Real-time Graphics

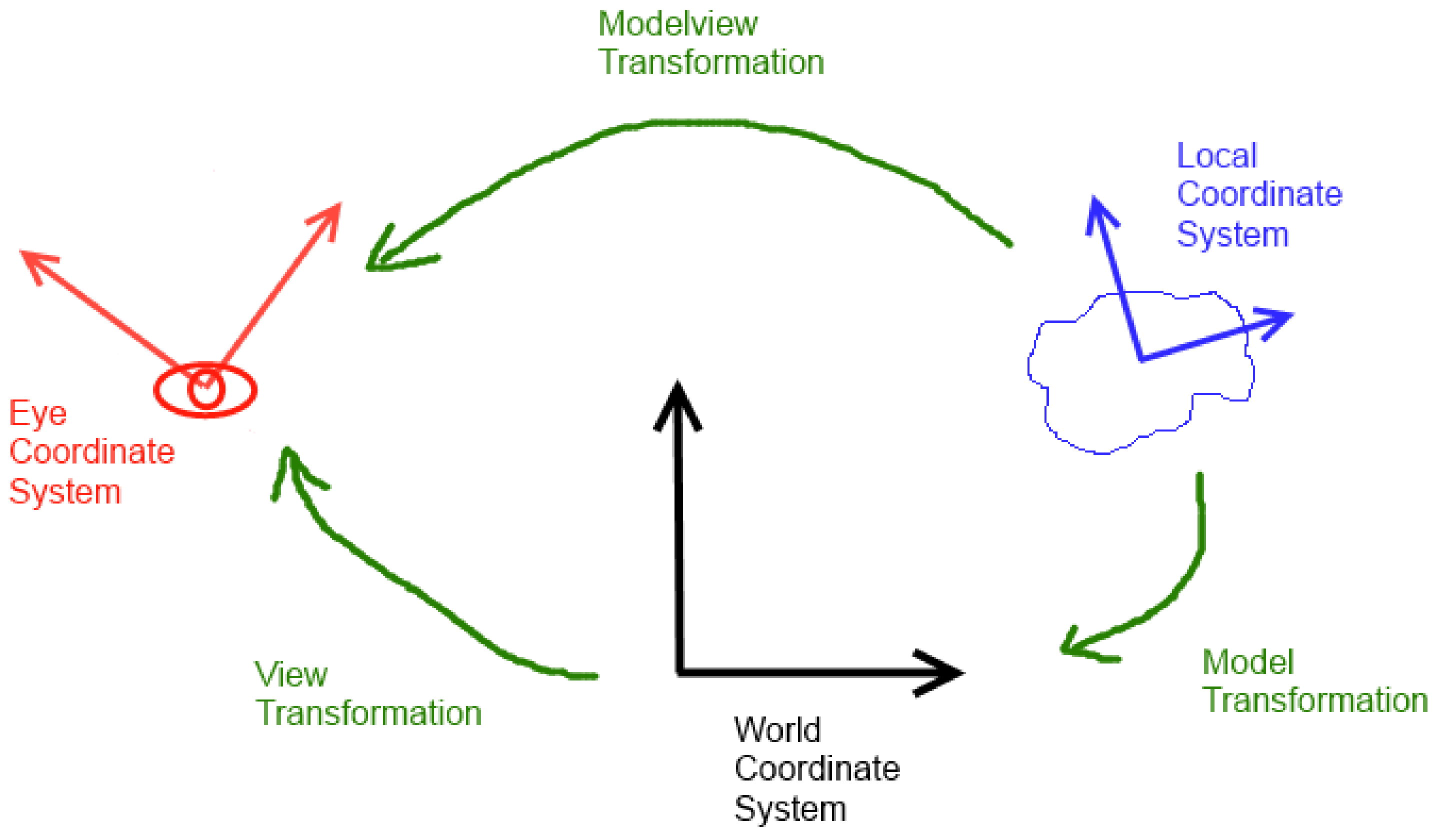
1.5. Object Pose & Animation

Martin Samuelčík
Juraj Starinský

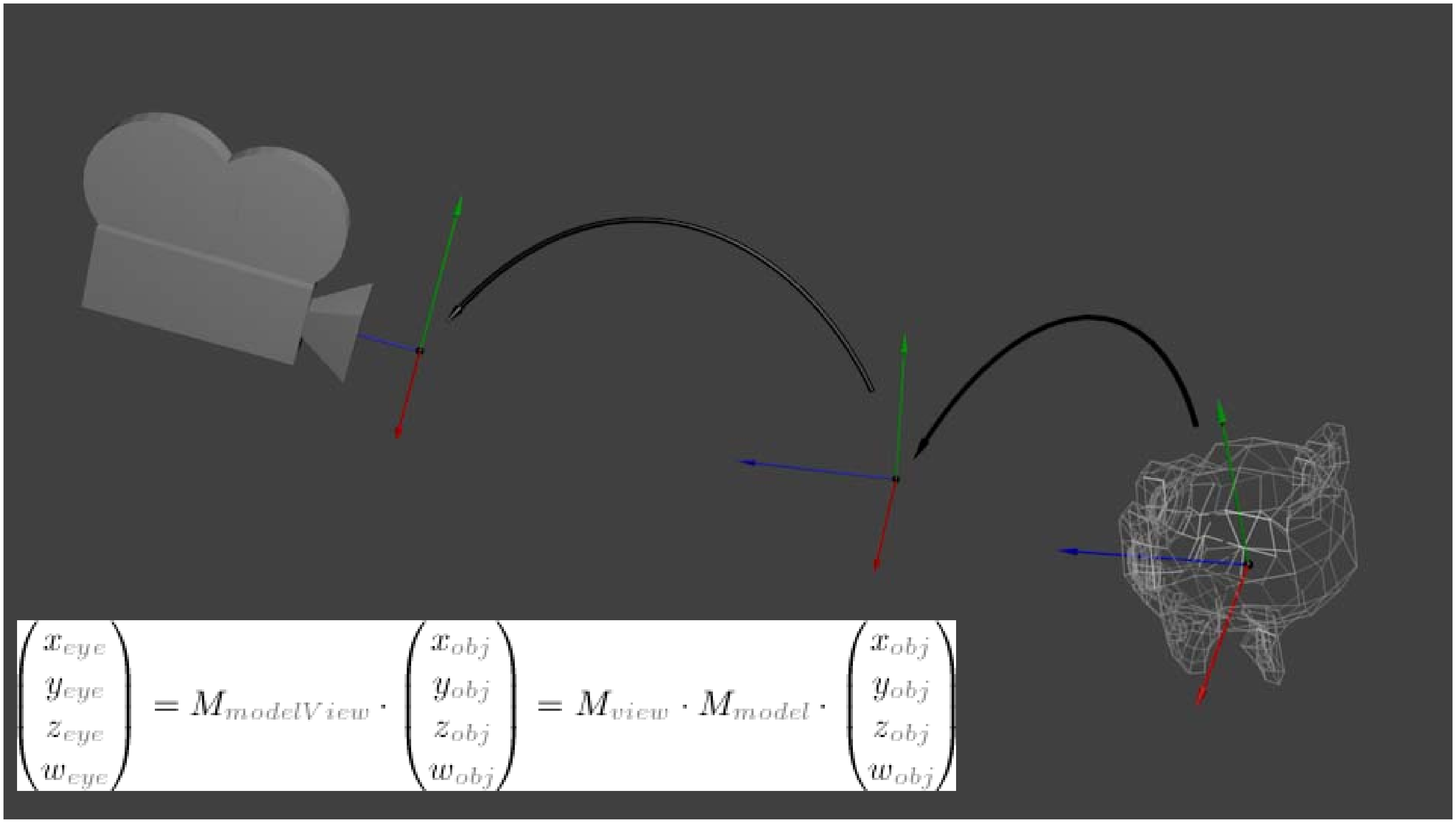
Local Coordinate System

- System for vertex coordinates of object
- Object (model) coordinate system = Object pose
- Usually defines translation, rotation and scale of object – for each object separately
- Represented by
  - Transformation to world coordinate system or transformation to eye coordinate system
  - 4x4 matrix
  - For animations, it is better to represent scale, translation and rotation separately

