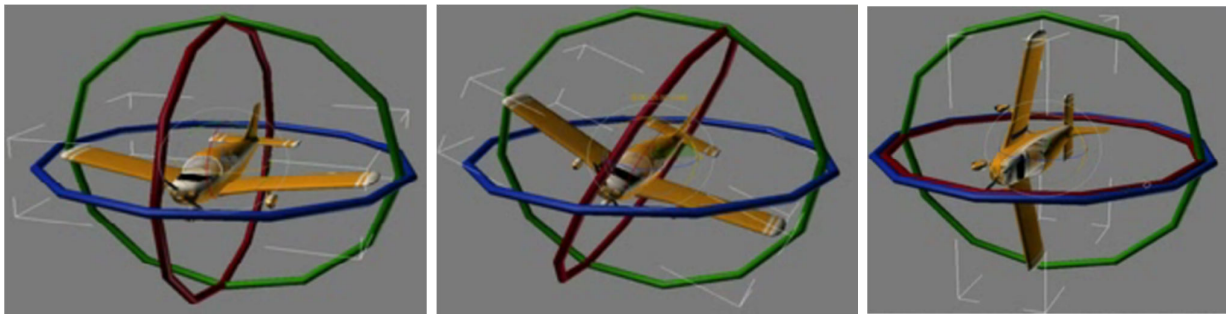
Universidade Federal de Santa Maria
Departamento de Eletrônica e Computação
Prof. Cesar Tadeu Pozzer
Disciplina: Computação Gráfica Avançada
pozzer@inf.ufsm.br
03/01/2013

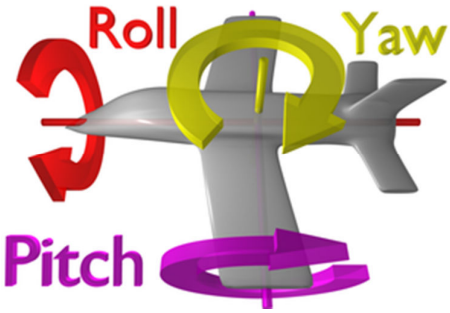
Rotação em eixo arbitrário

1. Gimbal Lock

Quando se usa ângulos de Euler para fazer rotações, pode ocorrer que, após algumas operações, dois eixos podem ficar alinhados, como mostrado na seguinte figura. Este problema também é muito comum com programas de modelagem 3D como Maya, 3Dstudio, dentre outros. Quando isso acontece, perde-se um grau de liberdade. Detalhes da origem do termo "Gimbal lock" podem ser vistos em [19, 21].



http://es.youtube.com/watch?v=rrUCBOlJdt4



http://en.wikipedia.org/wiki/Yaw,_pitch,_and_roll

Sabendo-se que a concatenação de transformações ao se usar ângulos de Euler ocorre na ordem $R_z R_y R_x$, se a primeira rotação for em y, apenas o eixo x é alterado (que passa a ser igual ao z, o que resulta no *gimbal lock*). Uma rotação inicial em x não muda os eixos de rotação y e z. A rotação em z muda os eixos x e y, ou seja, rotacionar em x equivale a rotacionar em y e vice-versa. Ao se usar **quatérnio**, qualquer rotação sempre ocorre no eixo que foi especificado, independente das transformações que a precederam, o que evita o surgimento do *gimbal lock*. Entretanto, para modelagem 3D, torna mais complexo a visualização das rotações necessárias.

Uma situação clara deste problema pode ser visualizada caso realize-se uma rotação de $\theta_y = 90^o$. Aplicando-se após uma rotação de $\theta_z = -90^o$ ou $\theta_x = 90^o$ obtém o mesmo resultado, ou seja, perdeu-se um grau de liberdade, pois rotações em z ou x têm o mesmo efeito. Pode verificar isso calculando-se as matrizes de transformação.

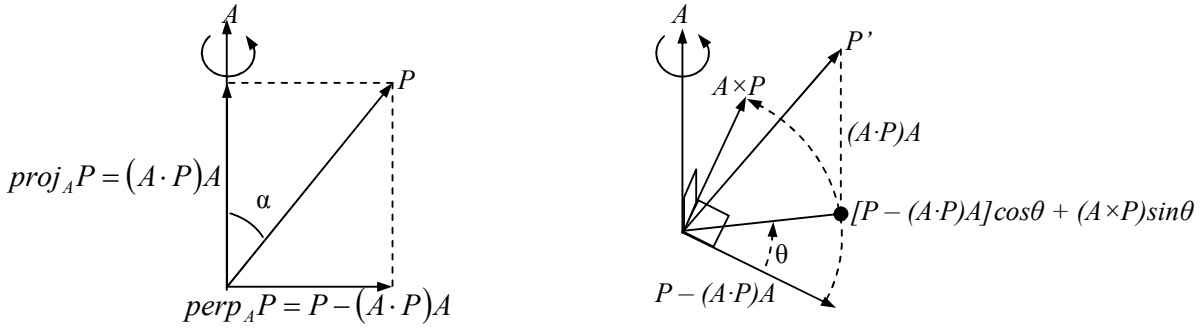
```
glRotatef(camera_roll,      0.0f, 0.0f, 1.0f); // Rotate X, Y, Z
glRotatef(camera_yaw,        0.0f, 1.0f, 0.0f);
glRotatef(camera_pitch,     1.0f, 0.0f, 0.0f);
```

