

Real-time Rendering 4th Study

Chapter 4: Transformations

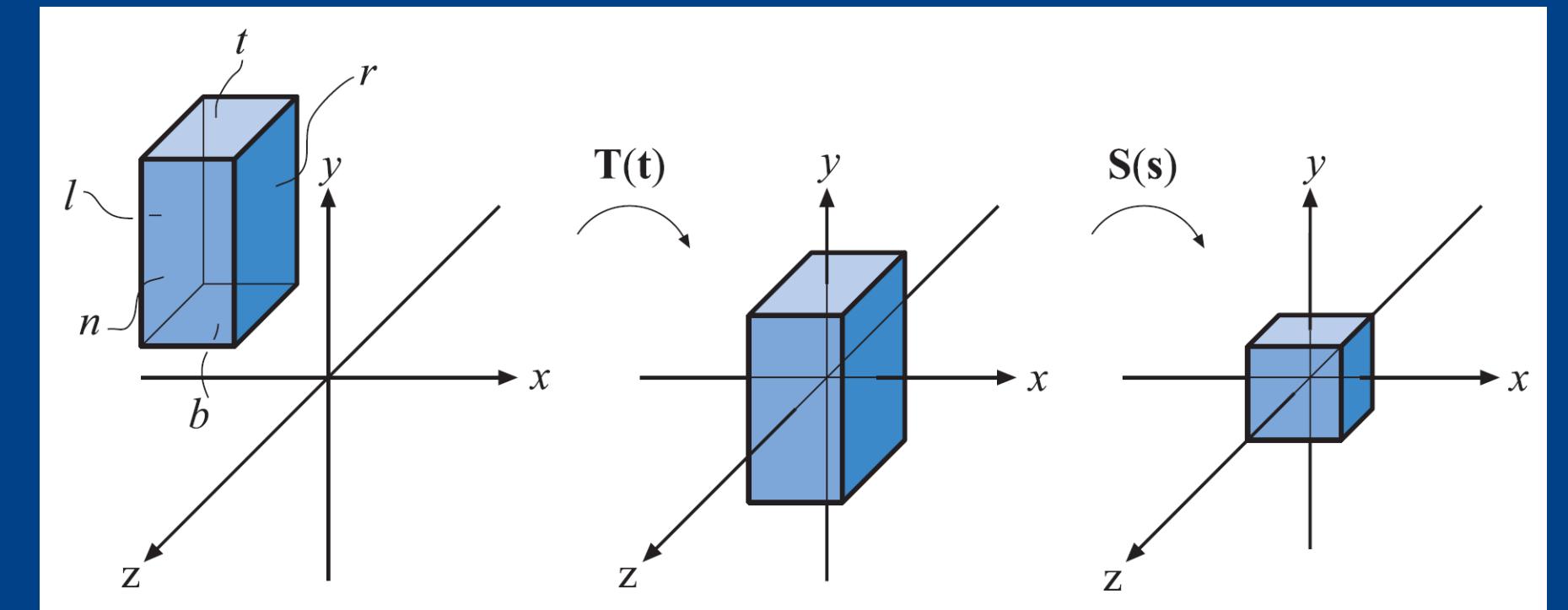
옥찬호

utilForever@gmail.com

- 변환의 큰 그림
- 기본 변환 5종 + 결합
- 제대로 된 법선 변환
- 오일러 각도와 분해
- 사원수 (Quaternion)
- 애니메이션 변환
- 투영 (Projection)

