

Affine Transformation

Affine/Linear Transformation

Most used deformations $p' = f(p)$

can be represented by a matrix $\rightarrow$ can be sent as uniform parameter to the shader.

- In standard 3D coordinates $p = (x, y, z)$

$$p' = L p + t$$

L : linear component (3×3 matrix)

t : translation

- In homogeneous coordinates $p = (x, y, z, 1)$

$$p' = A p$$

A : 4×4 matrix

$$A = \left(\begin{array}{ccc|c} L & t \\ \hline 0 & 1 \end{array} \right) = \left(\begin{array}{ccc|c} L_{xx} & L_{xy} & L_{xz} & t_x \\ L_{yx} & L_{yy} & L_{yz} & t_y \\ L_{zx} & L_{zy} & L_{zz} & t_z \\ \hline 0 & 0 & 0 & 1 \end{array} \right)$$

Properties:

- $\det(L)$: Change of volume when applying L to a shape

- $L^T L = 1 \Rightarrow$ Isometry

Affine/Linear Deformations

Translation

$$t(p) = (x + t_x, y + t_y, z + t_z)$$

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

Scaling

$$s(p) = (s_x x, s_y y, s_z z)$$

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

Rotation

Several possible representations
(see later)

Note: Isometry

Shearing

$$sh_{xy}(p) = (x + \lambda y, y, z)$$

$$sh_{xz}(p) = (x + \lambda z, y, z)$$

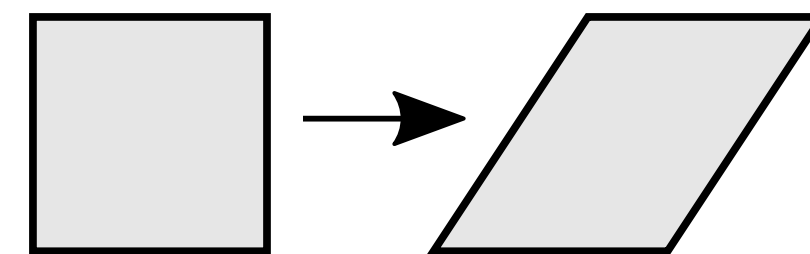
$$sh_{yx}(p) = (x, y + \lambda x, z)$$

...

$$Sh_{xy} = \begin{pmatrix} 1 & \lambda & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix} \quad Sh_{xz} = \begin{pmatrix} 1 & 0 & \lambda & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix} \quad Sh_{yx} = \begin{pmatrix} 1 & 0 & 0 & 0 \\ \lambda & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix} \dots$$

Usually avoided in Graphics

Note: $\det(Sh) = 1 \rightarrow$ constant volume (but no isometry)



Rotations

3D rotations have 3 dof, No unique representation

Matrix

$$R = \begin{pmatrix} R_{xx} & R_{xy} & R_{xz} \\ R_{yx} & R_{yy} & R_{yz} \\ R_{zx} & R_{zy} & R_{zz} \end{pmatrix}$$

$$R^T R = I$$

$$\det(R) = 1$$

(+) Computationally convenient

(-) Non-explicit dof, redundancies

Euler Angles

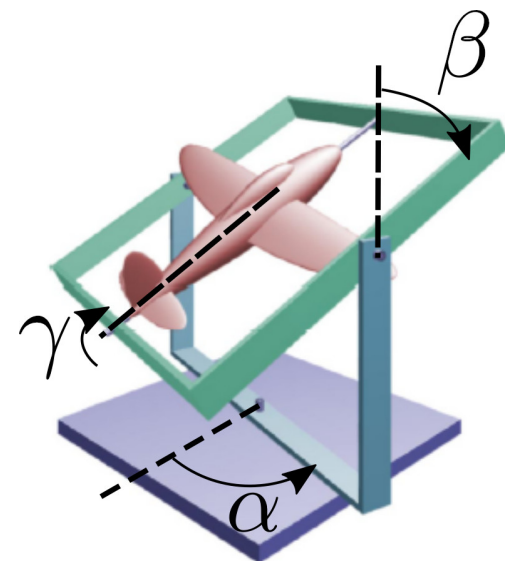
3 angles: (α, β, γ)

Composition of rotation around basic axes

Not unique (x-y-z, y-z-x, x-y-x', x-z-x', ...)