$$R_y(90^0)=\begin{bmatrix}0&0&1\\0&1&0\\1&0&0\end{bmatrix}\quad R_x(90^0)=\begin{bmatrix}1&0&0\\0&0&1\\0&-1&0\end{bmatrix}\quad R_z(-90^0)=R_z(270^0)=\begin{bmatrix}0&-1&0\\1&0&0\\0&0&1\end{bmatrix}$$

$$R_yR_x=R_zR_y=\begin{bmatrix}0&0&1\\0&1&0\\1&0&0\end{bmatrix}\begin{bmatrix}1&0&0\\0&0&1\\0&-1&0\end{bmatrix}=\begin{bmatrix}0&-1&0\\1&0&1\\0&0&1\end{bmatrix}\begin{bmatrix}0&0&1\\0&1&0\\1&0&0\end{bmatrix}=\begin{bmatrix}0&-1&0\\0&0&1\\1&-1&0\end{bmatrix}$$

2. Uso de matrizes de transformação

Para se fazer a rotação em um eixo $A = (x,y,z)$ **unitário** deve-se inicialmente decompor o vetor P em suas componentes **paralela** $(proj_AP)$ e **perpendicular** $(perp_AP)$ ao eixo A, conforme mostrado na seguinte figura. Deve-se observar que a **componente paralela se mantém constante** durante a rotação em relação ao eixo A. Desta forma, para se fazer a rotação, deve-se somente calcular a rotação do vetor $perp_AP$ em relação a A. Para isso, deve-se ter uma base no plano perpendicular ao eixo A. Esta base pode ser definida pelos vetores $(proj_AP)$ e $A\times P$.



Sabe-se que $P\cdot A=|P||A|\cos\alpha$, ou seja, a norma do vetor projetado é dada por $|P|\cos\alpha=\dfrac{P\cdot A}{|A|}$. Para se obter um vetor na direção do vetor A, deve-se multiplicar pelo vetor unitário A, ou seja,

$proj_AP=\dfrac{P\cdot A}{|A|}\dfrac{A}{|A|}=\dfrac{P\cdot A}{|A|^2}A$, para o caso do vetor A não ser unitário.

A rotação de $perp_AP$ no plano definido pelos vetores $P – (A\cdot P)A$ e $A\times P$ por um ângulo θ é dado por

$$[P-(A\cdot P)A]\cos\theta+(A\times P)\sin\theta$$

Sabe-se que a norma do vetor resultante do produto vetorial de dois vetores A e P é dada por $|A\times P|=|A||P|\sin\alpha$. Como $|A|=1$, temos que $|A\times P|=|P|\sin\alpha$. Como $|perp_AP|=|P|\sin\alpha$, a base do sistema de rotação tem dois vetores com a mesma norma, não sendo necessário normalizá-los.

Expandindo-se o termo $[P-(A\cdot P)A]\cos\theta$ obtém-se

$$P\cos\theta-(A\cdot P)A\cos\theta$$

Adicionando-se $proj_AP$ obtém-se P' como

$$P' = P\cos\theta - (A \cdot P)A\cos\theta + (A \times P)\sin\theta + (A \cdot P)A$$
$$P' = P\cos\theta + (A \times P)\sin\theta + (A \cdot P)A(1 - \cos\theta)$$

Em notação matricial temos

$$P = \begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} P_x \\ P_y \\ P_z \end{bmatrix}$$

$$A \times P = \begin{bmatrix} 0 & -A_z & A_y \\ A_z & 0 & -A_x \\ -A_y & A_x & 0 \end{bmatrix} \begin{bmatrix} P_x \\ P_y \\ P_z \end{bmatrix}$$

$$A(A \cdot P) = proj_A P = \frac{P \cdot A}{|A|^2} A = \begin{bmatrix} A_x^2 & A_x A_y & A_x A_z \\ A_x A_y & A_y^2 & A_y A_z \\ A_x A_z & A_y A_z & A_z^2 \end{bmatrix} \begin{bmatrix} P_x \\ P_y \\ P_z \end{bmatrix}$$

Ou seja, $P' = P\cos\theta + (A \times P)\sin\theta + (A \cdot P)A(1 - \cos\theta)$ pode ser expresso como

$$P' = \begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{bmatrix} P\cos\theta + \begin{bmatrix} 0 & -A_z & A_y \\ A_z & 0 & -A_x \\ -A_y & A_x & 0 \end{bmatrix} P\sin\theta + \begin{bmatrix} A_x^2 & A_x A_y & A_x A_z \\ A_x A_y & A_y^2 & A_y A_z \\ A_x A_z & A_y A_z & A_z^2 \end{bmatrix} P(1 - \cos\theta)$$

