Teoria i praktyka programowania gier komputerowych

Podstawy grafiki 3D

Piotr Fulmański

Wydział Matematyki i Informatyki,
Uniwersytet Łódzki, Polska

1 grudnia 2014

Spis treści

Any child-space position vector p_C can be transformed into a parent-space position vector p_P as follows

$$p_P = p_C M_{C \to P}$$

where

$$M_{C \to P} = \begin{bmatrix} i_C & 0 \\ j_C & 0 \\ k_C & 0 \\ t_C & 1 \end{bmatrix}$$

and

- i_C is the unit basis vector along the child space X-axis, expressed **in parent space coordinates**;
- j_C is the unit basis vector along the child space Y-axis, **in parent space**;
- k_C is the unit basis vector along the child space Z-axis, **in parent space**;
- t_C is the translation of the child coordinates system relative **to parent space**.

Affine transformations

Geometrically, affine transformations (affinities) preserve collinearity. So they transform parallel lines into parallel lines and preserve ratios of distances along parallel lines.

Any affine transformation matrix can be created by simply concatenating a sequence of 4×4 matrices representing pure translations, pure scale operations and pure rotations.
All affine 4×4 transformation matrices can be partitioned into four components

$$\begin{bmatrix} M_{3\times 3} & 0_{3\times 1} \\ t_{1\times 3} & 1_{1\times 1} \end{bmatrix}$$

where

- the upper 3×3 matrix M represents rotation and/or scale,
- the lower 1×3 vector t represents translation.

The following matrix T translates a point $p = [p_x \ p_y \ p_z]$ by the vector $t = [t_x \ t_y \ t_z]$ (p' is the translated point)

$$T = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ t_x & t_y & t_z & 1 \end{bmatrix}$$

In consequence we have

$$\begin{aligned} p + t &= [p_x \ p_y \ p_z \ 1] \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ t_x & t_y & t_z & 1 \end{bmatrix} \\ &= [(p_x + t_x) \ (p_y + t_y) \ (p_z + t_z)] \end{aligned}$$

The following matrix S scales the point $p = [p_x \;\; p_y \;\; p_z]$ by a factor s_x along the X-axis, s_y along the Y-axis, and s_z along the Z-axis

$$S = \begin{bmatrix} s_x & 0 & 0 & 0 \\ 0 & s_y & 0 & 0 \\ 0 & 0 & s_z & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

In consequence we have

$$\begin{aligned} p + t &= [p_x \;\; p_y \;\; p_z \;\; 1] \begin{bmatrix} s_x & 0 & 0 & 0 \\ 0 & s_y & 0 & 0 \\ 0 & 0 & s_z & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \\ &= [(s_x p_x) \; (s_y p_y) \; (s_z p_z) 1] \end{aligned}$$

[image of rotating point p by ϕ degrees to point p']

If we want to rotate point p by ϕ degrees to point p', we have simply that

$$\begin{aligned} p'_x &= |p'|\cos(\theta+\phi) \\ p'_y &= |p'|\sin(\theta+\phi) \end{aligned}$$

and

$$\begin{aligned} p_x &= |p|\cos(\theta) \\ p_y &= |p|\sin(\theta) \end{aligned}$$

Because we are dealing with rotations about the origin, thus we have

$$|p'| = |p|.$$

Using the trigonometric identities for the sum of angles we have that

$$\begin{aligned} p'_x &= |p|\cos(\phi)\cos(\theta) - |p|\sin(\phi)\sin(\theta) \\ p'_y &= |p|\cos(\phi)\sin(\theta) + |p|\sin(\phi)\cos(\theta) \end{aligned}$$

and finally

$$\begin{aligned} p'_x &= p_x\cos(\phi) - p_y\sin(\phi) \\ p'_y &= p_x\sin(\phi) + p_y\cos(\phi) \end{aligned}$$

Pushing this into matrix form

$$[p'_x \; p'_y] = [p_x \; p_y]\begin{bmatrix} \cos(\phi) & \sin(\phi) \\ -\sin(\phi) & \cos(\phi) \end{bmatrix}$$

Here, we have the rotation matrix for rotating a point in the $X - Y$ plane. Expanding this into 3D we have. . .