- 변환이 왜 중요한가
- 점 / 벡터 구분
- 선형 변환 vs 아핀 변환
- 동차 좌표

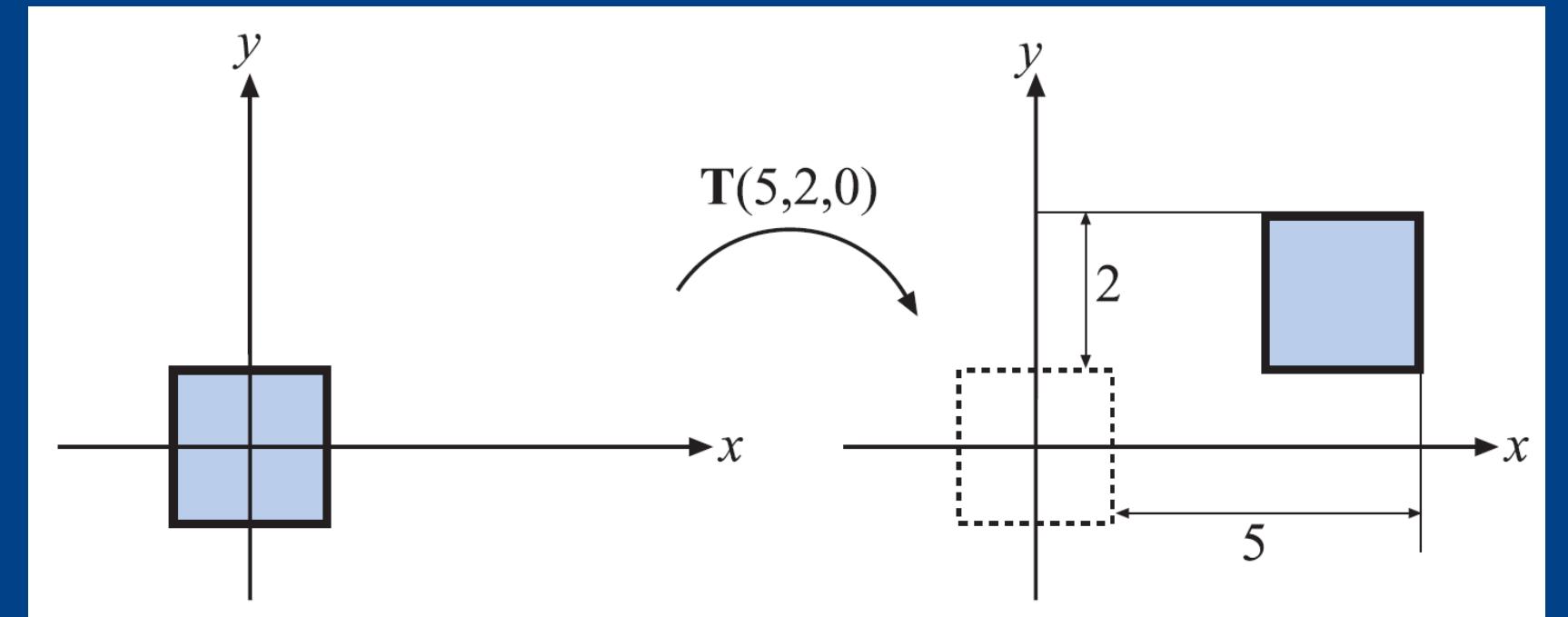
- 변환은 “같은 장면을 다른 관점/좌표계로 표현”하는 도구
 - 변환(Transform)이 하는 일
 - 오브젝트 / 빛 / 카메라의 위치와 방향을 정한다
 - 모양을 바꾸거나(스케일/전단) 애니메이션을 만든다
 - 여러 좌표계를 하나로 “정렬”해서 계산을 단순화한다
 - 3D를 2D 화면에 투영(Projection)해 렌더링한다
 - 그래픽스 API(GL / DirectX / Vulkan) 대부분이 4×4 행렬 기반
 - GPU 정점 셰이더는 정점을 행렬로 변환하는 것이 기본 작업



- 점(Point)과 벡터(Vector)는 다르다 → 동차 좌표가 해결

- 왜 구분해야 할까?