Local Coordinate System



Local Coordinate System



from http://www.opengl-tutorial.org

Pose matrices

- In OpenGL, transformation is represented as 4x4 matrix
- Column-major order in memory
- $m_0,m_1,m_2,m_4,m_5,m_6,m_8,m_9,m_{10}$ – rotation and scale
- m_{12},m_{13},m_{14} – translation
- m_3,m_7,m_{11} – 0
- m_{15} - 1

$$\begin{pmatrix} m_0 & m_4 & m_8 & m_{12} \\ m_1 & m_5 & m_9 & m_{13} \\ m_2 & m_6 & m_{10} & m_{14} \\ m_3 & m_7 & m_{11} & m_{15} \end{pmatrix}$$

Translation & scale

- Simple representation –as 3D vectors
  - Translation – (X,Y,Z)
  - Scaling – (x,y,z)
- Matrix representation:

$$\begin{bmatrix} 1 & 0 & 0 & X \\ 0 & 1 & 0 & Y \\ 0 & 0 & 1 & Z \\ 0 & 0 & 0 & 1 \end{bmatrix} \qquad \begin{bmatrix} x & 0 & 0 & 0 \\ 0 & y & 0 & 0 \\ 0 & 0 & z & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

Rotation

- Several types of representation
- 3x3 rotation matrix
  - Columns represent coordinate vectors after transformation
  - By adding zeroes and ones, easy conversion to 4x4 matrix
- Euler angles
  - 3 angles for rotation around coordinate axes - Yaw, Roll and Pitch
  - Problem – Gimbal lock
- Quaternion
  - Good for smooth rotation animation
  - Easy rotation around arbitrary axis

Rotation matrix

- Basic rotation matrix when rotating around coordinate axis by angle

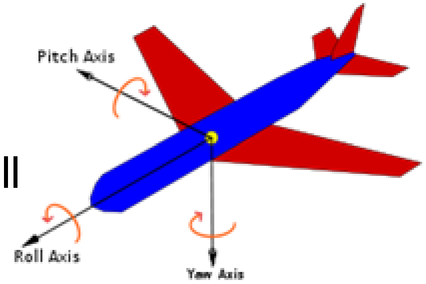
$$R_x(\theta) = \begin{bmatrix} 1 & 0 & 0 \\ 0 & \cos\theta & -\sin\theta \\ 0 & \sin\theta & \cos\theta \end{bmatrix} \qquad R_y(\theta) = \begin{bmatrix} \cos\theta & 0 & \sin\theta \\ 0 & 1 & 0 \\ -\sin\theta & 0 & \cos\theta \end{bmatrix} \qquad R_z(\theta) = \begin{bmatrix} \cos\theta & -\sin\theta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

- Rotation matrix for rotation around arbitrary axis given by direction (u_x, u_y, u_z)

$$R = \begin{bmatrix} \cos\theta + u_x^2(1-\cos\theta) & u_x u_y(1-\cos\theta) - u_z\sin\theta & u_x u_z(1-\cos\theta) + u_y\sin\theta \\ u_y u_x(1-\cos\theta) + u_z\sin\theta & \cos\theta + u_y^2(1-\cos\theta) & u_y u_z(1-\cos\theta) - u_x\sin\theta \\ u_z u_x(1-\cos\theta) - u_y\sin\theta & u_z u_y(1-\cos\theta) + u_x\sin\theta & \cos\theta + u_z^2(1-\cos\theta) \end{bmatrix}.$$

Euler angles

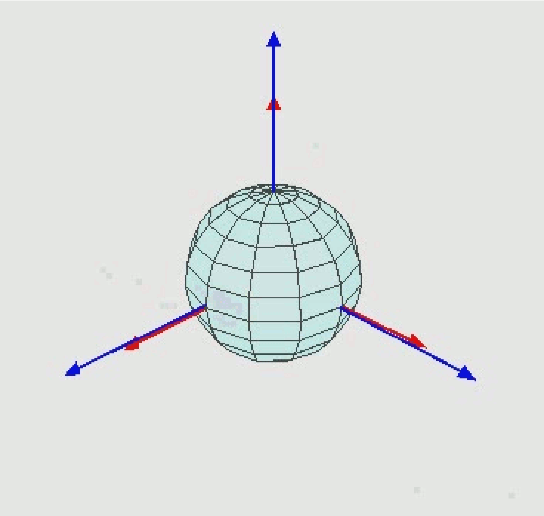
- Composition of 3 elementary rotations – several orders of composition
- Euler angles – (alpha, beta, gama)
- Proper Euler angles formalism
  - Order of rotation angles: - z-x-z, x-y-x, y-z-y, z-y-z, x-z-x, y-x-y
- Tait–Bryan angles formalism
  - Angle names: heading, elevation and bank, or yaw, pitch and roll
  - Order of rotation angles: x-y-z, y-z-x, z-x-y, x-z-y, z-y-x, y-x-z



