VQS Transformation with an Incremental Approach

Xin Li
Computer Science Department
Digipen Institute of Technology
xli@digipen.edu

Abstract

This paper presents an incremental method to 3D transformation by *VQS* structures. For interpolated scaling, rotation and translation between two animation key frames, with constant intervals in size, angle and distance, the proposed approach reduces the *VQS* transformation cost by more than an half. Moreover, all interpolations for *VQS* key frames are performed implicitly at no cost. In other words, the expensive trigonometric functions in quaternion Slerp and exponential functions in scalar interpolation are completely eliminated from the process. The method is fast, accurate, numerically stable and also preserves all geometric characteristics of the original transformation and interpolation algorithms.

1. Introduction

In 3D graphics, quaternions are often considered as the "weapon of choice" for rotations [1]. Similar to the homogenous matrix approach, the rotation by a quaternion is combined with a translation vector and a uniform scaling factor. The operation is referred as transformation by a *VQS structure* [3].

Mathematically, a *VQS* structure is defined as triplet $T = [v, q, s]$, where $v=[x_v, y_v, z_v, 0]$ is a "pure quaternion" representing a translation vector, $q=<u, w>$ is a quaternion with $u=<x_u, y_u, z_u>$ and s is a uniform scaling factor. A transformation of any given "pure quaternion" $r=<x, y, z, 0>$ is then calculated by operations defined on quaternions, namely multiplication, scale product and addition. It results in another pure quaternion r':

$$r' =< x', y', z', 0 > = s(qrq^{-1}) + v$$

That is, a VQS transformation is to rotate by q, scale by s and then translate by v. It is easy to verify that the result is equivalent to the matrix transformation. In practice, to save the computational time, quaternions are multiplied out to yield

$$r' = s((w^2 - u^2)r + 2(u \cdot r)u + 2w(u \times r)) + v \qquad \text{E.1-1}$$

where v, r and r' are treated as 3D vectors. The transformation by a *VQS* in this approach takes 28 multiplications and 17 additions/subtractions.

2. VQS Interpolation

In real-time simulation and video game software, *VQS* structures are often created as key frames to reduce the space occupied by animation data. At run_time, the intermediate transformations are generated by interpolating between key frames. If we assume $T_0=[v_0, q_0, s_0]$ and $T_n=[v_n, q_n, s_n]$ are two *VQS* key frames, then $T_t=[v_t, q_t, s_t]$, for any $t \in [0.0:1.0]$, can be computed by the following three interpolation algorithms:

(1) Lerp for translation:
$$v_t = (1-t)v_0 + tv_n$$

(2) Slerp for rotation:
$$q_t = \frac{1}{\sin(\alpha)}\left[\sin(\alpha - t\alpha)q_0 + \sin(t\alpha)q_n\right] \qquad \text{E.2-1}$$

(3) Exponential interpolation for scaling:
$$s_t = (\frac{s_n}{s_0})^t s_0$$

$T_t = [v_t, q_t, s_t]$ is then used for transformation to create a smooth motion. Note that the exponential interpolation (instead of linear) is used for scaling factors. This is intended to compensate the non-linearity of delusional perception of the human eye since it is more sensitive to changes in smaller objects than that in larger ones. Figure 2-1 compares the linear and exponential approaches between scales of 0.1 and 10 in 100 interpolation steps.

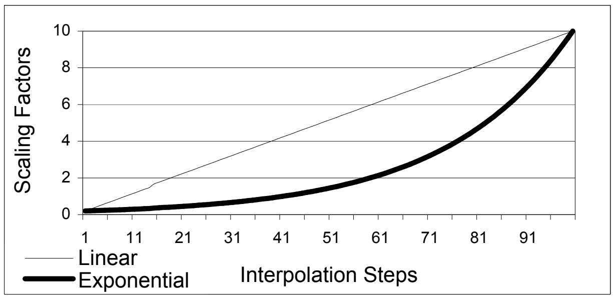

Figure 2-1: Comparison of Linear and exponential scaling

3. Incremental VQS interpolation

Now, we consider the incremental approach to *VQS* interpolation. That is, we further assume that, when interpolating between T_0 and T_n, it can be done incrementally through n steps with a fixed interval for locations, angles and scales. This allows us to change t from 0.0 to 1.0 with a constant interval $\Delta t = 1/n$ and obtain integer k such that