- 점 : “위치” (어디에 있나?) → 이동(Translation)의 영향을 받음
- 벡터 : “방향/변위” (어느 방향 / 얼마나?) → 이동의 영향을 받지 않음
- 회전 / 스케일 / 전단은 점과 벡터 모두에 영향을 준다
- 동차 좌표(Homogeneous Coordinates)
 - 4번째 성분 w 로 “점/벡터”를 한 시스템에서 다룬다
 - 점은 $w = 1$, 벡터는 $w = 0$ 으로 준다 (이동 성분이 벡터에 적용되지 않음)
 - $p = (x, y, z, 1)^T$, $v = (x, y, z, 0)^T$



- 선형 변환(Linear) vs 아핀 변환(Affine)
- 선형 변환의 조건
 - 벡터 덧셈 / 스칼라 곱을 보존
 - 원점을 항상 원점으로 보낸다 (이동 불가)
 - $f(\mathbf{a} + \mathbf{b}) = f(\mathbf{a}) + f(\mathbf{b}), f(c\mathbf{a}) = cf(\mathbf{a})$
- 아핀 변환 = 선형 + 이동
 - 실무에서 모델 / 월드 / 뷰 변환은 대부분 아핀 변환
 - 4×4 행렬 하나로 회전 / 스케일 / 전단 / 이동을 모두 표현 가능
 - 평행성은 유지되지만, 길이 / 각도는 항상 유지되지 않을 수 있음
- $\begin{bmatrix} \mathbf{L} & \mathbf{t} \\ \mathbf{0}^T & 1 \end{bmatrix} \mathbf{p}$

