III. Topics
22. Quaternions

Game Graphic Programming

Kyung Hee University
Software Convergence
Prof. Daeho Lee

Definitions

- An ordered pair of real numbers $\mathbf{z} = (a, b)$ is a complex number. The first component is called the real part, and the second component is called the imaginary part.
  - The absolute value, or magnitude, of the complex number $a + ib$ is defined as the length of the vector it represents which we know is given by:

$$|a+ib| = \sqrt{a^2 + b^2}$$

  - Polar representation and rotations

$$r = |a+ib|$$
$$a+ib = r\cos\theta + ir\sin\theta = r(\cos\theta + i\sin\theta)$$

Definition and Basic Operations

- An ordered 4-tuple of real numbers $\mathbf{q} = (x, y, z, w) = (q_1, q_2, q_1, q_4)$ is a quaternion.
- This is commonly abbreviated as $\mathbf{q} = (\mathbf{u}, w) = (x, y, z, w)$ and we call $\mathbf{u} = (x, y, z)$ the imaginary vector part and w the real part.
  - $(\mathbf{u}, a) = (\mathbf{v}, b)$ if and only if $\mathbf{u} = \mathbf{v}$ and $a = b$
  - $(\mathbf{u}, a) \pm (\mathbf{v}, b) = (\mathbf{u} \pm \mathbf{v}, a \pm b)$
  - $(\mathbf{u}, a)\, (\mathbf{v}, b) = (a\mathbf{v} + b\mathbf{u} + \mathbf{u} \times \mathbf{v}, ab - \mathbf{u} \cdot \mathbf{v})$

$$\mathbf{pq} = \begin{pmatrix} p_4 & -p_3 & p_2 & p_1 \\ p_3 & p_4 & -p_1 & p_2 \\ -p_2 & p_1 & p_4 & p_3 \\ -p_1 & -p_2 & -p_3 & p_4 \end{pmatrix} \begin{pmatrix} q_1 \\ q_2 \\ q_3 \\ q_4 \end{pmatrix}$$

Kyung Hee University
nize@khu.ac.kr

Special Products & Properties

- Special products
  - $\mathbf{i} = (1, 0, 0, 0)$, $\mathbf{j} = (0, 1, 0, 0)$, $\mathbf{k} = (0, 0, 1, 0)$
    - $\mathbf{i}^2=\mathbf{j}^2=\mathbf{k}^2=\mathbf{ijk}=-1$
    - $\mathbf{ij}=\mathbf{k}=-\mathbf{ji}$
    - $\mathbf{jk}=\mathbf{i}=-\mathbf{kj}$
    - $\mathbf{ki}=\mathbf{j}=-\mathbf{ik}$
- Properties
  - Quaternion multiplication is not commutative.
  - Quaternion multiplication is associative; however, this can be seen from the fact that quaternion multiplication can be written using matrix multiplication, and matrix multiplication is associative.
  - The quaternion $\mathbf{e} = (0, 0, 0, 1)$ serves as a multiplicative identity.
  - $\mathbf{p}(\mathbf{q} + \mathbf{r}) = \mathbf{pq} + \mathbf{pr}$ and $(\mathbf{q} + \mathbf{r})\mathbf{p} = \mathbf{qp} + \mathbf{rp}$.

$$\mathbf{pe} = \mathbf{ep} = \begin{pmatrix} p_4 & -p_3 & p_2 & p_1 \\ p_3 & p_4 & -p_1 & p_2 \\ -p_2 & p_1 & p_4 & p_3 \\ -p_1 & -p_2 & -p_3 & p_4 \end{pmatrix} \begin{pmatrix} 0 \\ 0 \\ 0 \\ 1 \end{pmatrix} = \begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix} \begin{pmatrix} p_1 \\ p_2 \\ p_3 \\ p_4 \end{pmatrix} = \begin{pmatrix} p_1 \\ p_2 \\ p_3 \\ p_4 \end{pmatrix}$$

Conversions

- s is a real number and $\mathbf{u} = (x, y, z)$ is a vector.
  - $s = (0, 0, 0, s)$
  - $\mathbf{u} = (x, y, z) = (\mathbf{u}, 0) = (x, y, z, 0)$
- Identity quaternion, $1 = (0, 0, 0, 1)$.
- A quaternion with zero real part is called a pure quaternion.
- A real number times a quaternion is just "scalar multiplication" and it is commutative.
  - $s\mathbf{q} = \mathbf{q}s$