$$t = k\Delta t = k/n, \quad \text{for } k=0, 1, \dots, n.$$

Under these assumptions, after replacing t with k/n, equations in E.2-1 become

$$\begin{aligned}
v_k &= kv_c + v_0, && \text{where } v_c = (v_n - v_0)/n \\
q_k &= q_c{}^k q_0, && \text{where } q_c = [\cos(\beta),\ \sin(\beta)u] \qquad \text{E.3-1} \\
s_k &= s_c^k s_0, && \text{where } s_c = (\frac{s_n}{s_0})^{1/n}
\end{aligned}$$

We refer those equations in E.3-1 as "incremental interpolations" for translation, rotation and scaling respectively. The conversions of the first and last equations are straightforward. However, the second equation for quaternion interpolation requires some explanations. Basically, it converts the original Slerp interpolation into its incremental form. In other words, E.3-1 says that the slerped quaternion q_k can be computed by multiplication of k-th power of a constant quaternion q_c and q_0, where $\beta = \alpha/n$ and u is a unit vector defined by components of q_0 and q_n as

$$u = (w_0 u_n - w_n u_0 + u_0 \times u_n)/\sin(\alpha).$$

The derivation is discussed in full length in [4].

Let $T_0 = [v_0, q_0, s_0]$ be the initial *VQS*. Equations in E.3-1 allow to rewrite the incrementally-interpolated *VQS* at step $k+1$ as $T_{k+1} = [v_{k+1}, q_{k+1}, s_{k+1}]$ with recursive functions of $T_k = [v_k, q_k, s_k]$ from the previous interpolation step. That is, if we assume $[v_0, q_0, s_0]$ at $k=0$, then we always have the following at any $k=1, \dots, n-1$:

$$\begin{aligned}
v_{k+1} &= (k+1)v_c + v_0 = v_c + v_k, \quad v_c = (v_n - v_0)/n \\
q_{k+1} &= q_c{}^{k+1} q_0 = q_c q_k, \quad q_c = [\cos(\beta),\ \sin(\beta)u] \qquad \text{E.3-2} \\
s_{k+1} &= s_c^{k+1} s_0 = s_c s_k, \qquad s_c = (\frac{s_n}{s_0})^{1/n}
\end{aligned}$$

Note that, in E.3-2, all v_c, q_c and s_c are constants and therefore can be pre-computed once for given T_0, T_n and interpolation steps n.

With the recursive definitions, transformations by a sequence of interpolated *VQS* structures, from T_0 to T_n in n steps of fixed intervals on any arbitrary vector r, can be written as

$$r_0 = [v_0, q_0, s_0]r$$

$$r_{k+1} = [v_{k+1}, q_{k+1}, s_{k+1}]r = s_{k+1}(q_{k+1} r q_{k+1}{}^{-1}) + v_{k+1}$$

Replacing with recursive definitions for translation, quaternion and scaling factor at $k+1$, we obtain

$$\begin{aligned}
r_{k+1} &= s_{k+1}(q_{k+1} r q_{k+1}{}^{-1}) + v_{k+1} \\
&= (s_c s_k)((q_c q_k) r (q_c q_k)^{-1}) + (k+1)v_c + v_0 \\
&= s_c(q_c(s_k(q_k r q_k{}^{-1}))q_c{}^{-1}) + (k+1)v_c + v_0 \\
&= s_c(q_c(s_k(q_k r q_k{}^{-1}) + v_k - v_k)q_c{}^{-1}) + (k+1)v_c + v_0 \\
&= s_c(q_c(s_k(q_k r q_k{}^{-1}) + v_k)q_c{}^{-1}) - s_c(q_c v_k q_c{}^{-1}) \\
&\quad + (k+1)v_c + v_0 \\
&= s_c q_c r_k q_c{}^{-1} - s_c q_c(kv_c + v_0)q_c{}^{-1} + (k+1)v_c + v_0 \\
&= s_c q_c r_k q_c{}^{-1} - ks_c q_c v_c q_c{}^{-1} - s_c q_c v_0 q_c{}^{-1} + (k+1)v_c + v_0 \\
&= s_c q_c r_k q_c{}^{-1} + v_c - s_c q_c v_0 q_c{}^{-1} + v_0 + k(v_c - s_c q_c v_c q_c{}^{-1})
\end{aligned}$$