The following matrix represents rotation about the X-axis by an angle ϕ

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

The following matrix represents rotation about the Y-axis by an angle θ

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

The following matrix represents rotation about the Z-axis by an angle γ

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

Atomic transformation matrices

Rotations – remarks

- The order of rotations matters.
- $R(-\theta) = R^{1}(\theta) = R^{T}(\theta)$.

Problems with matrix representation of a rotation

- We need to much floating-point values (nine while we just have three degrees of freedom).
- As a consequence of previous: expensive calculation.
- It's hard to find intermediate rotations between two known rotations.

We can think about quaternions like an extension to complex numbers. A number of the form

$$a + 0i + 0j + 0k,$$

where a is a real number, is called **real**, and a number of the form

$$0 + bi + cj + dk,$$

where b, c, and d are real numbers, is called **pure imaginary**. If

$$a + bi + cj + dk$$

is any quaternion, then a is called its **scalar part** and $bi + cj + dk$ is called its **vector part**. The scalar part of a quaternion is always real, and the vector part is always pure imaginary. Even though every quaternion is a vector in a four-dimensional vector space, it is common to define a vector to mean a pure imaginary quaternion. With this convention, a vector is the same as an element of the vector space R^3.
Hamilton called pure imaginary quaternions **right quaternions** and real numbers (considered as quaternions with zero vector part) **scalar quaternions**.

```
struct QUATERNION
{
  float x, y, z, w;

  QUATERNION() { }
  QUATERNION(float x, float y, float z, float w):
    x(x), y(y), z(z), w(w) { }
};
```

Jednostkowy kwaternion można utożsamiać z obrotem w przestrzeni 3D. Kwaternion tworzy się podając jednostkowy wektor, którego kierunek wskazuje oś obrotu oraz kąt, o jaki chcemy obracać wokół tego wektora (zwykle w radianach).
Informacji tych nie wpisujemy jednak do składowych kwaterniona bezpośrednio. Trzeba je zakodować według algorytmu jak na poniższym listingu, obliczając najpierw sinus i cosinus połowy podanego kąta.

```
void AxisToQuaternion(QUATERNION *Out,
                      const VEC3 &Axis,
                      float Angle)
{
    Angle *= 0.5f;
    float Sin = sinf(Angle);
    Out->x = Sin * Axis.x;
    Out->y = Sin * Axis.y;
    Out->z = Sin * Axis.z;
    Out->w = cosf(Angle);
}
```

Przypadkiem szczególnym jest obracanie wokół osi X, Y lub Z. Algorytm znacznie się wówczas upraszcza i dla optymalizacji warto przygotować osobne funkcje. Poniżej funkcja dla obrotu wokół osi X; dla pozostałych przypadków należy postąpić analogicznie.

```
void QuaternionRotationX(QUATERNION *Out, float a)
{
    a *= 0.5f;
    Out->x = sinf(a);
    Out->y = 0.0f;
    Out->z = 0.0f;
    Out->w = cosf(a);
}
```

As a set, the quaternions $\mathbb{H}$ are equal to R^4, a four-dimensional vector space over the real numbers. The quaternions looks a lot like a four-dimensional vector, but it behaves quite differently.
$\mathbb{H}$ has three operations: addition, scalar multiplication, and quaternion multiplication.

Quaternions support some of the familiar operations from vector algebra, such as vector addition. We have see a formula for addition – to remember it, if

$$q = (r, v),\ \ q \in \mathbb{H},\ \ r \in R,\ \ v \in R^3$$

then

$$(r_1,\ \ v_1) + (r_2,\ \ v_2) = (r_1 + r_2,\ \ v_1 + v_2).$$

However, we must remember that **the sum of two unit quaternions does not represent a 3D rotation, because such a quaternion would not be of unit length**.

One of the most important operations we will perform on quaternions is that of multiplication. Given two quaternions p and q representing two rotations P and Q, respectively, the product pq represents the composite rotation (i.e., rotation Q followed by rotation P[1]). We will restrict to the multiplication which is used in conjunction with 3D rotations, namely the Grassman product. If

$$q = (r, v), \;\; q \in \mathbb{H}, \;\; r \in R, \;\; v \in R^3$$

then

$$(r_1, v_1)(r_2, v_2) = (r_1 r_2 - v_1 \cdot v_2, r_1 v_2 + r_2 v_1 + v_1 \times v_2).$$

[1] Mind the order!