Conjugate and Norm

- The conjugate of a quaternion $\mathbf{q} = (q_1, q_2, q_3, q_4) = (\mathbf{u}, q_4)$ is denoted by $\mathbf{q}^*$ and defined by
  - $\mathbf{q}^* = - q_1 - q_2 - q_3 + q_4 = (-\mathbf{u}, q_4)$
  - $(\mathbf{pq})^* = \mathbf{q}^* \mathbf{p}^*$
  - $(\mathbf{p} + \mathbf{q})^* = \mathbf{q}^* + \mathbf{p}^*$
  - $(\mathbf{q}^*)^* = \mathbf{q}$
  - $(s\mathbf{q})^* = s\mathbf{q}^*$ for $s \in \mathbb{R}$
  - $\mathbf{q} + \mathbf{q}^* = (\mathbf{u}, q_4) + (-\mathbf{u}, q_4) = 2q_4$
  - $\mathbf{qq}^* = \mathbf{q}^*\mathbf{q} = q_1^2 + q_2^2 + q_3^2 + q_4^2 = \|\mathbf{u}\|^2 + q_4^2$
  - Norm (magnitude) $\|\mathbf{q}\| = \sqrt{\mathbf{qq}^*} = \sqrt{\|\mathbf{u}\|^2 + q_4^2}$
    - $\|\mathbf{q}^*\| = \|\mathbf{q}\|$
    - $\|\mathbf{pq}\| = \|\mathbf{p}\| \, \|\mathbf{q}\|$
  - A quaternion is a **unit quaternion** if it has a norm of one.
    - $\|\mathbf{pq}\|^2 = \|\mathbf{p}\|^2 \, \|\mathbf{q}\|^2$

Inverses

- Quaternion multiplication is not commutative, so we cannot define a division operator.
- Every nonzero quaternion has an inverse.

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

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

$$\left(\mathbf{q}^{-1}\right)^{-1} = \mathbf{q}$$

$$\left(\mathbf{p}\mathbf{q}\right)^{-1} = \mathbf{q}^{-1}\mathbf{p}^{-1}$$

Polar Representation

- Polar representation
  - For unit quaternion

$$\|\mathbf{q}\|^2 = \|\mathbf{u}\|^2 + q_4^2 = 1$$

- Polar representation

$$\mathbf{q} = \left(\sin\theta\mathbf{n}, \cos\theta\right) \qquad \text{for } \theta \in \left[0, \pi\right]$$

n is a unit vector $\quad \mathbf{n} = \dfrac{\mathbf{u}}{\|\mathbf{u}\|} = \dfrac{\mathbf{u}}{\sin\theta}$

$$\mathbf{q}^* = \left(-\sin\theta\mathbf{n}, \cos\theta\right) = \left(\sin(-\theta)\mathbf{n}, \cos(-\theta)\right)$$

Rotation Operator (1)

- Rotation operator
  - **q**: unit quaternion $\mathbf{q} = (\mathbf{u}, w)$
  - **v**: 3D point or vector $\mathbf{p} = (\mathbf{v}, 0)$

$$
\begin{aligned}
\mathbf{qpq}^{-1} &= \mathbf{qpq}^{*} \\
&= (\mathbf{u}, w)(\mathbf{v}, 0)(-\mathbf{u}, w) \\
&= (\mathbf{u}, w)(w\mathbf{v} - \mathbf{v}\times\mathbf{u}, \mathbf{v}\cdot\mathbf{u}) \\
&= \left(w(w\mathbf{v} - \mathbf{v}\times\mathbf{u}) + (\mathbf{v}\cdot\mathbf{u})\mathbf{u} + \mathbf{u}\times(w\mathbf{v} - \mathbf{v}\times\mathbf{u}), w(\mathbf{v}\cdot\mathbf{u}) - \mathbf{u}\cdot(w\mathbf{v} - \mathbf{v}\times\mathbf{u}) \right) \\
&= \left((w^2 - \mathbf{u}\cdot\mathbf{u})\mathbf{v} + 2(\mathbf{u}\cdot\mathbf{v})\mathbf{u} + 2w(\mathbf{u}\times\mathbf{v}), 0 \right)
\end{aligned}
$$