- Extrinsic rotations
  - rotations about fixed coordinate system
  - z-x-z – rotate about world z-coordinate axis by angle alpha, the rotate around world x-coordinate axis by angle beta, then rotate about world z-coordinate axis by angle gama
- Intrinsic rotations
  - rotations about local coordinate system
  - *z-x’-z″* = z-x-z – rotate about local z-coordinate axis by angle alpha, the rotate around local x-coordinate axis by angle beta, then rotate about local z-coordinate axis by angle gama

Euler angles

- Most used:
  - y-x-z with y = up vector, intrinsic
  - yaw = beta, pitch = alpha, roll = gama
- Conversion to rotation matrix
- Extrinsic rotations, x-y-z order
  - R = R_x(alpha)R_y(beta)R_z(gama)
- Intrinsic rotations, x-y-z order
  - R = R_z(gama)R_y(beta)R_x(alpha)



Intrinsic, z-x-z
wikipedia.org

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

Quaternions

- Good for smooth and continuous rotation
- Defined as 4 floating point values |q0 q1 q2 q3|
- Q = q0 + **i**q1 + **j**q2 + **k**q3
- Basis elements: $\mathbf{i}^2 = \mathbf{j}^2 = \mathbf{k}^2 = \mathbf{ijk} = -1$
- From arbitrary axis and angle of rotation
- q0 = cos (angle / 2)
- q1 = axis.x * sin (angle / 2)
- q2 = axis.y * sin (angle / 2)
- q3 = axis.z * sin (angle / 2)

Quaternions

- $Q = q0 + \mathbf{i}q1 + \mathbf{j}q2 + \mathbf{k}q3$
- Conjugation
  - $Q^* = q0 - \mathbf{i}q1 - \mathbf{j}q2 - \mathbf{k}q3$
- Norm
  - $|Q| = \mathrm{sqrt}(QQ^*) = \mathrm{sqrt}(q0^2 + q1^2 + q2^2 + q3^2)$
- Unit quaternion – with norm 1
- Inverse, Reciprocal
- $Q^{-1} = Q^*/|Q|^2 \; ; \; QQ^{-1} = 1$

Quaternions

- $P = p0 + \mathbf{i}p1 + \mathbf{j}p2 + \mathbf{k}p3$
- $Q = q0 + \mathbf{i}q1 + \mathbf{j}q2 + \mathbf{k}q3$
- Addition
  - $P+Q = (p0+q0)+(p1+q1)\mathbf{i}+(p2+q2)\mathbf{j}+(p3+q3)\mathbf{k}$
- Dot product
  - $P.Q = p0q0 + p1q1 + p2q2 + p3q3$
- Multiplication – Hamilton product
  - $PQ = p0q0-p1q1-p2q2-p3q3$
    $+(p0q1+p1q0+p2q3-p3q2)\mathbf{i} +$
    $+(p0q2-p1q3+p2q0+p3q1)\mathbf{j} +$
    $+(p0q3+p1q2-p2q1+p3q0)\mathbf{k}$

Quaternions

- Rotating vertex V=(x,y,z) by quaternion Q
- (x'**i**+y'**j**+z'**k**) = Q(x**i**+y**j**+z**k**)Q*
- Composing two rotations – simple multiplication of two quaternion
- From unit quaternion Q to 4x4 rotation matrix R

$$R = \begin{pmatrix} q0^2+q1^2-q2^2-q3^2 & 2q1q2-2q0q3 & 2q1q3+2q0q2 & 0 \\ 2q1q2+2q0q3 & q0^2-q1^2+q2^2-q3^2 & 2q2q3-2q0q1 & 0 \\ 2q1q3-2q0q2 & 2q2q3+2q0q1 & q0^2-q1^2-q2^2+q3^2 & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix}$$