(+) Meaningfull parameters

(-) Gimbal-lock

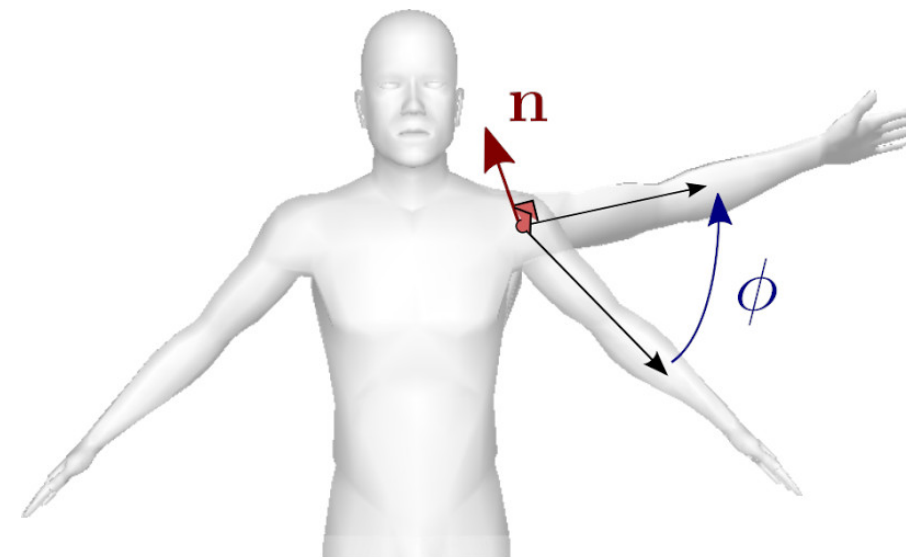


Axis angle

$(\mathbf{n}, \theta)$

(+) Meaningfull parameters

(-) No direct composition



Quaternion

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

(+) Composition and interpolation

(-) Less intuitive components

Cautions with transformations order

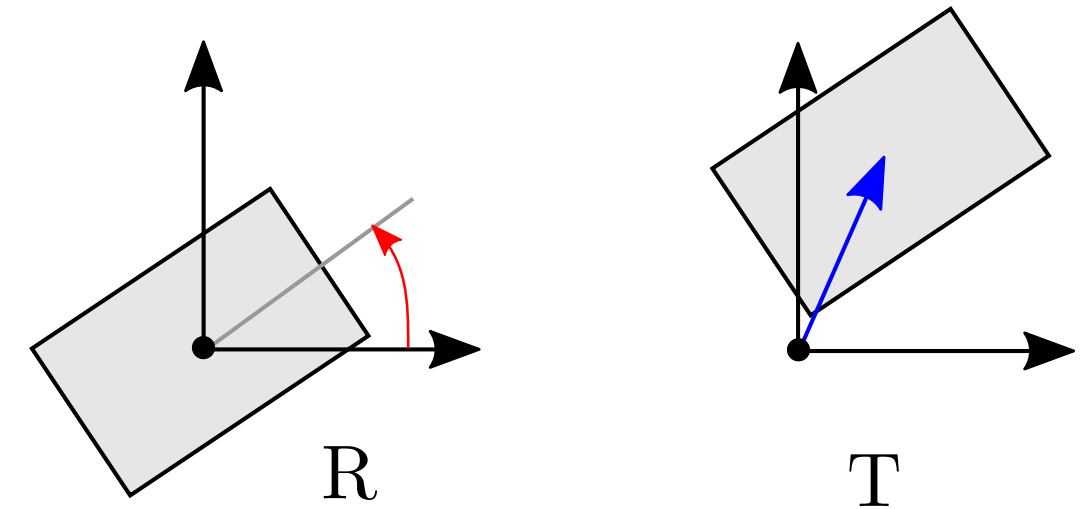
Take care, order of operation does matter !

Rotation r , Translation t : $r \circ t \neq t \circ r \Rightarrow M_1 = T R \neq R T = M_2$

Take care (2): transformation matrices applied to coordinates from right to left.

$$M_1 = TR = \left(\begin{array}{c|c} 1 & t \\ \hline 0 & 1 \end{array} \right) \left(\begin{array}{c|c} R & 0 \\ \hline 0 & 1 \end{array} \right) = \left(\begin{array}{c|c} R & t \\ \hline 0 & 1 \end{array} \right)$$

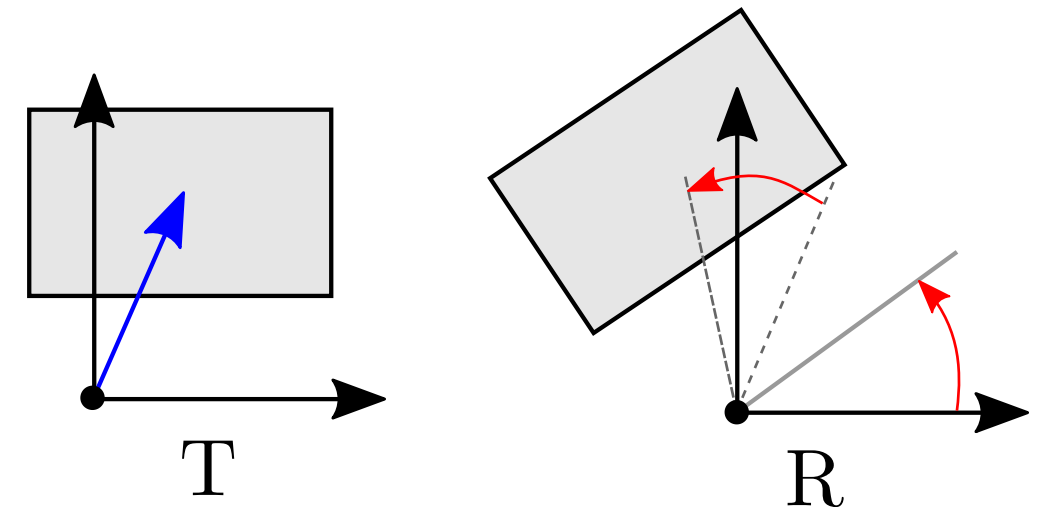
First rotates, then translates



$$M_2 = RT = \left(\begin{array}{c|c} R & 0 \\ \hline 0 & 1 \end{array} \right) \left(\begin{array}{c|c} 1 & t \\ \hline 0 & 1 \end{array} \right) = \left(\begin{array}{c|c} R & R t \\ \hline 0 & 1 \end{array} \right)$$