$$
\begin{aligned}
\mathbf{qpq}^{-1} &= (\cos^2\theta - \sin^2\theta)\mathbf{v} + 2\sin^2\theta(\mathbf{n}\cdot\mathbf{v})\mathbf{n} + 2\cos\theta\sin\theta(\mathbf{n}\times\mathbf{v}) \\
&= \cos(2\theta)\mathbf{v} + \left(1 - \cos(2\theta)\right)(\mathbf{n}\cdot\mathbf{v})\mathbf{n} + \sin(2\theta)(\mathbf{n}\times\mathbf{v})
\end{aligned}
$$

$$
\begin{aligned}
R_{\hat{\mathbf{n}}}(\mathbf{v}) &= (\hat{\mathbf{n}}\cdot\mathbf{v})\hat{\mathbf{n}} + \cos\theta\mathbf{v}_{\perp} + \sin\theta(\hat{\mathbf{n}}\times\mathbf{v}) \\
&= \cos\theta\mathbf{v} + \left(1 - \cos\theta\right)(\hat{\mathbf{n}}\cdot\mathbf{v})\hat{\mathbf{n}} + \sin\theta(\hat{\mathbf{n}}\times\mathbf{v})
\end{aligned}
$$

Rotation Operator (2)

- Quaternion rotation operator
  - Rotating a vector $\mathbf{v}$ about the axis $\mathbf{n}$ by an angle 2θ

$$R_q(v) = \mathbf{qpq}^{-1} = \mathbf{qpq}^{*}$$
$$= \cos(2\theta)\mathbf{v} + \left(1 - \cos(2\theta)\right)(\mathbf{n}\cdot\mathbf{v})\mathbf{n} + \sin(2\theta)(\mathbf{n}\times\mathbf{v})$$

  - Rotation quaternion

$$\mathbf{q} = \left(\sin\left(\frac{\theta}{2}\right)\mathbf{n}, \cos\left(\frac{\theta}{2}\right)\right)$$

Quaternion Rotation Operator & Matrix

- Quaternion rotation operator to matrix

$$\mathbf{qpq}^{-1} = (w^2 - \mathbf{u}\cdot\mathbf{u})\mathbf{v} + 2(\mathbf{u}\cdot\mathbf{v})\mathbf{u} + 2w(\mathbf{u}\times\mathbf{v}) = \mathbf{vQ}$$

$$= \mathbf{v}\begin{pmatrix} 1-2q_2^2-2q_3^2 & 2q_1q_2+2q_3q_4 & 2q_1q_3-2q_2q_4 \\ 2q_1q_2-2q_3q_4 & 1-2q_1^2-2q_3^2 & 2q_2q_3+2q_1q_4 \\ 2q_1q_3+2q_2q_4 & 2q_2q_3-2q_1q_4 & 1-2q_1^2-2q_2^2 \end{pmatrix}$$

- Matrix to quaternion rotation operator

$$\mathbf{R} = \begin{pmatrix} R_{11} & R_{12} & R_{13} \\ R_{21} & R_{22} & R_{23} \\ R_{31} & R_{32} & R_{33} \end{pmatrix}$$

$$q_1 = \frac{\sqrt{R_{11} - R_{22} - R_{33} + 1}}{2}$$

$$q_2 = \frac{R_{12} + R_{21}}{4q_1}$$

$$q_3 = \frac{R_{13} + R_{31}}{4q_1}$$

$$q_4 = \frac{R_{23} - R_{32}}{4q_1}$$

Kyung Hee University
nize@khu.ac.kr

Composition

- Composition

$$R_{\mathbf{q}}\left(R_{\mathbf{p}}(\mathbf{v})\right)=\mathbf{q}\left(\mathbf{p}\mathbf{v}\mathbf{p}^{-1}\right)\mathbf{q}^{-1}=\left(\mathbf{q}\mathbf{p}\right)\mathbf{v}\left(\mathbf{p}^{-1}\mathbf{q}^{-1}\right)=\left(\mathbf{q}\mathbf{p}\right)\mathbf{v}\left(\mathbf{q}\mathbf{p}\right)^{-1}$$

  - $\mathbf{p}$ and $\mathbf{q}$ are both unit quaternions.
    - $\mathbf{pq}$ is also a unit quaternion.

