Individual Assignment 2

Introduction to Computer Graphics

Casper B. Hansen
University of Copenhagen
Department of Computer Science
fvx507@alumni.ku.dk

last revision February 21, 2014

Abstract

We discuss the subject of geometric transformations in relation to computer graphics. In solving how we may perform such transformations, we look at how we can represent it in a convenient way and apply this to further develop how we can produce a uniform mathematical procedure by which we can arrive at the conclusion of how this translates into practice by concatenating such transformations. Finally we look at a possible implementation of geometric transformations in a 3D engine environment.

The reader is not expected to have any prior knowledge of the material presented. For the code the reader is expected to have some basic understanding of linear algebra and the C++ language.

Contents

1 The Problem

We seek a convenient mathematical expression that combines the properties of an object, and a uniform way of the transformation thereof.

1.1 Transformations

In our endeavor to produce these formulae, we must first define the transformations we require.

Translate displaces points by a vector.

Rotate displaces points by rotating them around an axis by an angle.

Scale displaces points by multiplying each component of the point by scale factors.

Shear displaces points by stretching them along an axis.

1.2 Euclidean Coordinates

In solving this, we consider an obvious choice of representation; for a point p in 2-dimensional space, we represent this in Euclidean coordinate space as $(p_x, p_y)^T$ — note $p \in \mathbb{R}^2$.

1.2.1 Translation

Given a point a, the translation tranformation of point a by point t, producing the point b is given by

$$\begin{bmatrix} b_x \\ b_y \end{bmatrix} = \begin{bmatrix} a_x \\ a_y \end{bmatrix} + \underbrace{\begin{bmatrix} t_x \\ t_y \end{bmatrix}}_{\mathbf{t}} = \begin{bmatrix} a_x + t_x \\ a_y + t_y \end{bmatrix} \tag{1}$$

Or, more compactly $b = a + \mathbf{t}(t_x, t_y)^T$.

Notice that this transformation is additive, not multiplicative, as the other transformation operations, as follows.

1.2.2 Rotation

Given a point a, the rotation transformation of point a around an axis by an angle θ, producing the point b is given by

$$\begin{aligned} \begin{bmatrix} b_x \\ b_y \end{bmatrix} &= \underbrace{\begin{bmatrix} \cos\theta & \sin\theta \\ -\sin\theta & \cos\theta \end{bmatrix}}_{\mathbf{R}(\theta)} \begin{bmatrix} a_x \\ b_y \end{bmatrix} \\ &= \begin{bmatrix} a_x \cos\theta - a_y \sin\theta \\ b_x \sin\theta - a_y \cos\theta \end{bmatrix} \end{aligned} \tag{2}$$

Or, more compactly $b = \mathbf{R}(\theta)a$.

1.2.3 Scaling

Given a point a, the scaling tranformation of point a by scale factors s, producing the point b is given by

$$\begin{bmatrix} b_x \\ b_y \end{bmatrix} = \underbrace{\begin{bmatrix} s_x & 0 \\ 0 & s_y \end{bmatrix}}_{\mathbf{S}} \begin{bmatrix} a_x \\ a_y \end{bmatrix} = \begin{bmatrix} s_x a_x \\ s_y a_y \end{bmatrix} \tag{3}$$

Or, more compactly $b = \mathbf{S}(s_x, s_y)a$.

1.2.4 Shearing

Given a point a, the shearing transformation of point a by the shear factors s, strecthing the point along an axis, producing the point b is given by

$$\begin{bmatrix} b_x \\ b_y \end{bmatrix} = \underbrace{\begin{bmatrix} 1 & s_x \\ 0 & 1 \end{bmatrix}}_{\mathbf{Sh}_x} \begin{bmatrix} a_x \\ a_y \end{bmatrix} = \begin{bmatrix} a_x + s_x a_y \\ a_y \end{bmatrix} \tag{4}$$

for shearing along the x-axis, and

$$\begin{bmatrix} b_x \\ b_y \end{bmatrix} = \underbrace{\begin{bmatrix} 1 & 0 \\ s_y & 1 \end{bmatrix}}_{\mathbf{Sh}_y} \begin{bmatrix} a_x \\ a_y \end{bmatrix} = \begin{bmatrix} a_x \\ a_y + s_y a_x \end{bmatrix} \tag{5}$$

for shearing along the y-axis. Or, more compactly $b = \mathbf{Sh}_\alpha(s_\alpha)a$, where α is an axis.