Combinando os termos e considerando *c = cos(θ)* e *s = sin(θ)*, obtém a matriz de rotação $R_A(\theta)$

$$P' = \left(\begin{bmatrix} c & 0 & 0 \\ 0 & c & 0 \\ 0 & 0 & c \end{bmatrix} + \begin{bmatrix} 0 & -A_z s & A_y s \\ A_z s & 0 & -A_x s \\ -A_y s & A_x s & 0 \end{bmatrix} + \begin{bmatrix} A_x^2(1-c) & A_x A_y(1-c) & A_x A_z(1-c) \\ A_x A_y(1-c) & A_y^2(1-c) & A_y A_z(1-c) \\ A_x A_z(1-c) & A_y A_z(1-c) & A_z^2(1-c) \end{bmatrix} \right) P$$

$$P' = R_A(\theta)P$$

$$R_A(\theta) = \begin{bmatrix} A_x^2(1-c)+c & A_x A_y(1-c)+A_z s & A_x A_z(1-c)-A_y s \\ A_x A_y(1-c)-A_z s & A_y^2(1-c)+c & A_y A_z(1-c)+A_x s \\ A_z A_x(1-c)+A_y s & A_y A_z(1-c)-A_x s & A_z^2(1-c)+c \end{bmatrix}$$

3. Quatérnios

Números complexos

Sabemos que uma rotação no plano euclidiano pode ser expressa por uma matriz de transformação $R(\theta) = \begin{pmatrix} \cos\theta & -\sin\theta \\ \sin\theta & \cos\theta \end{pmatrix}$. Como as funções *sin* e *cos* têm período de 2π, $R(\theta) = R(\theta + 2k\pi)$. Como o espaço de rotações no plano pode ser representado pelo circulo unitário, qualquer ponto *z* deste círculo pode ser representado por um número complexo *z=a+bi* da seguinte forma

$$z = e^{i\theta_z} = \cos\theta_z + i\sin\theta_z \qquad \text{(fórmula de Euler [13])}$$

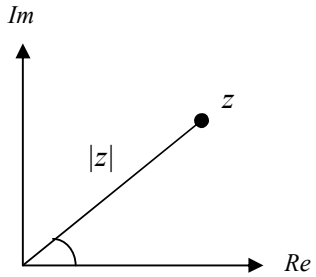
ou seja, a parte real $a = \mathrm{Re}(z) = \cos\theta_z$ representa o eixo x, e a parte imaginária $b = \mathrm{Im}(z) = \sin\theta_z$ o eixo y. Neste caso *z* é considerado um complexo unitário, pois $|z| = \sqrt{z\bar{z}} = \sqrt{a^2 + b^2} = 1$, onde $\bar{z} = a - bi$ é o conjugado de $z = a + bi$.

De forma mais genérica um ponto arbitrário *p = (x,y)* do plano, em sua forma polar, pode ser expresso como

$$p = re^{i\theta_p}$$

onde

$$r = \sqrt{x^2 + y^2}, \quad \theta_p = arctg\left(\frac{y}{x}\right)$$



O inverso de um número complexo é dado por

$$z^{-1} = \frac{1}{z} = \frac{1}{c + di}\frac{c - di}{c - di} = \frac{\bar{z}}{c^2 + d^2} = \frac{\bar{z}}{|z|^2}$$

Quatérnios

Expandindo-se para uma situação no R^4, pode-se definir uma base canônica representada por {1, i, j, k}, onde 1 = (1,0,0,0), i = (0,1,0,0), j = (0,0,1,0), k = (0,0,0,1). Neste caso, *i,j,k* são componentes complexas, e esta estrutura algébrica é conhecida como **quatérnio**, uma extensão dos números complexos. Historicamente, os quatérnios foram criados por William Rowan Hamilton em 1843 e somente em 1985, Shoemake os introduziu na Computação Gráfica. Atualmente são largamente utilizados em aplicações gráficas que fazem uso de rotações, como forma de evitar o *gimbal lock* e para facilitar interpolação entre rotações.

Um quatérnio pode ser visto como um vetor 4D que possui a seguinte forma:

$$q = \langle w, x, y, z \rangle = w + xi + yj + zk$$

Em representação vetorial pode-se escrever ***q*** = *s* + *v*, onde *s* representa a parte **escalar** (componente *w* de **q**) e *v* representa a parte **vetorial** (componentes *x*, *y* e *z* de **q**).

Valem as seguintes regras para multiplicação de componentes imaginárias:

$i^2 = j^2 = k^2 = -1$
$ij = -ji = k$
$jk = -kj = i$
$ki = -ik = j$

Propriedades de Quatérnios

A soma de dois quatérnios $\mathbf{q_1}$ e $\mathbf{q_2}$ é associativa e é dada por (s_1+s_2, v_1+v_2).

A multiplicação por um escalar *a* por um quatérnio **q** é dada por aq = (aw, av).