Quaternion Interpolation (1)

- Dot product for quaternions
  - Since quaternions are 4-tuples of real numbers, geometrically, we can visualize them as 4D vectors. In particular, unit quaternions are 4D unit vectors that lie on the 4D unit sphere.
  - With the exception of the cross product (which is only defined for 3D vectors), the vector math generalizes to 4-space (even n-space).
  - Dot product for quaternions

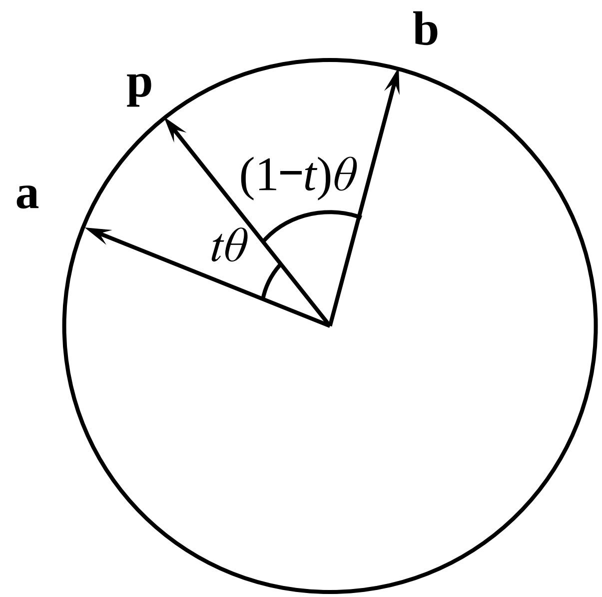
$$\mathbf{p} = (\mathbf{u}, s) \qquad \mathbf{q} = (\mathbf{v}, t)$$

$$\mathbf{p} \cdot \mathbf{q} = \mathbf{u} \cdot \mathbf{v} + st = \|\mathbf{p}\| \|\mathbf{q}\| \cos\theta$$

Quaternion Interpolation (2)

- Quaternion interpolation
  - Interpolation between **a** to **b** by an angle $t\theta$
  - **a**, **b** and **p** are unit quaternions.

$$\mathbf{p} = c_1\mathbf{a} + c_2\mathbf{b}$$
$$\mathbf{a}\cdot\mathbf{p} = c_1\mathbf{a}\cdot\mathbf{a} + c_2\mathbf{a}\cdot\mathbf{b}$$
$$\cos\left(t\theta\right) = c_1 + c_2\cos\theta$$
$$\mathbf{b}\cdot\mathbf{p} = c_1\mathbf{a}\cdot\mathbf{b} + c_2\mathbf{b}\cdot\mathbf{b}$$
$$\cos\left((1-t)\theta\right) = c_1\cos\theta + c_2$$



Quaternion Interpolation (3)

$$\cos\left(t\theta\right) = c_1 + c_2 \cos\theta$$

$$\cos\left((1-t)\theta\right) = c_1 \cos\theta + c_2$$

$$\begin{pmatrix} 1 & \cos\theta \\ \cos\theta & 1 \end{pmatrix} \begin{pmatrix} c_1 \\ c_2 \end{pmatrix} = \begin{pmatrix} \cos\left(t\theta\right) \\ \cos\left((1-t)\theta\right) \end{pmatrix}$$

$$c_1 = \frac{\sin\left((1-t)\theta\right)}{\sin\theta}$$

$$c_2 = \frac{\sin\left(t\theta\right)}{\sin\theta}$$

- Spherical interpolation

$$\mathrm{slerp}(\mathbf{a}, \mathbf{b}, t) = \frac{\sin\left((1-t)\theta\right)\mathbf{a} + \sin\left(t\theta\right)\mathbf{b}}{\sin\theta} \qquad \theta = \cos^{-1}\left(\mathbf{a} \cdot \mathbf{b}\right)$$

Kyung Hee University
nize@khu.ac.kr

Direct Math Quaternion Functions (1)