1.3 Homogeneous Coordinates

Consider a point $p_\alpha = (p_0, \ldots, p_{n-1})^T$ in n-nary Euclidean coordinate space. Then a corresponding homogeneous point $p_\beta = (wp_0, \ldots, wp_{n-1}, w)^T$ exists, where $w \neq 0$. Notice particularly that $p_\alpha \in \mathbb{R}^n$, whereas $p_\beta \in \mathbb{R}^{n+1}$. This is an inherent property of homogenous coordinates.

1.3.1 Translation

Given a Euclidean point a and displacement vector t, the translation transformation of point a displaced by vector t, producing the final point b is given by

$$\begin{bmatrix} wb_x \\ wb_y \\ wb_z \\ w \end{bmatrix} = \underbrace{\begin{bmatrix} 1 & 0 & 0 & t_x \\ 0 & 1 & 0 & t_y \\ 0 & 0 & 1 & t_z \\ 0 & 0 & 0 & 1 \end{bmatrix}}_{\mathbf{T}} \begin{bmatrix} a_x \\ a_y \\ a_z \\ 1 \end{bmatrix} = \begin{bmatrix} a_x + t_x \\ a_y + t_y \\ a_z + t_z \\ 1 \end{bmatrix} \tag{6}$$

or, equivalently $b = \mathbf{T}(t_x, t_y, t_z)^T a$.

1.3.2 Scaling

Given a Euclidean point a and scaling vector s, the scaling transformation of point a scaled by s, producing the transformed point b is given by

$$\begin{bmatrix} wb_x \\ wb_y \\ wb_z \\ w \end{bmatrix} = \underbrace{\begin{bmatrix} s_x & 0 & 0 & 0 \\ 0 & s_y & 0 & 0 \\ 0 & 0 & s_z & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}}_{\mathbf{S}(s_x, s_y, s_z)} \begin{bmatrix} a_x \\ a_y \\ a_z \\ 1 \end{bmatrix} = \begin{bmatrix} s_x a_x \\ s_y a_y \\ s_z a_z \\ 1 \end{bmatrix} \tag{7}$$

or, equivalently $b = \mathbf{S}(s_x, s_y, s_z)a$.

1.3.3 Rotation

In 3-dimensional space there are 3 distinct rotation matrices, one for each axis.

$$\mathbf{R}_x(\theta) = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & \cos\theta & -\sin\theta & 0 \\ 0 & \sin\theta & \cos\theta & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \tag{8}$$

$$\mathbf{R}_y(\theta) = \begin{bmatrix} \cos\theta & 0 & \sin\theta & 0 \\ 1 & 1 & 0 & 0 \\ -\sin\theta & 0 & \cos\theta & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \tag{9}$$

$$\mathbf{R}_z(\theta) = \begin{bmatrix} \cos\theta & -\sin\theta & 0 & 0 \\ \sin\theta & \cos\theta & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \tag{10}$$

Given a Euclidean point a and a rotation matrix $\mathbf{R}_\alpha(\theta)$, where $\alpha \in \{x, y, z\}$, the rotation transformation of point a around axis α by angle θ, producing the transformed point b is given by the formula to the right.

$$\begin{bmatrix} wb_x \\ wb_y \\ wb_z \\ w \end{bmatrix} = \mathbf{R}_\alpha(\theta) \begin{bmatrix} a_x \\ a_y \\ a_z \\ 1 \end{bmatrix} \tag{11}$$

1.3.4 Shearing

In 3-dimensional space there are 3 particular shearing transformations of interest, one for each axis along which we stretch the coordinates.

$$Sh_{yz}(s_y, s_z) = \begin{bmatrix} 1 & 0 & 0 & 0 \\ s_y & 1 & 0 & 0 \\ s_z & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \tag{12}$$

$$Sh_{xz}(s_x, s_z) = \begin{bmatrix} 1 & s_x & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & s_z & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \tag{13}$$

$$Sh_{xy}(s_x, s_y) = \begin{bmatrix} 1 & 0 & s_x & 0 \\ 0 & 1 & s_y & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \tag{14}$$