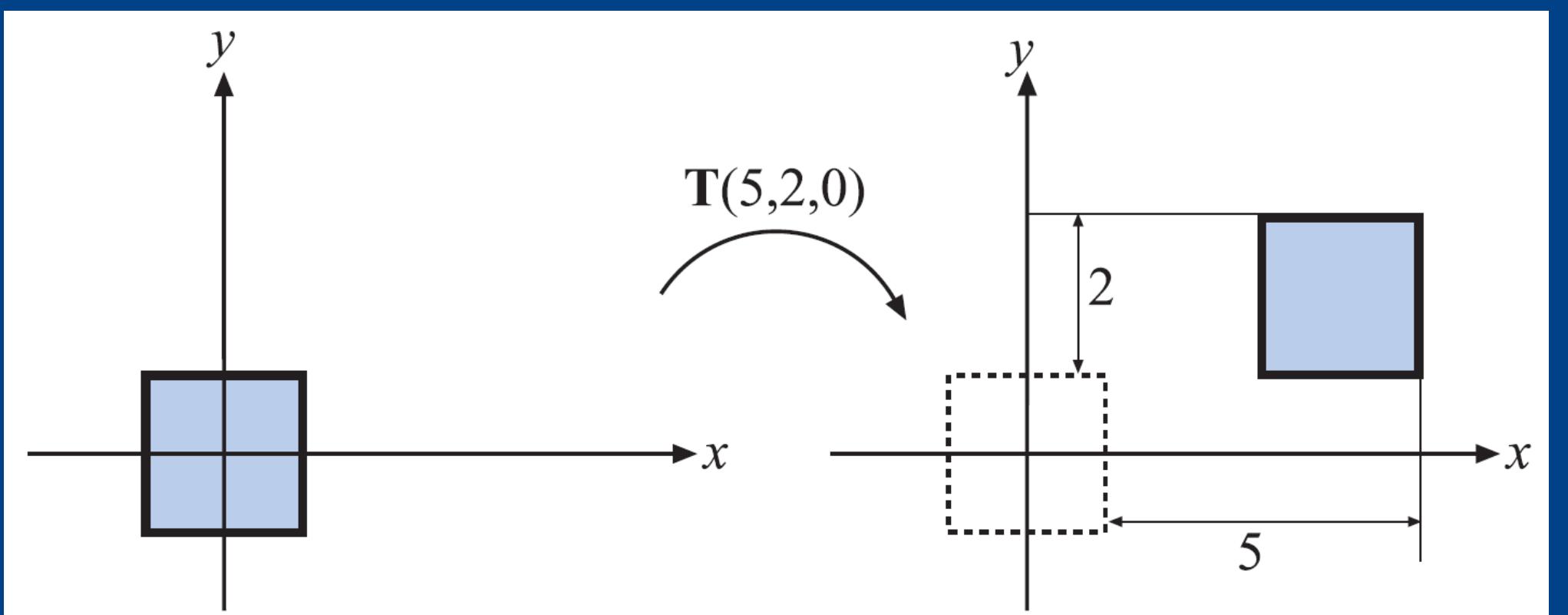
- 이동
- 회전
- 크기
- 전단
- 결합 순서

- 이동(Translation) : 모든 점을 같은 방향으로 평행 이동

- 이동 행렬 $\mathbf{T}(t)$

- 벡터 $\mathbf{t} = (t_x, t_y, t_z)$ 만큼 위치를 옮김
- 방향 벡터($w = 0$)는 영향을 받지 않음

$$\mathbf{T}(\mathbf{t}) = \mathbf{T}(t_x, t_y, t_z) = \begin{pmatrix} 1 & 0 & 0 & t_x \\ 0 & 1 & 0 & t_y \\ 0 & 0 & 1 & t_z \\ 0 & 0 & 0 & 1 \end{pmatrix}$$



- 역행렬 (되돌리기)

- $\mathbf{T}^{-1}(\mathbf{t}) = \mathbf{T}(-\mathbf{t})$ (반대 방향으로 이동)

- 회전(Rotation) : 원점을 지나는 축 주위로 각도만큼 회전

- 축 회전 행렬 (라디안 단위)

- $R_x(\theta), R_y(\theta), R_z(\theta)$ 는 가장 자주 쓰는 기본 회전

$$R_x(\phi) = \begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & \cos \phi & -\sin \phi & 0 \\ 0 & \sin \phi & \cos \phi & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix}$$

$$R_y(\phi) = \begin{pmatrix} \cos \phi & 0 & \sin \phi & 0 \\ 0 & 1 & 0 & 0 \\ -\sin \phi & 0 & \cos \phi & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix}$$

$$R_z(\phi) = \begin{pmatrix} \cos \phi & -\sin \phi & 0 & 0 \\ \sin \phi & \cos \phi & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix}$$

$$\begin{aligned} \mathbf{u} &= \begin{pmatrix} r \cos(\theta + \phi) \\ r \sin(\theta + \phi) \end{pmatrix} = \begin{pmatrix} r(\cos \theta \cos \phi - \sin \theta \sin \phi) \\ r(\sin \theta \cos \phi + \cos \theta \sin \phi) \end{pmatrix} \\ &= \underbrace{\begin{pmatrix} \cos \phi & -\sin \phi \\ \sin \phi & \cos \phi \end{pmatrix}}_{\mathbf{R}(\phi)} \underbrace{\begin{pmatrix} r \cos \theta \\ r \sin \theta \end{pmatrix}}_{\mathbf{v}} = \mathbf{R}(\phi)\mathbf{v}, \end{aligned}$$

- 회전은 길이 / 각도를 보존 \rightarrow 직교 행렬(Orthogonal)
- 따라서 $\mathbf{R}^{-1} = \mathbf{R}^T$