- Quaternion dot product $\mathbf{Q}_1$ and $\mathbf{Q}_2$
```
XMVECTOR XMQuaternionDot(XMVECTOR Q1, XMVECTOR Q2);
```
- Identity quaternion $(0, 0, 0, 1)$
```
XMVECTOR XMQuaternionIdentity();
```
- Conjugate of the quaternion $\mathbf{Q}$
```
XMVECTOR XMQuaternionConjugate(XMVECTOR Q);
```
- Norm of the quaternion $\mathbf{Q}$
```
XMVECTOR XMQuaternionLength(XMVECTOR Q);
```
- Normalizing a quaternion by treating it as a 4D vector
```
XMVECTOR XMQuaternionNormalize(XMVECTOR Q);
```
- Computing the quaternion product $\mathbf{Q}_1$ and $\mathbf{Q}_2$
```
XMVECTOR XMQuaternionMultiply(XMVECTOR Q1, XMVECTOR Q2);
```

Direct Math Quaternion Functions (2)

- Quaternions from axis-angle rotation representation.

```
XMVECTOR XMQuaternionRotationAxis(XMVECTOR Axis, FLOAT Angle);
```

- Quaternions from axis-angle rotation representation (the axis vector is normalized)

```
XMVECTOR XMQuaternionRotationNormal(XMVECTOR NormalAxis,FLOAT
Angle);
```

- Quaternion from a rotation matrix

```
XMVECTOR XMQuaternionRotationMatrix(XMMATRIX M);
```

- Rotation matrix from a unit quaternion

```
XMMATRIX XMMatrixRotationQuaternion(XMVECTOR Quaternion);
```

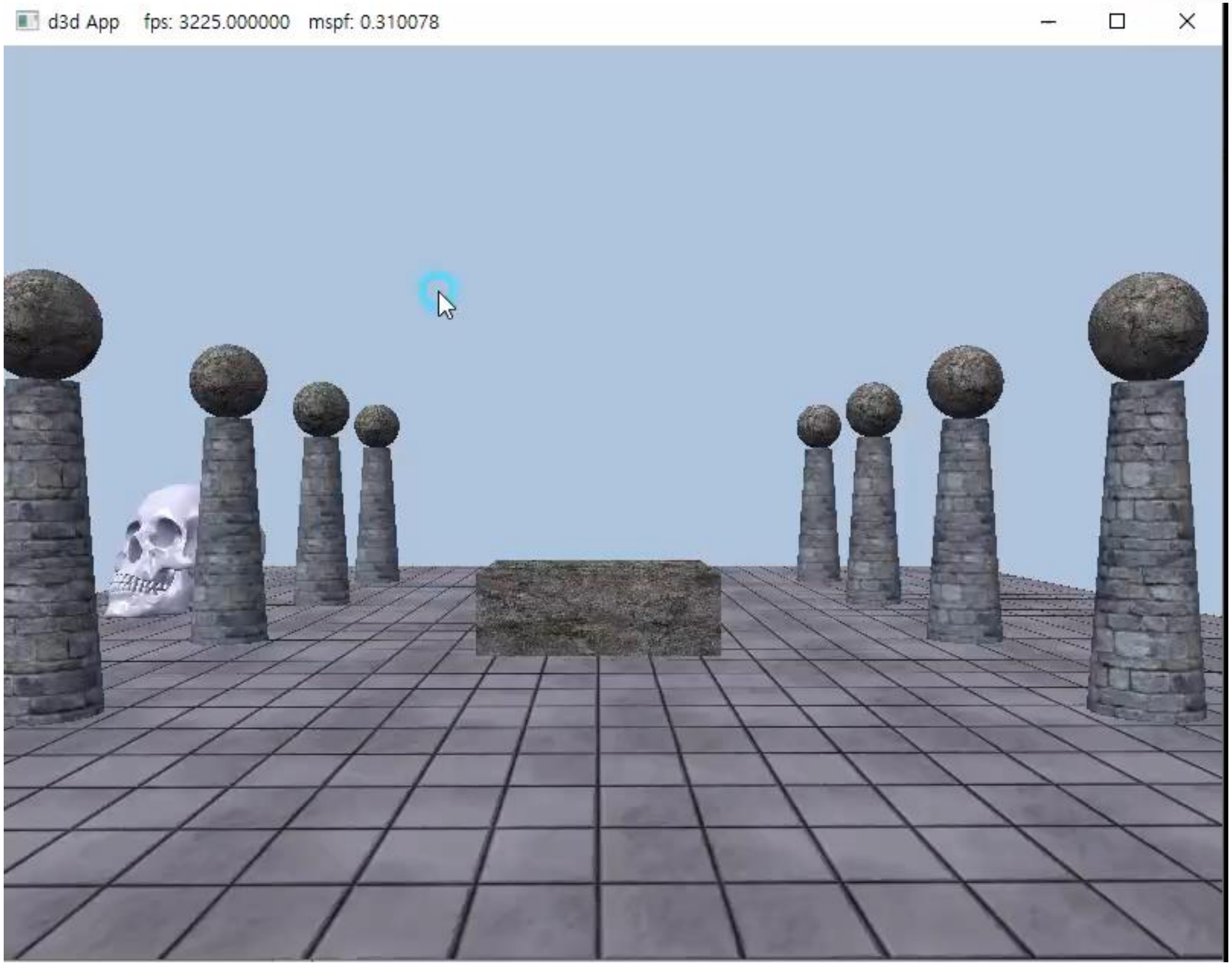
- Extracting the axis and angle rotation representation from the quaternion $\mathbf{Q}$

```
VOID XMQuaternionToAxisAngle(XMVECTOR *pAxis, FLOAT *pAngle,
XMVECTOR Q);
```

