we then obtain

$$\begin{bmatrix} x' \\ y' \end{bmatrix} = \begin{bmatrix} \cos\theta & -\sin\theta \\ \sin\theta & \cos\theta \end{bmatrix} \begin{bmatrix} x \\ y \end{bmatrix}$$

or

$$A' = R.A$$

where R is the 2x2 rotation matrix.

3.3. Homogeneous Coordinates

We've seen the following matrix transformations:

Translation:

$$A' = A + T$$

Rotation:

$$A' = R.A$$

Translation is achieved through matrix addition while rotation is achieved by matrix product. This means that we can combine any number of translation matrices through addition, and any number of rotation matrices through multiplication. However, we cannot combine translation and rotation matrices into a single matrix through the product operation. It would be very useful if we could do this because that would enable the composition of geometric transformations through a single matrix operation, say matrix product. Besides, it would be less computationally expensive, as explained below.

Homogenous Coordinates are just a way to overcome this problem. With homogenous coordinates, a series of geometric transformations can be applied in a sequence using matrix product. The result is usually called combined transformation matrix or CTM.

Therefore, translations and rotations expressed in homogenous coordinates are given by:

Translation:

$$A' = T.A$$

Rotation:

$$A' = R.A$$

In homogenous coordinates a point $P(x, y)$ is represented by the homogenous point $P(X, Y, W)$ where:

$$X = \frac{x}{w} \qquad \text{and} \qquad Y = \frac{y}{w},$$

where W usually equals 1 in computer graphics for simplicity.

Using homogenous coordinates, the Euclidean transformation matrices are expressed as 3x3 matrices as follows:

Translation:

$$T(dx, dy) = \begin{bmatrix} 1 & 0 & dx \\ 0 & 1 & dy \\ 0 & 0 & 1 \end{bmatrix}$$

Rotation:

$$R(\theta) = \begin{bmatrix} \cos\theta & -\sin\theta & 0 \\ \sin\theta & \cos\theta & 0 \\ 0 & 0 & 1 \end{bmatrix}$$

3.4. Rotation about an arbitrary point

The rotation matrix (above) works well if we intend to rotate a point around the origin. But, what about rotating the point (x, y) around the arbitrary point (x_a, y_a)?

The answer lies in the following procedure of three steps:

- Translate (x_a, y_a) to the origin, i.e. translate by $T(-x_a, -y_a)$.
- Perform the rotation $R(\theta)$.
- Translate so that point at the origin returns to the original location, i.e., translate by $T(x_a, y_a)$.

Therefore, to rotate an object made up of 5 vertices, each geometric transformation would need to be done 5 times. Overall, we have

3 Transformations x 5 Vertices = 15 calculations

what is computationally expensive.

The computational cost can be reduced using the CTM (Combined Transformation Matrix), i.e. by combining the transformations into a single CTM:

3 Transformations x Identity Matrix = 3 calculations

1 Transformation (CTM) x 5 Vertices = 5 calculations

so that the total number of calculations is equal to 8.

3.5. Order and composition of transformations

The order of geometric transformations of the CTM is relevant because the matrix product is not commutative. In fact,

Matrix product is associative:

When multiplying matrices, the order we carry out the multiplications is not relevant, that is

$$A.B.C = (A.B).C = A.(B.C)$$

Matrix multiplication is not commutative:

When multiplying matrices together, we carry out the multiplications is relevant, that is

$$A.B \neq B.A$$

The question is then how do we work out the order of our matrices when creating the CTM? Turning back to the steps to rotate a point around the point (x_a, y_a), let us rewrite the corresponding procedure:

- Translate (x_a, y_a) to the origin, i.e. translate by $T(-x_a, -y_a)$.
- Perform the rotation $R(\theta)$.
- Translate so that point at the origin returns to the original location, i.e., translate by $T(x_a, y_a)$.

Thus, the order of the CTM is:

$$CTM = T(x_a, y_a).R(\theta).T(-x_a, -y_a)$$

When we multiply the CTM by the point P we have

$$CTM.P = T(x_a, y_a).R(\theta).T(-x_a, -y_a).P$$

An important fact to bear in mind is that the transformation closest to the point P in the expression is the first transformation to be applied to P.

4. Affine Transformations

Euclidean transformations preserve the **distance** between points, and because of that they are then called rigid transformations.