Quaternions

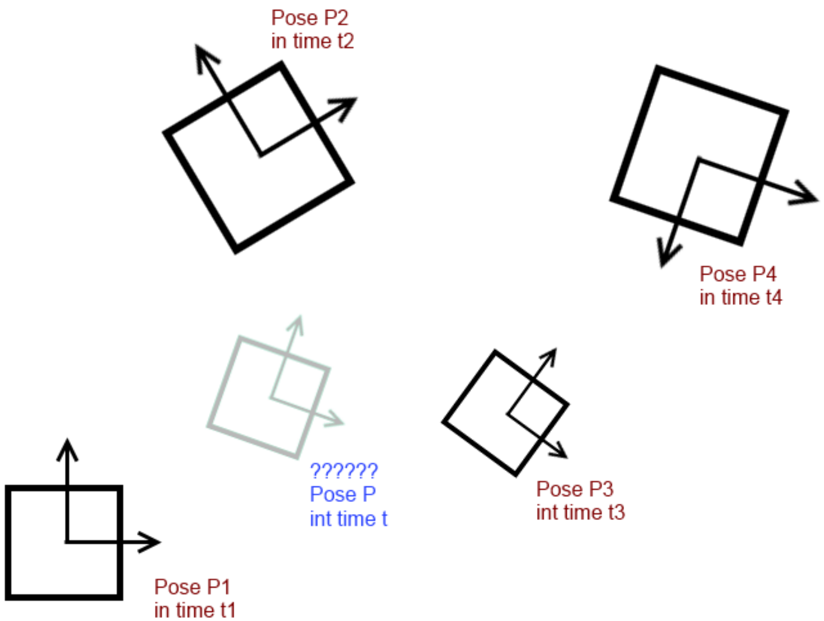
- Euler angles to quaternion
- Multiply 3 elementary quaternions in order based on predefined order anf intrinsic or extrinsic rotations
- Example:
  - x-y-z, extrinsic
  - Q_x = (cos(0.5*alpha), sin(0.5*alpha), 0, 0)
  - Q_y = (cos(0.5*alpha), 0, sin(0.5*alpha), 0)
  - Q_z = (cos(0.5*alpha), 0, 0, sin(0.5*alpha))
  - $Q = Q_x Q_y Q_z$

Creating final matrix

- Computing final Object Pose
- R – 4x4 rotation matrix
- S – 4x4 scale matrix
- T – 4x4 translation matrix
- M – Final OpenGL Matrix
- M = T.R.S
- Set M as model matrix before rendering given object

Pose Interpolation

- Given key poses P1, P2, ..., Pn in key times t1, t2, ..., tn
- Compute pose P in arbitrary time t