First translates, then rotates

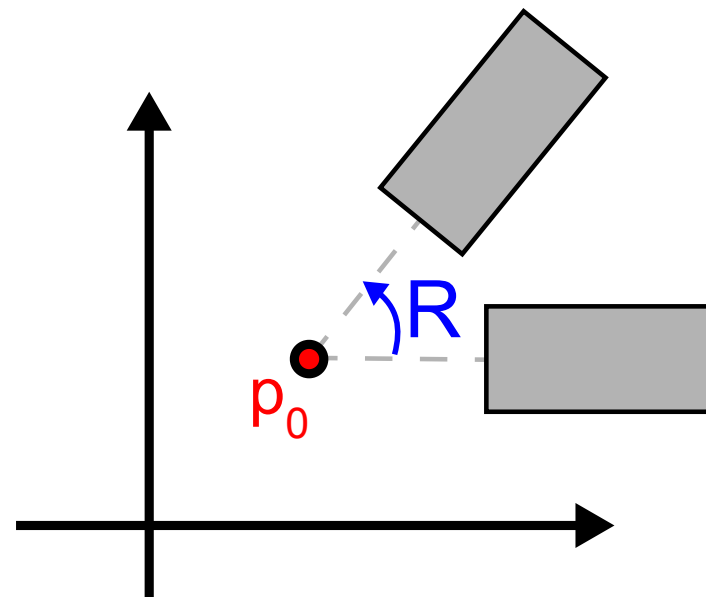
Rotation always happens around the origin.



Affine Transformation Exercises

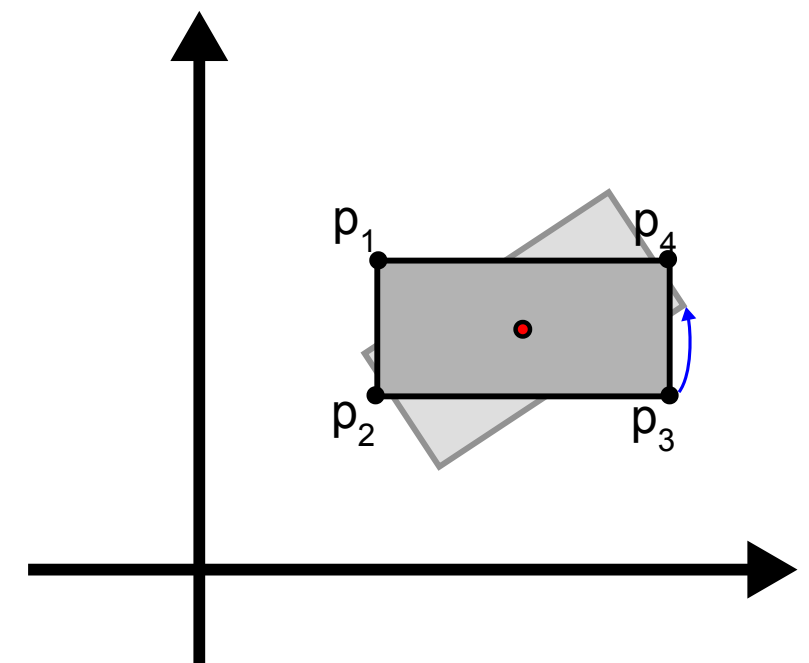
Q. Express the affine transform (as a 4×4 block matrix) corresponding to a rotation R applied around an arbitrary position p_0 in space.

> $M = \dots$



Q. Consider a shape defined by a triangular mesh with vertex positions $(p_i)_{i \in [1, N]}$.

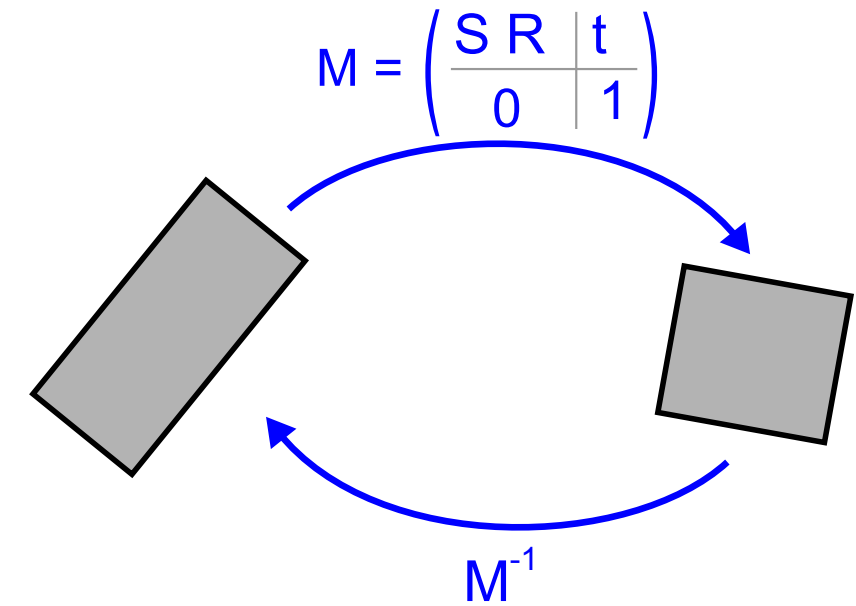
> Express the affine transform allowing to rotate the shape around its barycenter.