Let

$$V_0 = v_c - s_c q_c v_0 q_c{}^{-1} + v_0$$

$$V_c = v_c - s_c q_c v_c q_c{}^{-1}$$

We obtain

$$r_{k+1} = s_c q_c r_k q_c{}^{-1} + V_0 + kV_c \qquad \text{E.3-3}$$

It is important to point out that, since v_c, q_c and s_c are constants, so are V_0 and V_c. Furthermore, we recursively define

$$V_0 = v_c - s_c q_c v_0 q_c{}^{-1} + v_0$$

$$V_k = V_0 + kV_c = V_0 + (k-1)V_c + V_c = V_{k-1} + V_c$$

Substituting in E.3-3, we have

$$\begin{aligned}
V_k &= V_{k-1} + V_c \\
r_{k+1} &= s_c q_c r_k q_c{}^{-1} + V_k
\end{aligned} \qquad \text{E.3-4}$$

These equations are incremental formulas. For a pair of key *VQS* structures $T_0 = [v_0, q_0, s_0]$ and $T_n = [v_n, q_n, s_n]$, any vector r can be initially transformed by

$$r_0 = [v_0, q_0, s_0]r = s_0(q_0 r q_0{}^{-1}) + v_0$$

$$V_0 = v_0 - s_c q_c v_0 q_c{}^{-1} + v_c$$

Then it can be calculated based on the transformed vector incrementally in the previous step:

$$r_1 = [v_1, q_1, s_1]r = s_c(q_c r_0 q_c{}^{-1}) + V_0, \qquad V_1 = V_0 + V_c$$

$$r_2 = [v_2, q_2, s_2]r = s_c(q_c r_1 q_c{}^{-1}) + V_1, \qquad V_2 = V_1 + V_c$$

$$\dots$$

$$r_{k+1} = [v_{k+1}, q_{k+1}, s_{k+1}]r = s_c(q_c r_k q_c{}^{-1}) + V_k, \quad V_{k+1} = V_k + V_c$$

$$\dots$$

Assume $q_c = [w_c, u_c]$ and u_c=<x_c, y_c, z_c>. If quaternions are multiplied out, the second equation in E.3-4 then becomes

$$r_{k+1} = s_c((w_c^2 - u_c^2)\, r_k + 2(u_c \cdot r_k)\, u_c + 2w_c(u_c \times r_k)) + V_k$$

Define the following matrices based on u_c:

$$\hat{U}_c = \begin{vmatrix} x_c^2 & x_c y_c & x_c z_c \\ x_c y_c & y_c^2 & y_c z_c \\ x_c z_c & y_c z_c & z_c^2 \end{vmatrix} \text{ and } \widetilde{U}_c = \begin{vmatrix} 0 & -z_c & y_c \\ z_c & 0 & -x_c \\ -y_c & x_c & 0 \end{vmatrix}$$

Hence equation E.3-4 can be rewritten as

$$\begin{aligned} r_{k+1} &= s_c((w_c^2 - u_c^2) I r_k + 2\hat{U}_c r_k + 2w_c \widetilde{U}_c r_k) + V_k \\ &= M_c r_k + V_k \end{aligned} \qquad \text{E.3-5}$$

where I is an identity matrix and

$$M_c = s_c((w_c^2 - u_c^2) I + 2\hat{U}_c + 2w_c \widetilde{U}_c)$$

is a 3x3 matrix. '^' and '~' are 'head' and 'tilde' operators respectively that convert dot and cross products between vectors into multiplications of a matrix and a vector.