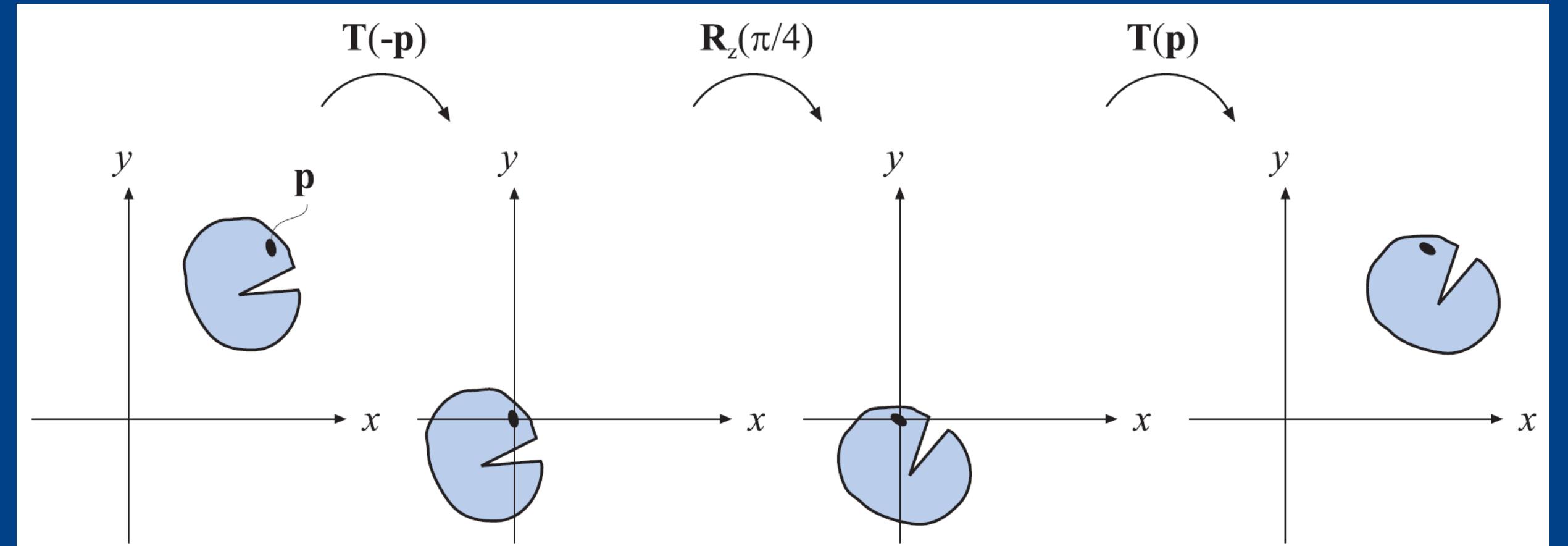
- 한 점 p 를 기준으로 회전 : “원점으로 옮기고 회전 후 되돌리기”

- 아이디어 (3단계)

- $T(-p)$: 회전 중심 p 를 원점으로 이동
- $R(\theta)$: 원점 기준 회전 수행
- $T(p)$: 원래 위치로 되돌림

- 결과 행렬

- $M = T(p)R(\theta)T(-p)$
- 이 패턴은 카메라 공전(Orbit) / 피벗 회전 등에서 매우 자주 등장



- 크기(Scaling) : 축 방향으로 늘리거나 줄이기

- 스케일 행렬 $\mathbf{S}(s_x, s_y, s_z)$

- (s_x, s_y, s_z) 배로 확대 / 축소
- $s_x = s_y = s_z$ 면 균등(Uniform) 스케일

$$\mathbf{S}(\mathbf{s}) = \begin{pmatrix} s_x & 0 & 0 & 0 \\ 0 & s_y & 0 & 0 \\ 0 & 0 & s_z & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix}$$

- 반사(Reflection)와 주의점

- 축 스케일에 음수가 포함되면 거울(반사) 변환이 된다
- $\det(\text{상단 } 3 \times 3) < 0$ 이면 “손잡이(Handedness)”가 뒤집힘
- 삼각형 Winding이 바뀌어 Backface Culling / 조명이 깨질 수 있음