Affine Transformation Exercises

Q. Consider the affine transform M parameterized by a scaling s , a rotation R , and a translation t .

> Express the inverse matrix M^{-1} with respect to s , R and t



Interpolating rotation

Do not use componentwise-interpolation on rotation matrix
 $\Rightarrow$ interpolate in **quaternion** space

Can use either:

- **SLERP** - Spherical Linear Interpolation

$$q(t) = \frac{\sin((1-t)\Omega)}{\sin(\Omega)} q_1 + \frac{\sin(t\Omega)}{\sin(\Omega)} q_2, \quad \text{with } \cos(\Omega) = q_1 \cdot q_2$$

Between two unit quaternions q_1, q_2

- **LERP** - Linear Interpolation

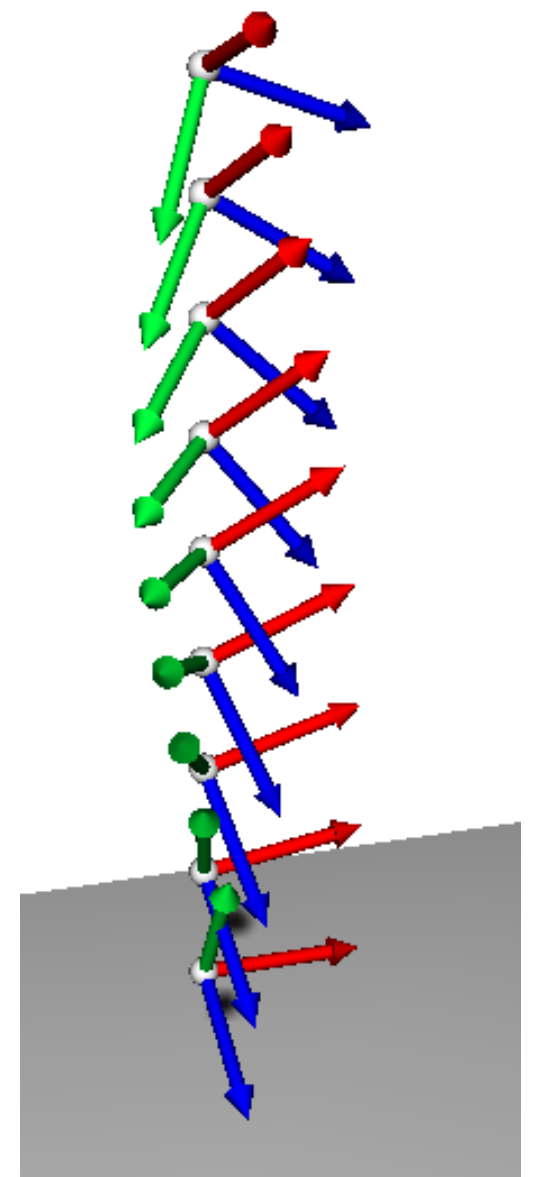
$$q(t) = \frac{\sum_j \alpha_j q_j}{\|\sum_j \alpha_j q_j\|}$$

When blending multiple quaternions q_j with weights α_j

Rem.

When interpolating b/w rotations and positions:

Use quaternion with rotation, componentwise-interpolation on position



Handling affine transformation : Polar Decomposition

Key-pose transforms are often given as 4×4 matrices.

How can we interpolate b/w affine transforms?

- Splitting linear M /translation part is easy.
- *Problem*: interpolating the linear part M mixes rotation, scaling, shearing

⇒ Split M into: rotation part, and scaling/shearing.

- Interpolate rotation with quaternion (ex. SLERP/LERP)
- Scaling/shearing using componentwise interpolation (ex. linear).