O conjugado de um quatérnio **q** = (s, v), representado por q^* ou $\bar{q}$, é dado por

$$q^* = \bar{q} = w - xi - yj - zk$$

$$q^* = \bar{q} = s - v$$

O produto interno (escalar) de dois quatérnios é dado por

$$q_1 \cdot q_2 = w_1 w_2 + \mathrm{x}_1 x_2 + y_1 y_2 + z_1 z_2$$

A norma de um quatérnio é dada por

$$|q|^2 = q\bar{q} = \bar{q}q = (s,v)(s,-v) = s^2 + v \cdot v = w^2 + \mathrm{x}^2 + \mathrm{y}^2 + \mathrm{z}^2$$

ou

$$|q| = \sqrt{w^2 + x^2 + y^2 + z^2}$$

Um quatérnio **q** é unitário quando |*q*| = 1. Para normalizar um quatérnio deve-se dividir cada componente pela norma

$$q' = \frac{s+v}{|q|} = \left\langle \frac{w}{|q|}, \frac{x}{|q|}, \frac{y}{|q|}, \frac{z}{|q|} \right\rangle$$

O inverso de quatérnio não zero ($|q| \neq 0$) é dada por

$$q^{-1} = \frac{\bar{q}}{q^2}$$

Se o quatérnio for unitário, temos $q^{-1} = \bar{q}$

```
void Quaternion::normalise()
{
   // Don't normalize if we don't have to
   float mag2 = w * w + x * x + y * y + z * z;
   if (fabs(mag2 - 1.0f) > TOLERANCE) {
      float mag = sqrt(mag2);
      w /= mag;
      x /= mag;
      y /= mag;
      z /= mag;
   }
}

Quaternion Quaternion::getConjugate() //somente para quaternio unitário
{
   return Quaternion(-x, -y, -z, w);
}
```

A multiplicação de quatérnios não é comutativa, ou seja, $\mathbf{q_1 q_2}$ é diferente de $\mathbf{q_2 q_1}$. Dados dois quatérnios $\mathbf{q_1} = w_1 + x_1 i + y_1 j + z_1 k$ e $\mathbf{q_2} = w_2 + x_2 i + y_2 j + z_2 k$, a multiplicação é dada por

$$\begin{aligned} \mathbf{q_1 q_2} = {} & (w_1 w_2 - x_1 x_2 - y_1 y_2 - z_1 z_2) \\ & + (w_1 x_2 + x_1 w_2 + y_1 z_2 - z_1 y_2) i \\ & + (w_1 y_2 - x_1 z_2 + y_1 w_2 + z_1 x_2) j \\ & + (w_1 z_2 + x_1 y_2 - y_1 x_2 - z_1 w_2) k \end{aligned}$$

Em representação vetorial, $\mathbf{q_1} = \mathrm{s}_1 + \mathrm{v}_1$ e $\mathbf{q_2} = \mathrm{s}_2 + \mathrm{v}_2$, temos

$$\mathbf{q_1q_2} = s_1s_2 - v_1.v_2 + s_1v_2 + s_2v_1 + v_1 \times v_2$$

ou

$$\mathbf{q_1q_2} = (s_1s_2 - v_1.v_2,\ s_1v_2 + s_2v_1 + v_1 \times v_2)$$ (partes escalar e vetorial)

```
Quaternion Quaternion::operator* (const Quaternion &rq) const
{
   // the constructor takes its arguments as (x, y, z, w)
   return Quaternion(w*rq.x + x*rq.w + y*rq.z - z*rq.y,
                     w*rq.y + y*rq.w + z*rq.x - x*rq.z,
                     w*rq.z + z*rq.w + x*rq.y - y*rq.x,
                     w*rq.w - x*rq.x - y*rq.y - z*rq.z);
}
```

Rotação com Quaternions

Para se poder usar quatérnios para representar rotação de um ponto *P* em eixo arbitrário, deve-se encontrar uma função $\varphi(P)$, que preserve ângulos, comprimentos e direção (*handedness*), ou seja $|\varphi(P)| = |P|$, $\varphi(P_1)\cdot\varphi(P_2) = P_1 \cdot P_2$ e $\varphi(P_1)\times\varphi(P_2) = \varphi(P_1 \times P_2)$.

A classe de funções do tipo $\varphi_q(P) = qPq^{-1}$, onde **q** é um quatérnio não nulo, satisfaz os requisitos para definição de rotações. A demonstração pode ser vista em [1].

O seguinte quatérnio **q** faz com que $P' = qPq^{-1}$ seja um ponto rotacionado de um ângulo θ em relação a um eixo arbitrário *A*. A demonstração pode ser vista em [1].

$$q = \cos\frac{\theta}{2} + A\sin\frac{\theta}{2}$$

Deve-se observar que se o vetor *A* for normalizado, o quatérnio será unitário

$$\begin{aligned} |q|^2 = q\bar{q} &= (\cos\theta, A\sin\theta)(\cos\theta, -A\sin\theta) \\ &= \cos^2\theta + \sin^2\theta(A\cdot A) \\ &= \cos^2\theta + \sin^2\theta \\ &= 1 \end{aligned}$$

Para aplicar a equação $P' = qPq^{-1}$ deve-se ter um ponto *P* qualquer do espaço R^3, representado por meio de um quatérnio com parte real nula, ou seja, *P = (0,P)*. A parte real deste produto é nula e a parte imaginária corresponde ao ponto rotacionado, ou seja, o ponto *P'* é dado por *P' = (0,P')*. O quatérnio que corresponde a rotação nula é **q** = (1, 0A) = (1,0) = (1,0,0,0).