Note that M_c is constant matrix since its elements are all constants. It can be pre-computed once for any given pair of key frames and stored with the *VQS* structure. When it comes to run-time transformation of any vector *r*, the next step (*k+1*) becomes a multiplication with a 3x3 matrix and then a vector addition, which takes only 9 multiplications and 9 additions from the previous step (*k*). Compared with E.1-1, it cuts the computational cost to less than half.

But wait, there is more: All interpolations are included for free. There is absolutely no additional cost for Lerp, Slerp or exponential interpolations.

4. Implementation

We now illustrate how *VQS* incremental transformation is implemented as a C++ class. To simplify the discussion, we assume that a set of utility functions (similar to those in DirectX libraries) that support the basic vector, matrix and quaternion classes and operations is available. The incremental *VQS* class (*iVQS*) definition includes four private member variables to store the constant incremental matrix (*Mc*), translation vectors (*Vk* and *Vc*) and the total number of interpolation steps (*Count*). The class also provides three public member functions that initialize the structure, step through interpolations and actually transform a given vector.

The pseudocode of the class is illurstrated in List 4-1. The member function implementation is straightforward based on equations in section 3 and therefore skipped.

```
class iVQS
{
  private:
   Matrix Mc;       // Incremental matrix;
   Vector Vk, Vc;   // Incremental translation;
   int    Count;    // Number of interpolation steps;

  public:
   // This function initializes the iVQS structure.
   // [v0, q0, s0] and [vn, qn, sn] are key frame VQS.
   // n represents desired incremental steps.
   // The function computes and stores incremental
   // constants Mc and Vc.
   void iVQSInit(Vector& v0,  Quaternion& q0,  float s0,
            Vector& vn, Quaternion& qn,  float sn,   int n);

   // This function steps to the next interpolated VQS and
   // increment the count. It is called once and only once
   // for each transformation frame. The function returns
   // FALSE at the end of n iterations or TRUE otherwise.
   Boolean iVQSStep(void);

   // This function returns a transformed vector based on
   // rk from previous step.
   Vector iVQSTransform(Vector& rk);
};
```

List 4-1: *iVQS* class definition

5. Error analysis

As mentioned earlier, this approach reduces the computational expense of *VQS* transformation of any vector to less than half. Yet it comes with Lerp, Slerp and Exponential interpolations for translation, quaternion and scaling factor at no cost at all. Since the method only uses additions and multiplications, the calculation is immune to numerical instability.

However, since the method is based on incremental steps over a constant interval, the implementation on any hardware and software platforms would introduce some cumulative floating-point round-off errors. In other words, transformations by this approach would inevitably deviate from the path of the original *VQS* interpolations.

To quantify accumulated errors and examine the "drifting" behavior, we designed a series of tests over 200 pairs of *VQS* structures that contain randomized parameters within controlled ranges ([-999.0 : 999.0] for translations, [1° : 89°] for rotations and [0.1 : 10.0] for uniform scaling). 200 vertices are first transformed by the original *VQS* with Lerp, Slerp and Exponential interpolations through 100 fixed steps between *VQS* pairs. Then the incremental approach is applied under the same test conditions.

List 5-1 contains the pseudo code for the error analysis.

```
1. Generate 200 random vectors;
2. Create 200 pairs of random VQS in a specified range;
3. For each pair of VQS, repeat:
   3.1. Interpolate by regular and incremental
        algorithms in 100 steps.
   3.2. At each step, do the following:
        3.2.1. Transform each of 200 vectors by
               both interpolated VQS structures.
        3.2.2. For each pair of resulting vectors,
               calculate the errors in between.
        3.2.3. Record the maximum and average
               error at each step.
4. Print the record errors.
```