- Polar decomposition: $M = R D$

- R : Rotation matrix
- D : Positive semi-definite matrix

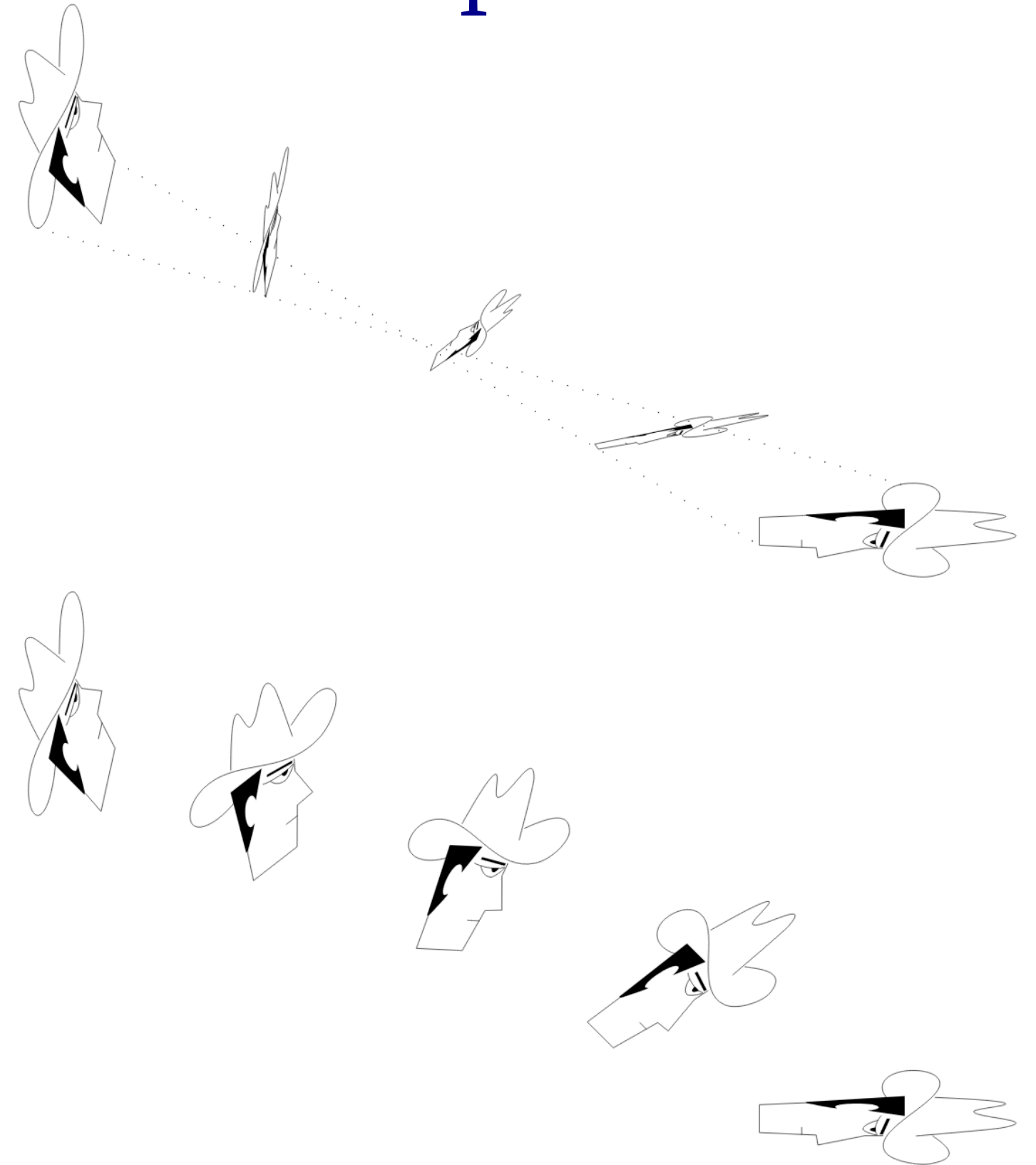
[K. Shoemake and T. Duff, *Matrix Animation and Polar Decomposition*. Graphics Interface 92.]

- Polar decomposition is obtained from SVD

$$\text{SVD}(M) = W \Sigma V^T \text{ with } R = W V^T, \quad D = V \Sigma V^T$$

- Or numerically, R can be computed using the following iterative scheme

$$R_0 = M, \quad R_{i+1} = 0.5 (R_i + (R_i^{-1})^T)$$



Handling affine transformation

Algorithm to interpolate between two general 4×4 matrices M_1, M_2

1. Extract translation p_1, p_2 from M_1, M_2 .
2. Compute R_1, R_2 (3×3 rotation matrices), and D_1, D_2 (3×3 scaling/shearing matrices) from M_1, M_2
3. Interpolate linearly position and scaling/shearing $p(t) = (1 - t) p_1 + t p_2, D(t) = (1 - t) D_1 + t D_2$
4. Compute quaternions q_1, q_2 from R_1, R_2

$$\text{Note } M \rightarrow q \text{ with } q = \left(\frac{M_{zy} - M_{yz}}{2r}, \frac{M_{xz} - M_{zx}}{2r}, \frac{M_{yx} - M_{xy}}{2r}, \frac{r}{2} \right), r = \sqrt{1 + M_{xx} + M_{yy} + M_{zz}}$$

5. Compute $q(t) = \text{SLERP}(q_1, q_2, t)$
6. Convert back to matrix $q(t) \rightarrow R(t)$
7. Compute final matrix $M(t) = R(t) D(t)$ with translation $p(t)$.

Camera and OpenGL

In OpenGL: no real notion of "camera"

Unique viewpoint - cube $(x, y, z) \in [-1, 1]^3$ - Normalized Device Coordinate

Left-right/x, Bottom-up/y, Front-back/z (toward negative z)

No perspective (displays (x,y) for the smallest z)

Any camera effect (position, orientation, perspective)
must be coded "manually" in the shader.

All these effects can be represented as matrices.

Three common matrices

Model matrix

Affine transform - specific to a shape.

Local object coordinates to world space.