Como exemplo, suponha a rotação do ponto *P = (2,0,1)* em um ângulo de 90º em torno do eixo z. Neste caso, deve-se usar o quatérnio $q = (\cos 45^0, \sin 45^0(0,0,1)) = \left(\left(\frac{1}{\sqrt{2}}\right), \left(\frac{1}{\sqrt{2}}\right)(0,0,1)\right) = \left(\left(\frac{1}{\sqrt{2}}\right), \left(0,0,\frac{1}{\sqrt{2}}\right)\right)$ e o ponto P = (0,(2,0,1)) expresso na forma de quatérnio. Observe que o quatérnio P não precisa ser unitário.

$$\begin{aligned} P' &= qPq^{-1} \\ &= \left(\left(\frac{1}{\sqrt{2}}\right), \left(0,0,\frac{1}{\sqrt{2}}\right)\right) * (0,(2,0,1)) * \left(\left(\frac{1}{\sqrt{2}}\right), \left(0,0,-\frac{1}{\sqrt{2}}\right)\right) \end{aligned}$$

$$= \left(0-0-0-\frac{1}{\sqrt{2}}, \left(\frac{2}{\sqrt{2}}+0+0-0,\ 0-0+0+\frac{2}{\sqrt{2}}, \frac{1}{\sqrt{2}}+0-0-0\right)\right) * \left(\left(\frac{1}{\sqrt{2}}\right), \left(0,0,-\frac{1}{\sqrt{2}}\right)\right)$$

$$= \left(-\frac{1}{\sqrt{2}}, \left(\frac{2}{\sqrt{2}}, \frac{2}{\sqrt{2}}, \frac{1}{\sqrt{2}}\right)\right) * \left(\left(\frac{1}{\sqrt{2}}\right), \left(0,0,-\frac{1}{\sqrt{2}}\right)\right)$$

$$= \left(-\frac{1}{2}-0-0+\frac{1}{2}, \left(0+1-1-0, 0+1+1+0, \frac{1}{2}+0-0+\frac{1}{2}\right)\right)$$

$$= (0,(0,2,1))$$

Como resultado, obteve-se o quatérnio **q** = (0,(0,2,1)). Tomando-se apenas a parte imaginária, tem-se o ponto P´ = (0,2,1), como esperado.

Como o produto de dois quatérnios também representa uma rotação, pode-se concatenar várias rotações em um único quatérnio. O produto $\mathbf{q_1 q_2}$ representa uma rotação primeiro por $\mathbf{q_2}$ seguido por $\mathbf{q_1}$. Uma vez que

$$\begin{aligned} & q_1\left(q_2 P q_2^{-1}\right) q_1^{-1} \\ &= \left(q_1 q_2\right) P\left(q_2^{-1} q_1^{-1}\right) \\ &= \left(q_1 q_2\right) P\left(q_1 q_2\right)^{-1} \\ &= q_r P q_r^{-1} \end{aligned}$$

pode-se concatenar quantos quatérnios se desejar para gerar um único quatérnio representando toda série de rotações. Isso mostra que qualquer conjunto de rotações pode ser reduzida a uma única rotação em um único eixo.

O seguinte exemplo ilustra este processo para concatenação de uma rotação θ_x = -60º seguida de uma rotação θ_y = 90º. Estas rotações são representadas pelos quatérnios

$$q_x = \left(\cos -30^0, \sin -30^0 (1,0,0)\right) = \left(\frac{\sqrt{3}}{2}, -\frac{1}{2}(1,0,0)\right)$$

$$q_y = \left(\cos 45^0, \sin 45^0 (0,1,0)\right) = \left(\frac{1}{\sqrt{2}}, \frac{1}{\sqrt{2}}(0,1,0)\right).$$

Calculando $\mathbf{q_r} = q_y q_x$ obtem-se

$$q_r = \left(\frac{1}{\sqrt{2}}, \frac{1}{\sqrt{2}}(0,1,0)\right)\left(\frac{\sqrt{3}}{2}, -\frac{1}{2}(1,0,0)\right) = \left(\frac{\sqrt{6}}{4}, \frac{\sqrt{10}}{4}\left(-\frac{1}{\sqrt{5}}, \frac{\sqrt{3}}{\sqrt{5}}, \frac{1}{\sqrt{5}}\right)\right)$$

ou seja, estas duas rotações equivalem a rotação de θ = 104.5º em torno do eixo $A = \left(-\frac{1}{\sqrt{5}}, \frac{\sqrt{3}}{\sqrt{5}}, \frac{1}{\sqrt{5}}\right)$, pois $\cos\left(\frac{104.5}{2}\right)^0 = \frac{\sqrt{6}}{4}$.

Uma vantagem menor do uso de quatérnio se refere ao número de operações necessárias para concatenação de transformações. A seguinte tabela ilustra um comparativo com matrizes de rotação.

Método	Armazenamento	# multiplicações	# adições	# total operações
Matriz rotação	9	27	18	45
Quatérnio	4	16	12	28

Para aplicar uma rotação a um ponto, o uso de matriz de rotação necessita menor custo computacional.