Affine transformations generalize Euclidean transformations in the sense that they don not preserve distance but **parallelism** instead. This means that two parallel lines remain parallel after applying a affine transformation. Because of this principal invariant, other properties are preserved. For example, an affine transformation also preserves collinearity (i.e., all points of a line remain on a line after transformation) and ratios of distances or proportions (e.g., the midpoint of a line segment remains the midpoint after transformation).

An affine transformation is also called an affinity. Examples of affine transformations are contraction, expansion, dilation, reflection, rotation, shear, similarity transformations, spiral similarities, and translation, as are their combinations. In general, an affine transformation is the result of a composition of rotations, translations, dilations, and shears.

As seen above, rotations and translations are Euclidean transformations. Let us then see the other two basic affine transformations.

Dilation or Scaling:

In scaling, we change the size of an object. Scaling makes an object bigger or smaller in the x and/or y direction.

Scaling a point (x, y) by a factor s_x along the x axis and s_y along the y axis requires we multiply each coordinate by the corresponding scaling factor:

$$\begin{cases} x' = s_x . x \\ y' = s_y . y \end{cases}$$

or, using the matrix notation,

$$\begin{bmatrix} x' \\ y' \\ 1 \end{bmatrix} = \begin{bmatrix} s_x & 0 & 0 \\ 0 & s_y & 0 \\ 0 & 0 & 1 \end{bmatrix} . \begin{bmatrix} x \\ y \\ 1 \end{bmatrix}$$

Shearing:

Shearing enjoys the property that all points along a given line l remain fixed, while other points are shifted parallel to l by a distance that is proportional to their perpendicular distance from l. Note that shearing an object in the plane does not change its area at all. As a margin note, let us say that shearing can easily be generalized to three dimensions, where planes are translated instead of lines.

Shearing a point (x, y) by a factor h_x along the x axis and h_y along the y axis is given by the following equations:

$$\begin{cases} x' = x + h_x . y \\ y' = y + h_y . x \end{cases}$$

or, using the matrix notation,

$$\begin{bmatrix} x' \\ y' \\ 1 \end{bmatrix} = \begin{bmatrix} 1 & h_x & 0 \\ h_y & 1 & 0 \\ 0 & 0 & 1 \end{bmatrix} . \begin{bmatrix} x \\ y \\ 1 \end{bmatrix}$$

The effect of a shearing looks like “pushing” an object in a direction that is parallel to a coordinate axis in 2D (or coordinate plane in 3D). Note that we can do this only in the x-direction as follows

$$\begin{cases} x' = x + h_x . y \\ y' = y \end{cases}$$

or in the y-direction

$$\begin{cases} x' = x \\ y' = y + h_y . x \end{cases}$$

5. 3D Transformations in OpenGL

In modern OpenGL, geometric transformations are defined using GLM (OpenGL Mathematics) and GLSL (OpenGL Shading Language) as follows:

$$\underset{\text{myMatrix}}{\begin{bmatrix} a & b & c & d \\ e & f & h & i \\ j & k & l & m \\ n & o & p & q \end{bmatrix}} \times \underset{\text{myVector}}{\begin{bmatrix} x \\ y \\ z \\ w \end{bmatrix}} = \underset{\text{transformedVector}}{\begin{bmatrix} ax + by + cz + dw \\ ex + fy + hz + iw \\ jx + ky + lz + mw \\ nx + oy + pz + qw \end{bmatrix}}$$

In C++, with GLM:

```
glm::mat4 myMatrix;
glm::vec4 myVector;
// fill myMatrix and myVector somehow
glm::vec4 transformedVector = myMatrix * myVector; // Again, in this order ! this is important.
```

In GLSL :

```
mat4 myMatrix;
vec4 myVector;
// fill myMatrix and myVector somehow
vec4 transformedVector = myMatrix * myVector; // Yeah, it's pretty much the same than GLM
```

<u>Translation</u>:

In GLM, the translation is defined as follows:

```
glm::mat4 glm::translate(
        glm::mat4 const & m,
        glm::vec3 const & translation);
```

which transforms a matrix with a translation 4x4 matrix m created from 3 scalars.