- Slerp ($\mathbf{Q}_1$, $\mathbf{Q}_2$, t)

```
XMVECTOR XMQuaternionSlerp(XMVECTOR Q1, XMVECTOR Q2, FLOAT t);
```

Rotation Demo



Keyframe & Animation (1)

```
// AnimationHelper.cpp/h
struct Keyframe {
   Keyframe();
   ~Keyframe();

    float TimePos;
    DirectX::XMFLOAT3 Translation;
    DirectX::XMFLOAT3 Scale;
    DirectX::XMFLOAT4 RotationQuat;
};
struct BoneAnimation {
   float GetStartTime()const;
   float GetEndTime()const;
   void Interpolate(float t, DirectX::XMFLOAT4X4& M)const;
   std::vector<Keyframe> Keyframes;
};

BoneAnimation mSkullAnimation;
```

Keyframe & Animation (2)

```
void QuatApp::DefineSkullAnimation() {
    XMVECTOR q0 = XMQuaternionRotationAxis(
      XMVectorSet(0.0f, 1.0f, 0.0f, 0.0f), XMConvertToRadians(30.0f));
    XMVECTOR q1 = XMQuaternionRotationAxis(
      XMVectorSet(1.0f, 1.0f, 2.0f, 0.0f), XMConvertToRadians(45.0f));
    XMVECTOR q2 = XMQuaternionRotationAxis(
      XMVectorSet(0.0f, 1.0f, 0.0f, 0.0f), XMConvertToRadians(-30.0f));
    XMVECTOR q3 = XMQuaternionRotationAxis(
      XMVectorSet(1.0f, 0.0f, 0.0f, 0.0f), XMConvertToRadians(70.0f));

    mSkullAnimation.Keyframes.resize(5);
    mSkullAnimation.Keyframes[0].TimePos = 0.0f;
    mSkullAnimation.Keyframes[0].Translation
      = XMFLOAT3(-7.0f, 0.0f, 0.0f);
    mSkullAnimation.Keyframes[0].Scale = XMFLOAT3(0.25f, 0.25f, 0.25f);
    XMStoreFloat4(&mSkullAnimation.Keyframes[0].RotationQuat, q0);

    mSkullAnimation.Keyframes[1].TimePos = 2.0f;
    mSkullAnimation.Keyframes[1].Translation
      = XMFLOAT3(0.0f, 2.0f, 10.0f);
    mSkullAnimation.Keyframes[1].Scale = XMFLOAT3(0.5f, 0.5f, 0.5f);
    XMStoreFloat4(&mSkullAnimation.Keyframes[1].RotationQuat, q1);
```


Kyung Hee University
nize@khu.ac.kr

Keyframe & Animation (3)

```
    mSkullAnimation.Keyframes[2].TimePos = 4.0f;
    mSkullAnimation.Keyframes[2].Translation
      = XMFLOAT3(7.0f, 0.0f, 0.0f);
    mSkullAnimation.Keyframes[2].Scale = XMFLOAT3(0.25f, 0.25f, 0.25f);
    XMStoreFloat4(&mSkullAnimation.Keyframes[2].RotationQuat, q2);

    mSkullAnimation.Keyframes[3].TimePos = 6.0f;
    mSkullAnimation.Keyframes[3].Translation
      = XMFLOAT3(0.0f, 1.0f, -10.0f);
    mSkullAnimation.Keyframes[3].Scale = XMFLOAT3(0.5f, 0.5f, 0.5f);
    XMStoreFloat4(&mSkullAnimation.Keyframes[3].RotationQuat, q3);

    mSkullAnimation.Keyframes[4].TimePos = 8.0f;
    mSkullAnimation.Keyframes[4].Translation
      = XMFLOAT3(-7.0f, 0.0f, 0.0f);
    mSkullAnimation.Keyframes[4].Scale = XMFLOAT3(0.25f, 0.25f, 0.25f);
    XMStoreFloat4(&mSkullAnimation.Keyframes[4].RotationQuat, q0);
}
```