Pose interpolation

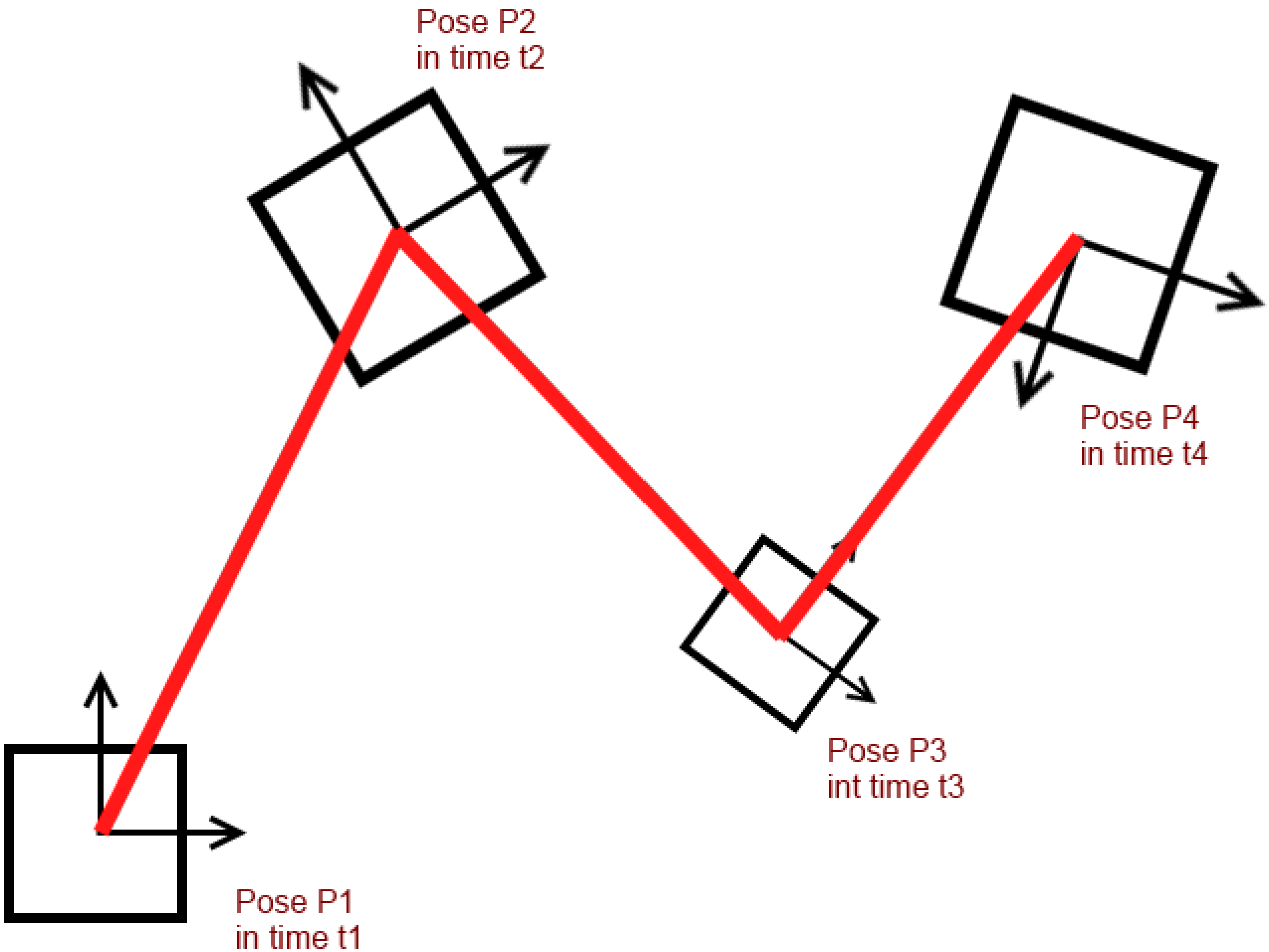
- Parameters from affine space
  - Position, scale, color, ...
  - Independent interpolation of each coordinate
  - Linear (lerps), cubic
- Parameter from spherical space
  - Rotation
  - Interpolation over sphere
  - Spherical linear (slerp), Bezier cubic

Linear affine interpolation

- Given knots $t_1, t_2, \ldots, t_n$
- Given float values $v_1, v_2, \ldots, v_n$
- Given interpolation value t
- Find span j such that t is in $<t_j, t_{j+1}>$
- If $t_j = t_{j+1}$, then $v = v_j$
- Resulting interpolated value

$$v = \frac{t_{j+1} - t}{t_{j+1} - t_j} v_j + \frac{t - t_j}{t_{j+1} - t_j} v_{j+1}$$

Linear affine interpolation



Linear sphere interpolation

- Because of quaternion representation, rotations can be treated as values over unit sphere in 4D
- We must interpolate values on sphere
- Can not interpolate values of rotation matrix independently
- Linear affine interpolation of Euler angles
- Interpolation of two quaternions Q1, Q2 with value t in <0,1>

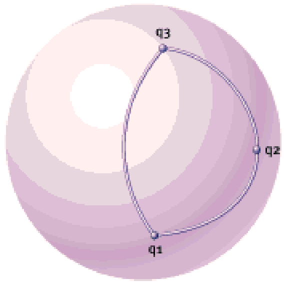
$$slerp(Q1, Q2, t) = \frac{\sin((1-t)\Omega)}{\sin\Omega} Q1 + \frac{\sin(t\Omega)}{\sin\Omega} Q2$$

$$\cos\Omega = Q1.Q2$$

Linear sphere interpolation

- Given knots $t_1, t_2, \ldots, t_n$
- Given quaternions $Q_1, Q_2, \ldots, Q_n$
- Given interpolation value t
- Find span j such that t is in $<t_j, t_{j+1}>$
- If $t_j = t_{j+1}$, then $Q = Q_j$
- Resulting interpolated quaternion