Given a Euclidean point a and a shearing matrix $\mathbf{Sh}_{\alpha\beta}(s_\alpha, s_\beta)$, where α and β are the opposite coordinate constituents of the axis γ we wish to shear, or stretch point a over, producing the transformed point b is given by the formula on the right

$$\begin{bmatrix} wb_x \\ wb_y \\ wb_z \\ w \end{bmatrix} = \mathbf{S}_{\alpha\beta}(s_\alpha, s_\beta) \begin{bmatrix} a_x \\ a_y \\ a_z \\ 1 \end{bmatrix} \tag{15}$$

2 The Solution

The conversion from Euclidean coordinate space into homogeneous coordinate space allows for a uniform transformation procedure, namely by matrix multiplication. Further, this uniform procedure allows for the concatenation of multiple transformations into one single transformation matrix. As such, we can conclude that homogeneous coordinates represent a significant advantage in accordance with the purposes.

2.1 Concatenating Transformations

Let p be a homogeneous point in n-ary space, and let $M_0, M_1, \ldots, M_{m-1}$ be a series of transformation matrices which we wish to apply to point p, in that particular order.

Recall from linear algebra that the order of multiplication produces different results. That is, $AB \neq BA$, and therefore we must take this into consideration whilst proceeding. Now, the purpose of a transformation matrix M is to transform a vector a producing another vector b, so we want to end up with a vector after the transformation, from linear algebra we know that our equation then must be of the form $b = Ma$.

Drawing upon this fact, we get the initial transformation $p_b = M_0 p_a$, which applies the first transformation matrix M_0 to point p_a producing the transformed point p_b. Further, applying the next transformation M_1 to p_b producing the next transformed point p_c we have, by the same fact, that $p_c = M_1 p_b$. By induction it then follows that $p_{final} = (M_m(M_{m-1}(\ldots(M_0)\ldots)p_{original}$. Notice particularly the order of the matrices. This concludes how to apply several transformations in succession. Note that this also allows us to concatenate rotations on all axis.

2.2 Optimizing for efficiency

Although the order in which the matrices appear in the equation matter, it is not so for the order in which they are calculated. That is, $A(BC) = (AB)C$.

The shader graphics pipeline works by applying three steps; a vertex stage, a fragment stage and geometry stage. Each applied in succession of the previous. The transformations are applied in the vertex stage, meaning that each matrix transformation must be done for each vertex. Instead of applying m matrix multiplications for each vertex we can optimize this stage by recognizing the mentioned theorem. We see that

$$(M_m(M_{m-1}(\ldots(M_0)\ldots)p = (M_m M_{m-1} \ldots M_0)p \tag{16}$$

Therefore, if we calculate the matrix transformations first, producing a so-called combined transformation matrix, or simply transformation matrix, we need not spend as much time in the vertex stage, as we would only have a single matrix multiplication for each vertex.

This is where the solution of using homogeneous coordinates shines, as opposed to using Euclidean coordinates, which wouldn't allow for such an optimization.

2.3 Order of transformations

In thinking of in which order we should apply the individual transformations discussed, we need to think of what we wish to achieve. Most notibly we would like for rotations to be applied before any translation displacement, otherwise we would rotate the vertices around the origin *after*, which would produce a different result. Scaling doesn't affect any following transformations, and so we choose to scale first. For shearing to occur in local object space this needs to happen immediately following scaling, and since we know that translation should happen after rotation, we have the final order of transformations; scale, shear, rotate and shear.

Our final transformation matrix then becomes

$$M = T(R_z R_y R_x)ShS \tag{17}$$

which we can apply to a point producing a transformed point by $p_{final} = Mp_{original}$.