Keyframe & Animation (4)

```
// AnimationHelper.cpp/h
void BoneAnimation::Interpolate(float t, XMFLOAT4X4& M) const {
    if( t <= Keyframes.front().TimePos ) {
        XMVECTOR S = XMLoadFloat3(&Keyframes.front().Scale);
        XMVECTOR P = XMLoadFloat3(&Keyframes.front().Translation);
        XMVECTOR Q = XMLoadFloat4(&Keyframes.front().RotationQuat);

        XMVECTOR zero = XMVectorSet(0.0f, 0.0f, 0.0f, 1.0f);
        XMStoreFloat4x4(&M, XMMatrixAffineTransformation(
            S, zero, Q, P));
    }
    else if( t >= Keyframes.back().TimePos ) {
        XMVECTOR S = XMLoadFloat3(&Keyframes.back().Scale);
        XMVECTOR P = XMLoadFloat3(&Keyframes.back().Translation);
        XMVECTOR Q = XMLoadFloat4(&Keyframes.back().RotationQuat);

        XMVECTOR zero = XMVectorSet(0.0f, 0.0f, 0.0f, 1.0f);
        XMStoreFloat4x4(&M, XMMatrixAffineTransformation
            (S, zero, Q, P));
    }
// XMMATRIX XM_CALLCONV XMMatrixAffineTransformation (
//     FXMVECTOR Scaling,     FXMVECTOR RotationOrigin,
//     FXMVECTOR RotationQuaternion,     GXMVECTOR Translation);
```

Keyframe & Animation (5)

```
else { // t is between two key frame, so interpolate.
    for(UINT i = 0; i < Keyframes.size()-1; ++i) {
        if( t >= Keyframes[i].TimePos
            && t <= Keyframes[i+1].TimePos )
        {
            float lerpPercent = (t - Keyframes[i].TimePos)
                / (Keyframes[i+1].TimePos - Keyframes[i].TimePos);

            XMVECTOR s0 = XMLoadFloat3(&Keyframes[i].Scale);
            XMVECTOR s1 = XMLoadFloat3(&Keyframes[i+1].Scale);

            XMVECTOR p0 = XMLoadFloat3(&Keyframes[i].Translation);
            XMVECTOR p1 = XMLoadFloat3(&Keyframes[i+1].Translation);
```


Kyung Hee University
nize@khu.ac.kr

Keyframe & Animation (6)

```
            XMVECTOR q0 = XMLoadFloat4(&Keyframes[i].RotationQuat);
            XMVECTOR q1 = XMLoadFloat4(&Keyframes[i+1].RotationQuat);

            XMVECTOR S = XMVectorLerp(s0, s1, lerpPercent);
            XMVECTOR P = XMVectorLerp(p0, p1, lerpPercent);
            XMVECTOR Q = XMQuaternionSlerp(q0, q1, lerpPercent);

            XMVECTOR zero = XMVectorSet(0.0f, 0.0f, 0.0f, 1.0f);
            XMStoreFloat4x4(&M, XMMatrixAffineTransformation(S,
               zero, Q, P));
            break;
         }
      }
   }
}
```

Keyframe & Animation (7)

```
void QuatApp::Update(const GameTimer& gt) {
    OnKeyboardInput(gt);

    mAnimTimePos += gt.DeltaTime();
    if(mAnimTimePos >= mSkullAnimation.GetEndTime()) {
        // Loop animation back to beginning.
        mAnimTimePos = 0.0f;
    }

    mSkullAnimation.Interpolate(mAnimTimePos, mSkullWorld);
    mSkullRitem->World = mSkullWorld;
    mSkullRitem->NumFramesDirty = gNumFrameResources;

// …
}
```