GD,9801,Fig3



$$Q = slerp(Q_j, Q_{j+1}, \frac{t - t_j}{t_{j+1} - t_j})$$

Cubic affine interpolation

- Linear interpolation is only C^0 – sudden changes in velocity vector in key values
- We want at least C^1 interpolation – we must compute tangent vectors in key points
- Instead of linear segments, we use segments given by cubic polynomials
- Each segment described in Hermite form

Cubic affine interpolation

- Given knots $t_1, t_2, ..., t_n$
- Given float values $v_1, v_2, ..., v_n$
- Given tangent values $m_1, m_2, ..., m_n$
- Given interpolation value t, find span (segment) j such that t is in $<t_j, t_{j+1}>$
- Hermite cubic segment value

$$s = \frac{t - t_j}{t_{j+1} - t_j}$$

$$v = H(s) = (2s^3 - 3s^2 + 1)v_j + (s^3 - 2s^2 + s)(t_{j+1} - t_j)m_j +$$
$$+ (-2s^3 + 3s^2)v_{j+1} + (s^3 - s^2)(t_{j+1} - t_j)m_{j+1}$$

Computing tangents

- Finite differences

$$m_k = \frac{v_{k+1} - v_k}{2(t_{k+1} - t_k)} + \frac{v_k - v_{k-1}}{2(t_k - t_{k-1})}$$