List 5-1: Error analysis pseudo code

Let r = <*x, y, z*> be a vector transformed by a *VQS* structure. Let r' = <*x', y', z'*> be the vector transformed by the incremental approach from the same vector with the same interpolation steps. For each pair of r and r', four types of relative errors are computed as follows

$$\sigma_x = \frac{|x - x'|}{|r|}, \sigma_y = \frac{|y - y'|}{|r|}, \sigma_y = \frac{|z - z'|}{|r|}, \ \sigma_d = \frac{|r - r'|}{|r|}$$

where σ_x, σ_y and σ_z are dircrepancies of x, y and z components between vectors processed by two different approaches. σ_d is the distance between them. To avoid the "magnitude effect", all errors are normalized by the length of the vector. The maximum component errors among all 200 transformed vectors are recorded at each interpolation step. For the distance, we log both maximum and average errors at each step. The numbers are depicted in Figure 5-1 and Figure 5-2.

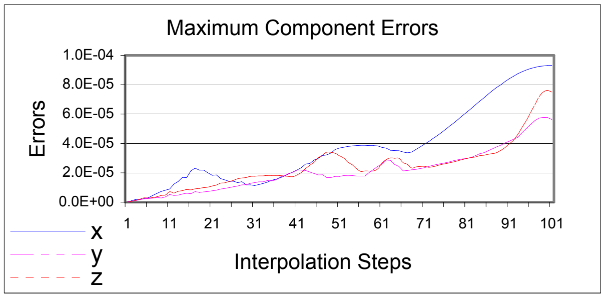

Figure 5-1: Cumulative round-off errors (components)

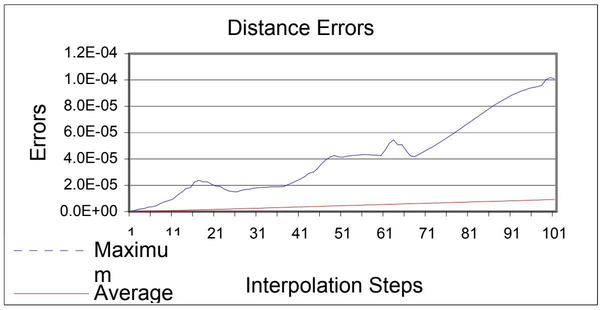

Figure 5-2: Cumulative round-off errors (distances)

As expected, locations and components of incrementally interpolated *VQS* transformations tend to drift away from original *VQS* transformations. The maximum cumulative round-off errors increase with the number of interpolation steps between the starting and ending *VQS* structures.

However, the tests also indicate that this drifting behavior is insignificantly trivial. The average relative errors in distance is less than 0.001% with 100 steps of incremental interpolation and transformation. Even in the worst cases, all maximum round-off errors for components and distance are below 0.01%.

Note that the single-precision 32-bits floating point was used in the above tests. The round-off errors should be further reduced under double-precision 64-bits operations.

6. Conclusion

As pointed out earlier, the proposed incremental *VQS* method in this paper is fast and accurate enough for most animations in 3D real-time simulation and video game software. However, since the transformation by the incremental *VQS* can not be easily concatenated in a hierarchical structure, it is more suitable for objects with a "flat structure". That is, the animation key frames would have to be specified with respect to the world coordinate system. Furthermore, although the method cuts down the computational cost to less than an half, it will likely double the storage for object geometric data since vertices from the previous step must be saved for transformations in the next frame.

References:

[1] Ken Shoemake, "Animating Rotation with Quaternion Curves", Computer Graphics, Volume 19, Number 3, 1985.

[2] David Eberly, "Quaternion Algebra and Calculus", http://www.geometrictools.com/Documentation/Quaternions.pdf , September 27, 2002.

[3] Warren Robinett and Richard Holloway, "The Visual Display Transformation for Virtual reality", Technical Report (TR94-031), UNC-Chapel Hill, Sept. 1994.

[4] Xin Li, "To Slerp, Or Not to Slerp", Game Developer Magazine, August, 2006.

[5] Jonathan Blow, "Hacking Quaternions", Game Developer Magazine, March 2002.

[6] Jonathan Blow, "Understanding Slerp, Then Not Using It", Game Developer Magazine, April 2004.

[7] Thomas Busser, "PolySlerp, A Fast and Accurate Polynomial Approximation of Slerp", Game Developer Magazine, February, 2004.