3 Implementation

All implementation details related to this paper are found in `Object.cpp` on lines 28–77, an excerpt of which is given below. The `Object` class is designed to represent an object in 3-dimensional space.

```
glm::mat4 Object::getTranform()
{
    float s[16] =
    {
        _scale.x,   0.0f,       0.0f,       0.0f,
        0.0f,       _scale.y,   0.0f,       0.0f,
        0.0f,       0.0f,       _scale.z,   0.0f,
        0.0f,       0.0f,       0.0f,       1.0f
    };

    float r_x[16] =
    {
        1.0f,   0.0f,                   0.0f,                   0.0f,
        0.0f,   glm::cos(_rotation.x),  glm::sin(_rotation.x),  0.0f,
        0.0f,  -glm::sin(_rotation.x),  glm::cos(_rotation.x),  0.0f,
        0.0f,   0.0f,                   0.0f,                   1.0f
    };

    float r_y[16] =
    {
        glm::cos(_rotation.y),  0.0f,  -glm::sin(_rotation.y),  0.0f,
        0.0f,                   1.0f,   0.0f,                   0.0f,
        glm::sin(_rotation.y),  0.0f,   glm::cos(_rotation.y),  0.0f,
        0.0f,                   0.0f,   0.0f,                   1.0f
    };

    float r_z[16] =
    {
        glm::cos(_rotation.z),  glm::sin(_rotation.z),  0.0f,   0.0f,
       -glm::sin(_rotation.z),  glm::cos(_rotation.z),  0.0f,   0.0f,
        0.0f,                   0.0f,                   1.0f,   0.0f,
        0.0f,                   0.0f,                   0.0f,   1.0f
```

```
33      };
34
35      float t[16] =
36      {
37          1.0f,           0.0f,           0.0f,           0.0f,
38          0.0f,           1.0f,           0.0f,           0.0f,
39          0.0f,           0.0f,           1.0f,           0.0f,
40          _position.x,    _position.y,    _position.z,    1.0f
41      };
42
43      glm::mat4 s_mtx = glm::make_mat4(s);
44      glm::mat4 r_mtx = glm::make_mat4(r_z) * glm::make_mat4(r_y) * glm::
            make_mat4(r_x);
45      glm::mat4 t_mtx = glm::make_mat4(t);
46
47      matrix = t_mtx * r_mtx * s_mtx;
48
49      return matrix;
50  }
```

Note that the matrices appear transposed, this is a consequence of how the GLM type `glm::mat4` is declared — this is intensional. As it were, I accidentally didn't remember to implement shearing, and since it was too late to implement before the deadline it is left out of this excerpt as well, as this isn't reflected in the actual code, and would be a false reference. I assure the reader however, I do know how to do so, only time constraints held me back from doing so.

As the code shows, 5 matrices are declared and initialized using the values stored in the object, which describes the how the object is displaced in 3D space by way of Euclidean coordinates and vectors. One for scaling, one for each rotation, and lastly one for translation. Then, we use these to produce the GLM types of the equivalent matrices. For the premultiplications, we first produce a transformation matric containing all three rotations, for convenience and readability. Thereafter, we multiply the translation, rotation and scaling matrices together — note the order.

3.1 Tests

I didn't find a suitable testing method, but I have set up the program in such a way that the reader may test the solution without altering the code in any way. Controls have been set up, such that the arrow keys rotates the object around the x and y axis. Also, a half-implemented camera solution can be controlled using the w, a, s and d keys (this is not a fully functional camera yet, but a work in progress).

The only other test I can concieve of is screenshots. A series has been provided below.

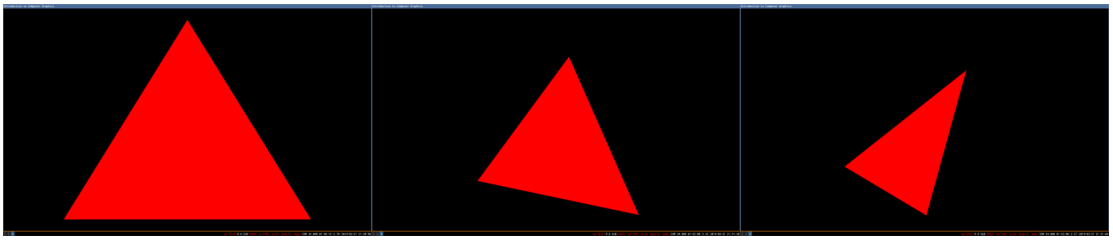

Figure 1: Screenshots of the implemention