View matrix

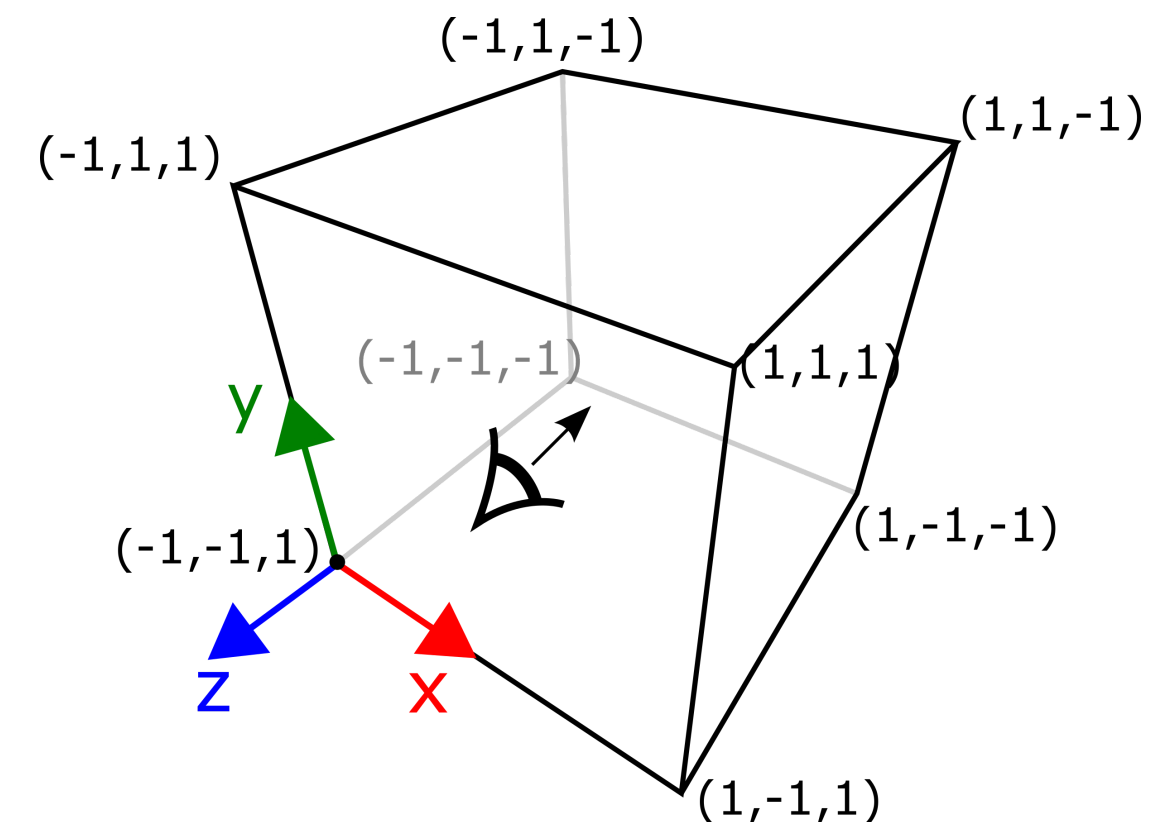
Rigid transform - position/orientation of the camera in the scene.

World space to camera/view/eye space

Projection matrix

Perspective matrix - map frustum to normalized cube. (/or ortho projection)

Camera space to Normalized Device Coordinate (clip/screen space)



$$p = \text{Projection} \times \underbrace{\text{View} \times \text{Model}}_{\text{ModelView}} \times p_0$$

View representation as matrix

Camera parameterized by:

A center C ; right, up, front unit vectors
expressed in global coordinates

Frame matrix:

Transform unit coordinates into camera vectors

$$\text{Frame} \times (1, 0, 0, 0) = \text{right}$$

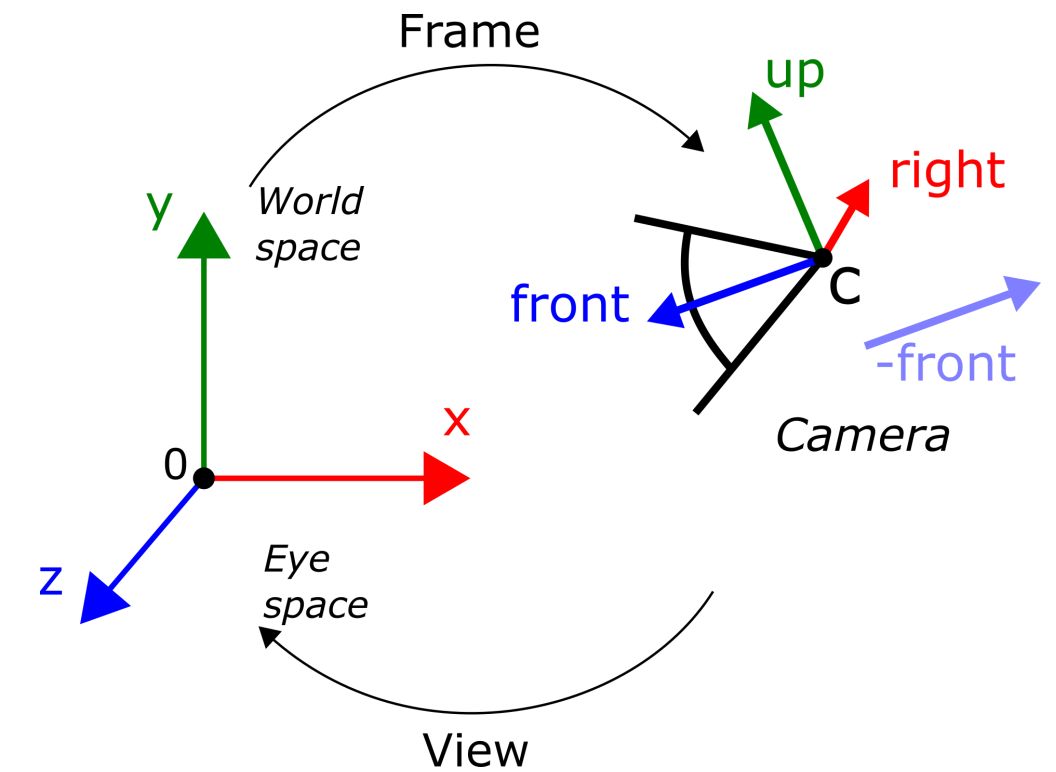
$$\text{Frame} \times (0, 1, 0, 0) = \text{up}$$