If

$$q = (r, v), \ q \in \mathbb{H}, \ r \in R, \ v \in R^3$$

then norm $|q|$ is defined as follows

$$|q| = \sqrt{q\overline{q}} = \sqrt{\overline{q}q} = \sqrt{r^2 + v_x^2 + v_y^2 + v_z^2},$$

where $\overline{q}$ denotes conjugation (to be explain). To normalize vector the following formula have to be used

$$\text{normalize}(q) = \frac{q}{|q|} = \left[\frac{v_x}{|q|} \ \frac{v_y}{|q|} \ \frac{v_z}{|q|} \ \frac{r}{|q|} \right].$$

Conjugate of a quaternion q is defined as follows

$$\overline{q} = (r, -v)$$

where

$$q = (r, v), \ q \in \mathbb{H}, \ r \in R, \ v \in R^3.$$

The inverse of a quaternion q is denoted q^{-1} and is defined as a quaternion which, when multiplied by the original, yields the scalar 1 (i.e., $qq^{-1} = 0i + 0j + 0k + 1$)

$$q^{-1} = \frac{\overline{q}}{|q|^2}$$

where

$$q = (r, v), \;\; q \in \mathbb{H}, \;\; r \in R, \;\; v \in R^3.$$

What is nice, because in computer games quaternions represent 3D rotations, they are always of unit length. So, for our purposes, the inverse and the conjugate are identical:

$$q^{-1} = \overline{q}$$

where

$$q = (r, v),\ \ q \in \mathbb{H},\ \ r \in R,\ \ v \in R^3.$$

Other properties

$$\overline{(pq)} = \overline{q}\,\overline{p},$$

$$(pq)^{-1} = q^{-1}p^{-1}.$$

Rewrite vector v in quaternion form v_q

$$v_q = (0, v) = [v_x \;\; v_y \;\; v_z \;\; 0].$$

The rotated vector v' by a quaternion q can be found as follows

$$v' = \text{rotate}(v, q) = q v_q q^{-1}.$$

Consider three distinct rotations, represented by the quaternions q_1, q_2 and q_3. We want to apply rotation 1 first, followed by rotation 2 and finally rotation3. The composite rotation quaternion q_{comp} can be found and applied to vector v (in its quaternion form, v_q) to get rotated vector v' as follows

$$v' = q_3 q_2 q_1 v_q q_1^{-1} q_2^{-1} q_3^{-1} = q_{comp} v_q q_{comp}^{-1}.$$

If we let

$$q = (r, v) = [v_x \ \ v_y \ \ v_z \ \ r] = [x \ \ y \ \ z \ \ w]$$

then matrix representation of 3D rotation M we can find as follow

$$M = \begin{bmatrix} 1 - 2y^2 - 2z^2 & 2xy + 2zw & 2xz - 2yw \\ 2xy - 2zw & 1 - 2x^2 - 2z^2 & 2yz + 2xw \\ 2xz + 2yw & 2yz - 2xw & 1 - 2x^2 - 2y^2 \end{bmatrix}$$

Given two quaternions q_A and q_B representing rotations A and B, we can find an intermediate rotation q_{LERP} that is t percent of the way from A to B as follows

$$
\begin{aligned}
q_{\text{LERP}} &= \text{LERP}(q_A, q_B, t) = \frac{(1-t)\mathbf{q_A} + t\mathbf{q_B}}{|(1-t)\mathbf{q_A} + t\mathbf{q_B}|} \\
&= \text{normalize}\left(\begin{bmatrix} (1-t)v_x^A + tv_x^B \\ (1-t)v_y^A + tv_y^B \\ (1-t)v_z^A + tv_z^B \\ (1-t)r^A + tr^B \end{bmatrix}^T \right).
\end{aligned}
$$

The problem with the LERP is that it effectively interpolates along a chord of the hypersphere, rather than along the surface of the hypersphere itself. This leads to rotational animations that do not have a constant angular speed when the parameter t is changing at a constant rate. The rotation will appear slower at the end points and faster in the middle of the animation.

To solve this problem, we can use a variant of the LERP operation known as spherical linear interpolation, or SLERP for short

$$\mathrm{SLERP}(p, q, t) = t_p p + t_q q,$$

where

$$\begin{aligned} t_p &= \frac{\sin((1-t)\theta)}{\sin(\theta)}, \\ t_q &= \frac{\sin(t\theta)}{\sin(\theta)}, \end{aligned}$$

and

$$\theta = \arccos(p \cdot q).$$