- 전단(Shearing) : 한 축이 다른 축에 비례해 밀려나는 변형

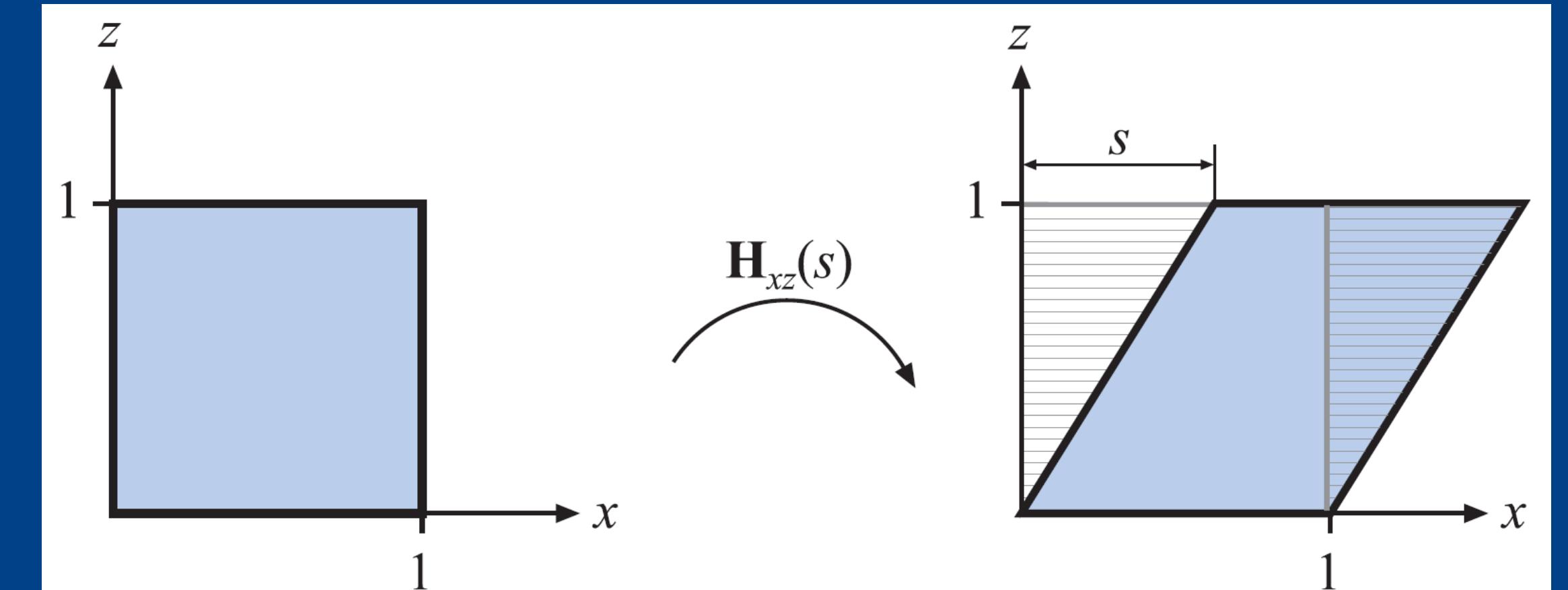
- 전단의 직관

- 사각형을 “기울여” 평행사변형으로 만드는 변형
- 특수 효과(왜곡, 지터링), 또는 행렬 분해에서 전단 성분을 분리할 때 등장

- 예 : x 가 z 에 비례해 변함 ($\mathbf{H}_{xz}(s)$)

$$\mathbf{H}_{xz}(s) = \begin{pmatrix} 1 & 0 & s & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix}$$