$$\text{Frame} \times (0, 0, 1, 0) = -\text{front}$$

$$\text{Frame} \times (0, 0, 0, 1) = c$$

Matrix of column-oriented vectors of camera directions

$$\text{Frame} = \begin{pmatrix} | & | & | & | \\ \text{right} & \text{up} & -\text{front} & c \\ | & | & | & | \\ 0 & 0 & 0 & 1 \end{pmatrix} = \begin{pmatrix} O & c \\ 0 & 1 \end{pmatrix}$$



View matrix: inverse of the Frame matrix

Transforms the camera vectors into the unit coordinates

$$\text{View} \times \text{right} = (1, 0, 0, 0)$$

$$\text{View} \times \text{up} = (0, 1, 0, 0)$$

$$\text{View} \times -\text{front} = (0, 0, 1, 0)$$

$$\text{View} \times c = (0, 0, 0, 1)$$

Orientation O can be read on the rows

$$\text{View} = \begin{pmatrix} (\dots & \text{right} & \dots) & | \\ (\dots & \text{up} & \dots) & | \\ (\dots & -\text{front} & \dots) & | \\ 0 & 0 & 0 & 1 \end{pmatrix} = \begin{pmatrix} O & -O^T c \\ 0 & 1 \end{pmatrix}$$

Camera in shader - Question

Q. In a vertex shader the View matrix is passed as uniform parameter
How can you compute the position of the camera ?

A usefull function to set a point of view is the standard *look_at(eye, center, up)*

- eye: position of the viewer
- center: position that the viewer is looking at
- up: a direction representing the "vertical direction" of the viewer.

Q. How to compute the *View* matrix from the *look_at* parameters ?

Interacting with camera

Camera orientation has 3 degrees of freedom (+ 3 dof in position)

How to handle a camera orientation using a mouse (2-dof) ?

Assuming a freely oriented camera

Common approaches

"Spherical camera"/coordinates (θ, ϕ)

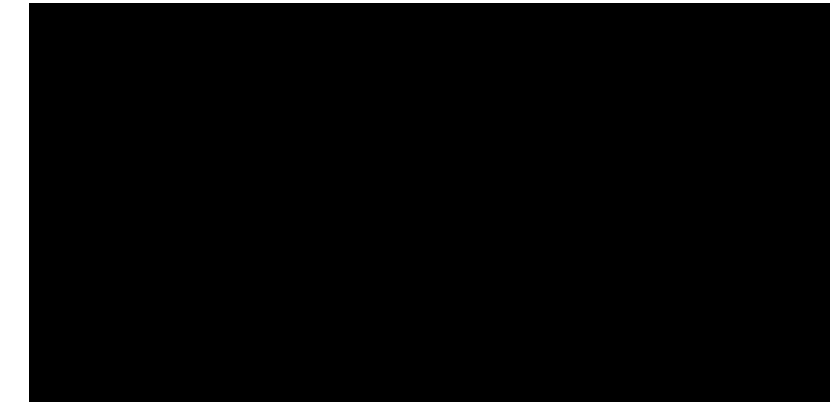
- (+) Good to easily "turn around" oriented structure
- (-) Lacks a dof: No "twist": position on the unit sphere force the *angle*
Not adapted to turn around non-oriented/badly-oriented objects.

Spherical coordinates + twist (ex. roll, pitch, yaw)

- (+) Precise and complete control
- (-) Require an additional key/controller to the mouse to control the 3 dof

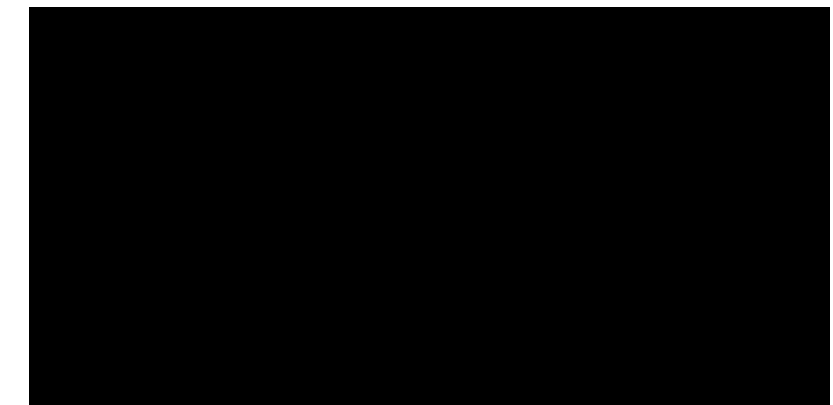
Example of "Spherical camera"

Correct orientation



Bad orientation of the shape

spherical camera is "stuck" in this orientation



Interacting with camera - ArcBall/Trackball metaphore

Other approach: **ArcBall/Trackball behavior**

3 dof: left/right, top/down, but also twist

Idea: Use the 2D cursor position as "3D point on the trackball"

Motion between 2 cursor positions = 3D Rotation applied to the ball

[K. Shoemake. A User Interface for Specifying Three-Dimensional Orientation Using a Mouse. Graphics Interface, 1992.]