Let us see an example:

$$\begin{bmatrix} 1 & 0 & 0 & 10 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \times \begin{bmatrix} 10 \\ 10 \\ 10 \\ 1 \end{bmatrix} = \begin{bmatrix} 20 \\ 10 \\ 10 \\ 1 \end{bmatrix}$$

In C++, with GLM:

```
#include <glm/gtx/transform.hpp> // after <glm/glm.hpp>

glm::mat4 myMatrix = glm::translate(glm::mat4(), glm::vec3(10.0f, 0.0f, 0.0f));
glm::vec4 myVector(10.0f, 10.0f, 10.0f, 0.0f);
glm::vec4 transformedVector = myMatrix * myVector; // guess the result
```

In GLSL :

```
vec4 transformedVector = myMatrix * myVector;
```

Rotation:

In GLM, the rotation is defined as follows:

```
glm::mat4 glm::rotate(
        glm::mat4 const & m,
        float angle,
        glm::vec3 const & axis);
```

which transforms a matrix with a rotation 4x4 matrix m created from an axis of 3 scalars and an angle expressed in degrees.

Let us see an example:

In C++:

```
// Use #include <glm/gtc/matrix_transform.hpp> and #include <glm/gtx/transform.hpp>
glm:vec3 myRotationAxis( 1.0f, 0.0f, 0.0f );
glm:mat4 rot = glm::rotate( angle_in_degrees, myRotationAxis);
```

Scaling:

In GLM, the scale is defined as follows:

```
glm::mat4 glm::scale(
        glm::mat4 const & m,
        glm::vec3 const & factors);
```

which transforms a matrix with a scale 4x4 matrix m created from a vector of 3 components.

Let us see an example:

In C++:

```
// Use #include <glm/gtc/matrix_transform.hpp> and #include <glm/gtx/transform.hpp>
glm:mat4 myScalingMatrix = glm::scale(2.0f, 2.0f, 2.0f );
```

For more details on GLM matrices, the reader is referred to:
http://www.opengl-tutorial.org/beginners-tutorials/tutorial-3-matrices/

Also, the GLM manual can be found at:
http://glm.g-truc.net/glm.pdf

6. Example

Let us see a program that draws a house. Then, such a house is shifted in steps of 0.1 units along the x-axis and y-axis up to reach 10 units in each axis.
The program is available at:
http://www.di.ubi.pt/~agomes/cg/praticas/movinghouse.zip

7. Programming Exercises

1. Re-write the house-building program implemented above in a way that all building blocks (body, roof, windows and door) are constructed from the origin. Then, use translations to place these building blocks at the desired locations. Each building block is constructed in a separate function. In the case of the window, we need to call its function twice because we are assuming the house has two windows.
2. Add the toggle facilities to the previous program in a manner to add/remove the house body by pressing the ‘b’ key, roof by pressing the ‘r’ key, windows by pressing the ‘w’ key, and door by pressing the ‘d’ key.
3. Let us now replicate twice the house. The first copy of the original house must be reduced to ¾ and placed side-by-side on the left of the original house. The second house copy must be scaled up to 5/4 and placed side-by-side on the right of the original house.
4. Let us now add the bright sun to the scene. The sun can be generated using the program concerning the Exercise 5 of the practical P01. The user can change the position of the sun by clicking on the ‘s’ key. The trajectory of the sun is a circle arc.
5. Based on the original code of movinghouse.zip, put the elements of the house (i.e., windows, door, and roof) to move away from the body center.
6. Based on the original code of movinghouse.zip, put the elements of the house (i.e., windows, door, and roof) to move away or translating them from the body center.
7. Based on the original code of movinghouse.zip, put the elements of the house (i.e., windows, door, and roof) to rotate about the body center.
8. Based on the original code of movinghouse.zip, put the elements of the house (i.e., windows, door, and roof) to rotate about <u>and</u> to move away from the body center simultaneously.

8. Final Remarks

1. Transformations are mathematical functions that allow us to <u>model</u> and to <u>navigate</u> within 2D and 3D spaces.

2. In computer graphics, we use three main transformations: <u>translation</u>, <u>scaling</u> and <u>rotation</u>.

3. Homogenous coordinates allow us to treat translation, scaling and rotation in the same manner. Consequently, all affine transformations can be combined into a CTM that substantially reduces the calculations that need to be made.

4. Matrix multiplication is associative but not commutative.
5. A CTM may combine many transformations into a single matrix.