- Cardinal spline
  – Tension parameter c

$$m_k = (1-c)\frac{v_{k+1} - v_{k-1}}{t_{k+1} - t_{k-1}}$$

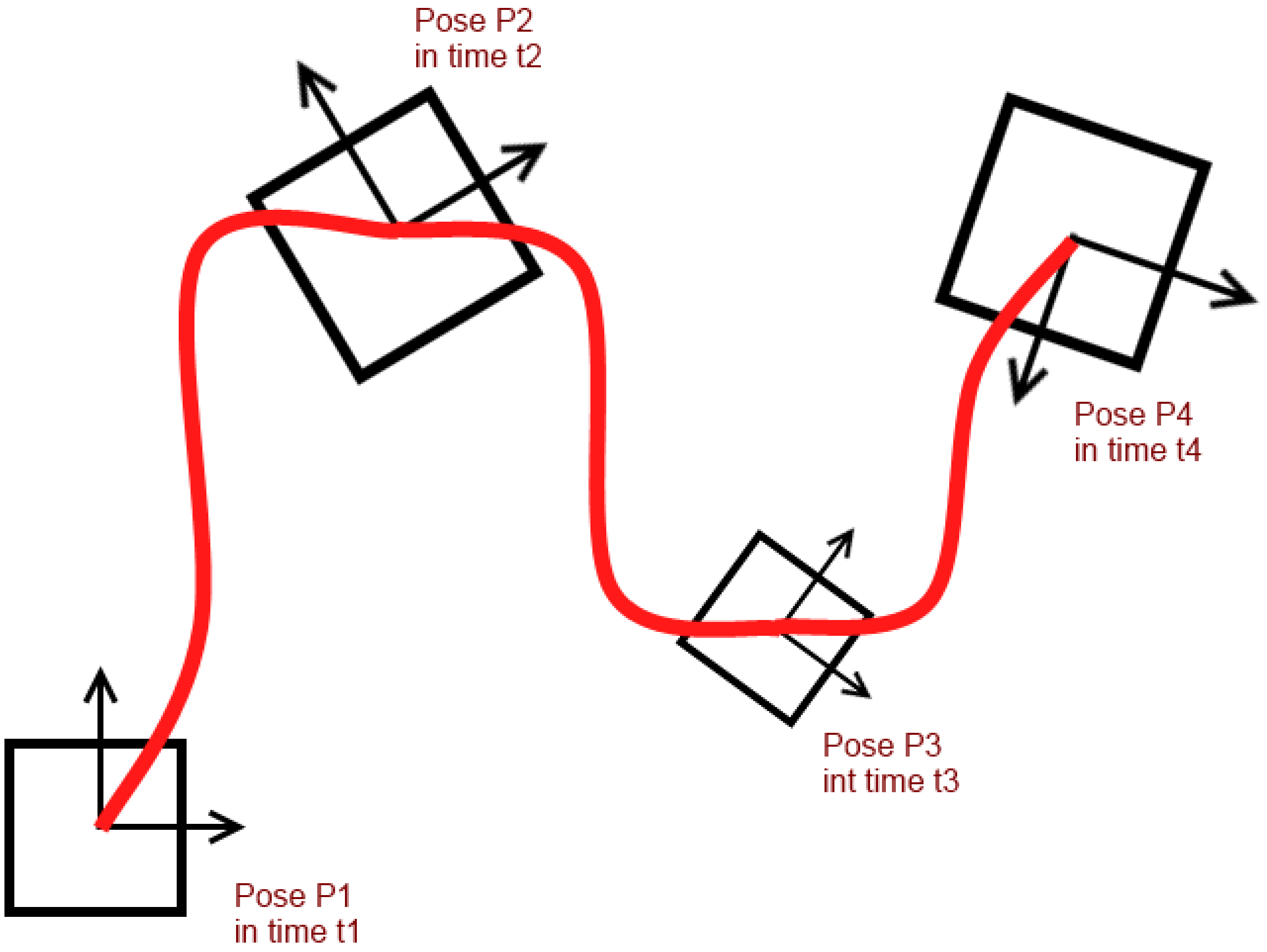
- Catmul-Rom spline

$$m_k = \frac{v_{k+1} - v_{k-1}}{t_{k+1} - t_{k-1}}$$

- Kochanek-Bartels spline
  – Tension, bias and continuity parameters

$$m_k = \frac{(1-t)(1+b)(1+c)}{2}\frac{v_{k+1} - v_k}{t_{k+1} - t_k} + \frac{(1-t)(1-b)(1+c)}{2}\frac{v_k - v_{k-1}}{t_k - t_{k-1}}$$

Cubic affine interpolation



Cubic sphere interpolation

- Given knots $t_1, t_2, \ldots, t_n$
- Given quaternions $Q_1, Q_2, \ldots, Q_n$
- Find span j such that t is in $<t_j, t_{j+1}>$
- Performing cubic Bezier interpolation on 4D sphere
- For segment between quaternions Q_j, Q_{j+1}, compute two other control points A_j, B_{j+1} so we can construct cubic Bezier curve using advanced de Casteljau algorithm on 4D sphere

Cubic sphere interpolation

- $L_j = DoubleArc(Q_{j-1}, Q_j) = 2(Q_{j-1} \cdot Q_j)Q_{j-1} - Q_j = Q_j Q_{j-1}^{-1} Q_j$
- $L_{j+1} = DoubleArc(Q_j, Q_{j+1}) = Q_{j+1} Q_j^{-1} Q_{j+1}$
- $A_j = BisectArc(L_j, Q_{j+1}) = (L_j + Q_{j+1}) / |L_j + Q_{j+1}|$
- $A_{j+1} = BisectArc(L_{j+1}, Q_{j+2}) = (L_{j+1} + Q_{j+2}) / |L_{j+1} + Q_{j+2}|$
- $B_{j+1} = DoubleArc(A_{j+1}, Q_{j+1})$
- DeCasteljau algorithm for value $u = (t - t_j)/(t_{j+1} - t_j)$
  - $P1 = slerp(Q_j, A_j, u)$
  - $P2 = slerp(A_j, B_{j+1}, u)$
  - $P3 = slerp(B_{j+1}, Q_{j+1}, u)$
  - $P4 = slerp(P1, P2, u)$, $P5 = slerp(P2, P3, u)$
  - $Q = slerp(P4, P5, u)$

Looping animation

- Defining closed animation curves
- Setting t_{n+1} as time of one loop animation
- Setting first and last key values same
- $v_{n+1}=v_1$
- Working with knots, tangents and values in cyclic way
- $v_0=v_n, v_{-1}=v_{n-1}, v_{n+1}=v_1, v_{n+2}=v_2, \ldots$

Questions?