Algorithm:

Inputs $p_1 = (p_{1x}, p_{1y})$, $p_2 = (p_{2x}, p_{2y})$ in screen coord.

$q_{1/2} = \text{ArcBallProjection}(p_{1/2})$

R = rotation between vectors (q_1, q_2)

$\text{ArcBallProjection}(p)$

$d = \text{norm}(p)$

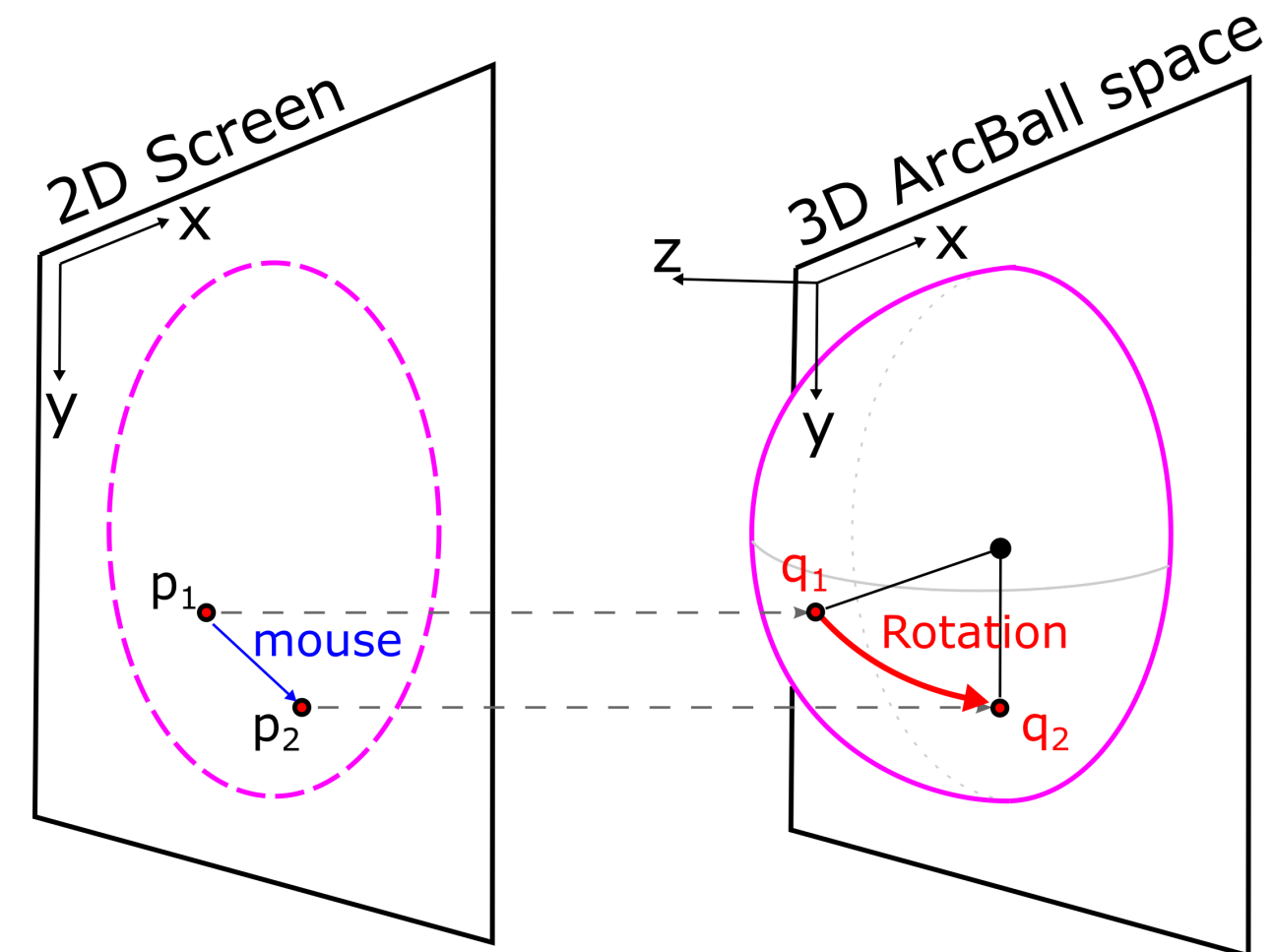
if ($d < 1/\sqrt{2}$)

$q = (p, \sqrt{1 - d^2})$

else // hyperbola

$q = (p, 1/(2d))$

return q



Note: the hyperbola smoothly extends the sphere in the entire screen.

Interacting with camera - ArcBall/Trackball metaphore

Natural way to rotate around a shape

(+) No privileged axis

No reference configuration, works in incrementing rotations

Behavior is orientation invariant

(-) Less precise than individual dof control

Drift of twist when moving back and forth