Método	# multiplicações	# adições	# total operações
Matriz rotação	9	6	15
Quatérnio	21	18	39

Como multiplicar um quatérnio por outro gasta 16 multiplicações, e como são necessárias duas multiplicações para rotacionar um ponto, deve-se fazer otimizações para reduzir este processamento para apenas 21 multiplicações. O seguinte código ilustra esta otimização [8].

```
Vector3 Quaternion::operator* (const Vector3 &vec) const (revisar)
{
   float t2 =   a*b
   float t3 =   a*c
   float t4 =   a*d
   float t5 =  -b*b
   float t6 =   b*c
   float t7 =   b*d
   float t8 =  -c*c
   float t9 =   c*d
   float t10 = -d*d
   return Vector3( 2*( (t8 + t10)*v1 + (t6 -  t4)*v2 + (t3 + t7)*v3 ) + vec.x,
                   2*( (t4 +  t6)*v1 + (t5 + t10)*v2 + (t9 - t2)*v3 ) + vec.y,
                   2*( (t7 -  t3)*v1 + (t2 +  t9)*v2 + (t5 + t8)*v3 ) + vec.z);
}
```

```
Vector3 Quaternion::operator* (const Vector3 &vec) const
{
        Vector3 vn(vec);
        vn.normalise();

        Quaternion vecQuat, resQuat;
        vecQuat.x = vn.x;
        vecQuat.y = vn.y;
        vecQuat.z = vn.z;
        vecQuat.w = 0.0f;

        resQuat = vecQuat * getConjugate();
        resQuat = *this * resQuat;

        return (Vector3(resQuat.x, resQuat.y, resQuat.z));
}
```

Muitas vezes pode ser necessário representar um quatérnio **unitário** por uma matriz de rotação, especialmente quando se está utilizando alguma API gráfica, como no caso do OpenGL. A transformação envolve o uso de álgebra de matrizes. A demonstração pode ser encontrada em [1].

$$M_q = \begin{pmatrix} 1-2y^2-2z^2 & 2xy-2wz & 2xz+2wy & 0 \\ 2xy+2wz & 1-2x^2-2z^2 & 2yz-2wx & 0 \\ 2xz-2wy & 2yz+2wx & 1-2x^2-2y^2 & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix}$$

Caso o quatérnio não for unitário ($w^2 + x^2 + y^2 + z^2 \neq 1$), deve-se utilizar a seguinte matriz de conversão:

$$M_q = \begin{pmatrix} w^2+x^2+y^2+z^2 & 2xy-2wz & 2xz+2wy & 0 \\ 2xy+2wz & w^2-x^2+y^2-z^2 & 2yz-2wx & 0 \\ 2xz-2wy & 2yz-2wx & w^2-x^2-y^2+z^2 & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix}$$

Deve-se observar que esta matriz está no formato adotado pela API OpenGL (*column-major format*), ou seja, a matriz aparece no lado direito da multiplicação. Deve-se lembrar que a multiplicação de matrizes *RP* equivale a $P^T R^T$. Deve-se calcular a matriz $M_q^{'} = M_q^T$ caso seja adotado matrizes em formato *row-major*.

```
Matrix4 Quaternion::toMatrix() const
{
   float x2 = x * x;
   float y2 = y * y;
```

```
    float z2 = z * z;
    float xy = x * y;
    float xz = x * z;
    float yz = y * z;
    float wx = w * x;
    float wy = w * y;
    float wz = w * z;
    return Matrix4(1.0f - 2.0f * (y2 + z2), 2.0f * (xy - wz), 2.0f * (xz + wy), 0.0f,
                   2.0f * (xy + wz), 1.0f - 2.0f * (x2 + z2), 2.0f * (yz - wx), 0.0f,
                   2.0f * (xz - wy), 2.0f * (yz + wx), 1.0f - 2.0f * (x2 + y2), 0.0f,
                   0.0f, 0.0f, 0.0f, 1.0f);
}
```

Para se obter um quatérnio a partir de uma matriz de rotação deve-se realizar os seguintes cálculos. Deve-se verificar se a expressão $1+m_{00}+m_{11}+m_{22}$, que representa a soma da diagonal principal, também chamada de **traço da matriz**, é maior que zero. Caso contrário, deve-se fazer um tratamento especial. Para isso, consulte [4, 16].

$$w=\frac{\sqrt{1+m_{00}+m_{11}+m_{22}}}{2}$$

$$x=\frac{(m_{21}-m_{12})}{(4w)} \quad y=\frac{(m_{02}-m_{20})}{(4w)} \quad z=\frac{(m_{10}-m_{01})}{(4w)}$$

Outra transformação importante é a conversão de ângulos de Euler para quatérnios. O processo é muito simples. Basta transformar cada ângulo de Euler (θ_x no eixo x - *pitch*, θ_y no eixo y – *yaw* e θ_z no eixo z - *roll*) em um quatérnio e após multiplicar estes quatérnios para obter um quatérnio resultante. Assumindo a ordem de multiplicação de matrizes do OpenGL, temos que a transformação $R_xR_yR_z$ é dada por

$$q_x=(\cos(\theta_x/2),(\sin(\theta_x/2),0,0))$$
$$q_y=(\cos(\theta_y/2),(0,\sin(\theta_y/2),0))$$
$$q_z=(\cos(\theta_z/2),(0,0,\sin(\theta_z/2)))$$
$$q=q_xq_yq_z$$

Para realizar o processo inverso, ou seja, obter os ângulos de Euler e eixo a partir do quatérnio utiliza-se o seguinte algoritmo.

```
void Quaternion::toAxisAngle(Vector3 *axis, float *angle)
{
    float scale = sqrt(x * x + y * y + z * z);
    axis->x = x / scale;
    axis->y = y / scale;
    axis->z = z / scale;
    *angle = acos(w) * 2.0f;
}
```

Deve-se observar que fazendo-se uso de rotação e eixo arbitrário, pode-se obter o mesmo resultado que com o uso de quatérnion.

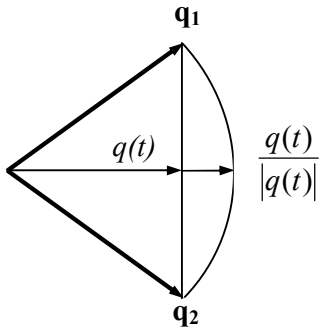
4. Interpolação Linear Esférica - SLERP

A interpolação entre vetores é uma estratégia muito usada para se fazer animações de modelos, considerando-se que cada *keyframe* esteja associada a um vetor direção. A forma mais simples de se fazer a interpolação de dois vetores (ou quatérnios) é por meio da interpolação linear (no plano), como mostrado na seguinte equação.

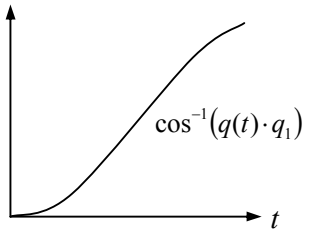
$$q(t) = (1-t)q_1 + tq_2$$

O problema desta solução é que o vetor resultante *q(t)* não mantém a mesma norma durante essa interpolação, da mesma forma como ocorre a animação via o formato MD2. Uma solução é fazer a normalização do vetor resultante, com a seguinte equação.

$$q(t) = \frac{(1-t)q_1 + tq_2}{\left|(1-t)q_1 + tq_2\right|}$$



Mesmo essa solução ainda não é adequada. Pode-se observar que a taxa de variação (velocidade angular) do vetor q(t) não é constante. Como mostrado no seguinte gráfico, a variação é maior no ponto médio entre os vetores q_1 e q_2.



Deste modo, deseja-se achar uma função *q(t)* que interpole os vetores com uma taxa constante e que mantenha norma unitária do vetor resultante.

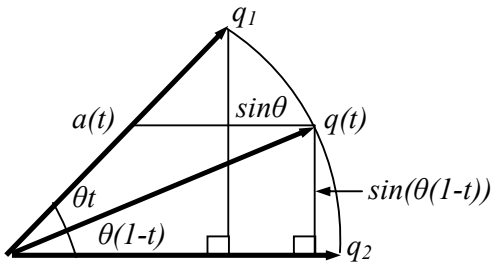
$$q(t) = a(t)q_1 + b(t)q_2$$

Se os vetores q_1 e q_2 forem ortogonais, a solução torna-se bem simples, e é dada por

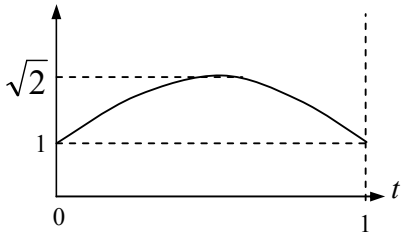
$$q(t) = \cos(t\theta)q_1 + \sin(t\theta)q_2$$

sendo θ o ângulo entre os vetores q_1 e q_2. Porém, se os vetores não forem ortogonais, deve-se utilizar a seguinte equação. A demonstração pode ser vista em [1,27,28,29,30].

$$q(t) = \frac{\sin[\theta(1-t)]}{\sin\theta}q_1 + \frac{\sin[\theta t]}{\sin\theta}q_2$$

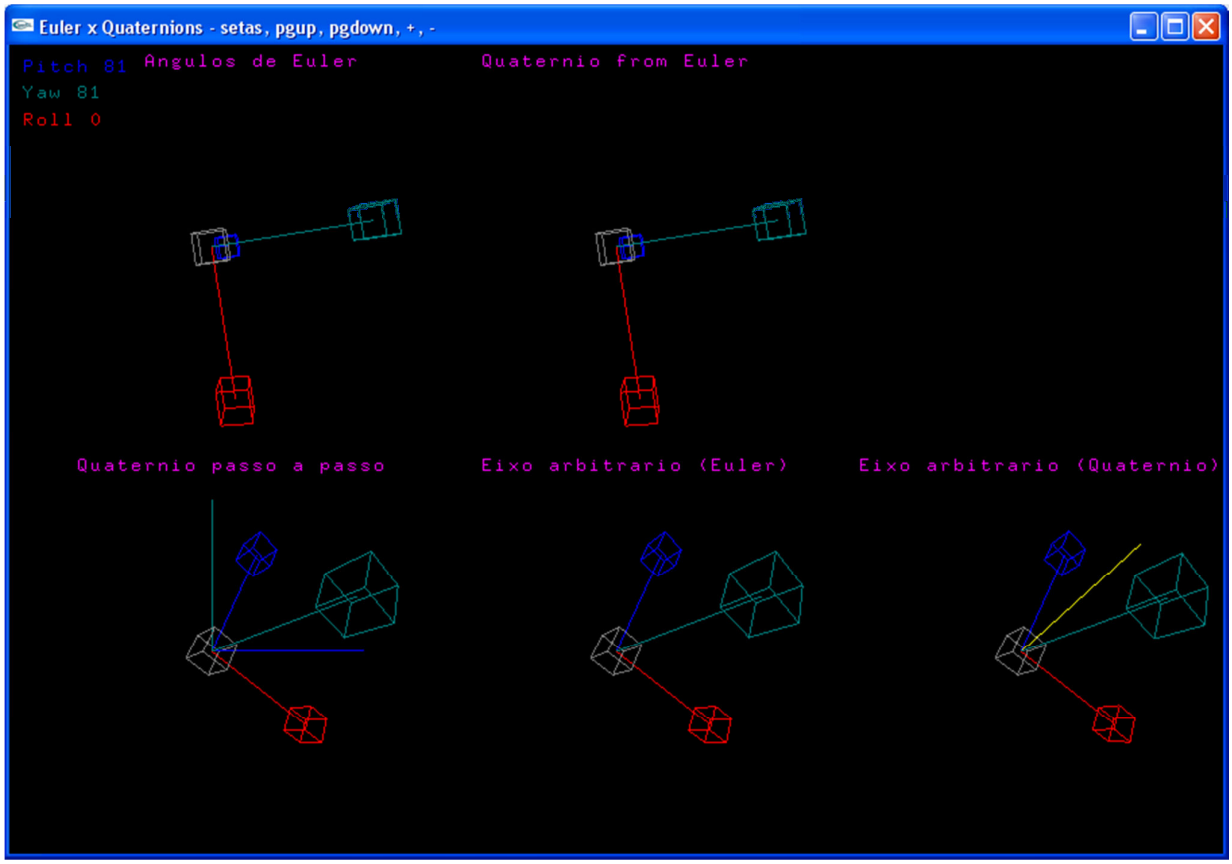


Os termos $\frac{\sin\theta(1-t)}{\sin\theta} + \frac{\sin\theta t}{\sin\theta}$ desta equação geram valores entre [1, $\sqrt{2}$] com *t* variando no intervalo [0,1], como mostrado na seguinte figura. Cada termo isolado tem variação não linear entre [0, 1], justamente para garantir a variação constante na velocidade angular do vetor resultante *q(t)*.



5. Programa Demo

Este programa foi desenvolvido para comprar as diferentes formas de aplicar rotação em eixos. Ele ilustra o uso de ângulos de Euler, quaternions extraídos de ângulos de Euler, rotação via quatérnion, rotação em eixo arbitrário e eixo arbitrário extraído de quatérnion. Por ele, pode-se nitidamente notar que as duas primeiras soluções não tratam o problema de gimbal lock. Esse demo pode ser baixado no site da disciplina.



6. Referências Bibliográficas

[1] Eric Lengyel. Mathematics for 3D Game Programming and Computer Graphics. Charles River Media, 2002.
[2] Gomes, J., Velho, L. *Computação Gráfica, Volume 1*. IMPA, 1998.
[3] Marcelo Gattass, PUC-Rio, Notas de aula.
[4] http://www.j3d.org/matrix_faq/matrfaq_latest.html
[5] http://en.wikipedia.org/wiki/Quaternions
[6] http://www.gamasutra.com/features/19980703/quaternions_01.htm
[7] http://www.genesis3d.com/~kdtop/Quaternions-UsingToRepresentRotation.htm

[8] http://en.wikipedia.org/wiki/Quaternion_rotation
[9] http://www.genesis3d.com/~kdtop/Quaternions-UsingToRepresentRotation.htm
[10] http://www.euclideanspace.com/maths/geometry/rotations/conversions/matrixToQuaternion/index.htm
[11] http://www.nationmaster.com/encyclopedia/Quaternions-and-spatial-rotation
[12] http://www.opengl.org/resources/faq/technical/transformations.htm
[13] http://en.wikipedia.org/wiki/Euler%27s_formula
[14] http://www.ime.unicamp.br/~vaz/4nion.htm
[15] http://en.wikipedia.org/wiki/Slerp
[16] http://www.euclideanspace.com/maths/geometry/rotations/conversions/matrixToQuaternion/index.htm
[17] http://www.gamedev.net/reference/articles/article1095.asp
[18] http://everything2.com/e2node/Gimbal%2520Lock
[19] http://www.allanbrito.com/2007/04/18/gimbal-lock/
[20] http://www.anticz.com/eularqua.htm
[21] http://history.nasa.gov/alsj/gimbals.html
[22] http://www.dhpoware.com/demos/glCamera2.html
[23] http://en.wikipedia.org/wiki/Yaw,_pitch,_and_roll
[24] http://www.cprogramming.com/tutorial/3d/rotation.html
[25] http://www.gamedev.net/reference/articles/article1199.asp
[26] http://www.gamedev.net/community/forums/topic.asp?topic_id=25314
[27] 3D math primer for graphics and game development. Fletcher Dunn,Ian Parberry
[28] Visualizing quatérnions. Andrew J. Hanson
[29] 3D game engine design: A Practical Approach to Real-Time Computer Graphics. David H. Eberly
[30] http://number-none.com/product/Understanding%20Slerp,%20Then%20Not%20Using%20It/