- 역행렬은 s 의 부호를 바꾸면 됨
- $\det(\mathbf{H}) = 1 \rightarrow$ 부피(Volume) 보존



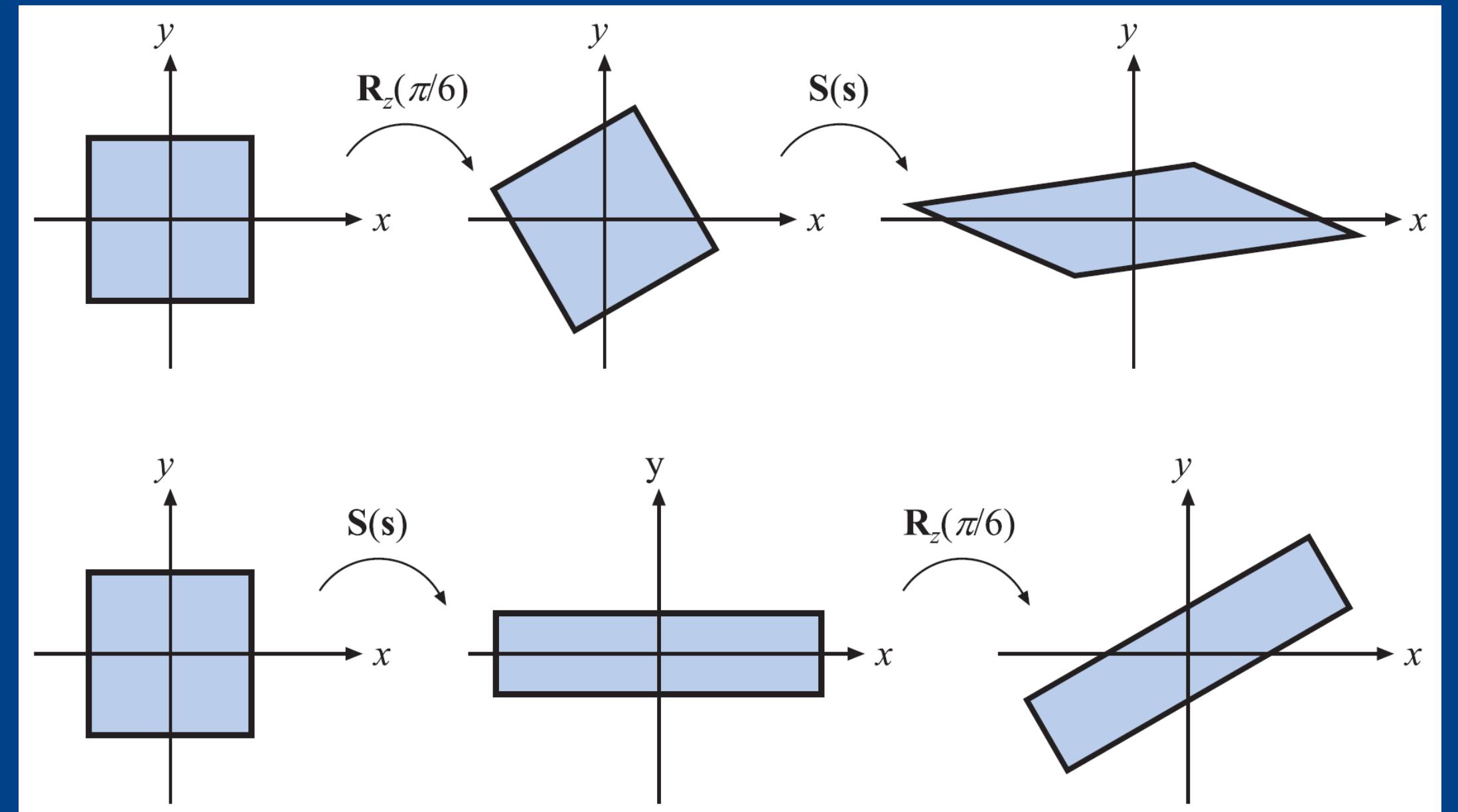
- **변환 결합(Concatenation) : 순서와 결과를 바꾼다**

- 행렬 곱은 교환법칙이 성립하지 않는다

- 일반적으로 $AB \neq BA$
- 따라서 “먼저 무엇을 적용할지”가 매우 중요
- 오른쪽 행렬이 먼저 적용됨 (열 벡터 기준)
- $\mathbf{p}' = (\mathbf{TRS})\mathbf{p}$
 - 먼저 \mathbf{S}
 - 그 다음 \mathbf{R}
 - 마지막 \mathbf{T}

- 왜 결합하나?

- 정점이 수천 개면 $\mathbf{S} \rightarrow \mathbf{R} \rightarrow \mathbf{T}$ 를 매번 곱하는 것보다 한 번 $\mathbf{M} = \mathbf{TRS}$ 를 만든 뒤 모든 정점에 적용하는 게 훨씬 빠름



- 강체(Rigid-body) 변환 : 회전 + 이동 (모양은 그대로)

- 특징

- 길이와 각도가 보존됨 (변형 없음)
- 카메라 / 오브젝트의 "배치(Pose)"에 핵심

$$\mathbf{X} = \mathbf{T}(t)\mathbf{R} = \begin{pmatrix} r_{00} & r_{01} & r_{02} & t_x \\ r_{10} & r_{11} & r_{12} & t_y \\ r_{20} & r_{21} & r_{22} & t_z \\ 0 & 0 & 0 & 1 \end{pmatrix}$$

- 법선(Normal) 변화 : 정점과 같은 행렬을 쓰면 틀릴 수 있다

- 문제 : 비균등 스케일 / 전단이 있으면...

- 표면은 변형되는데, 법선도 “같이” 변형되면 수직성이 깨질 수 있음

- 조명은 법선에 민감 → 바로 화면 오류로 나타남

- 해결 방법 : Inverse-Transpose

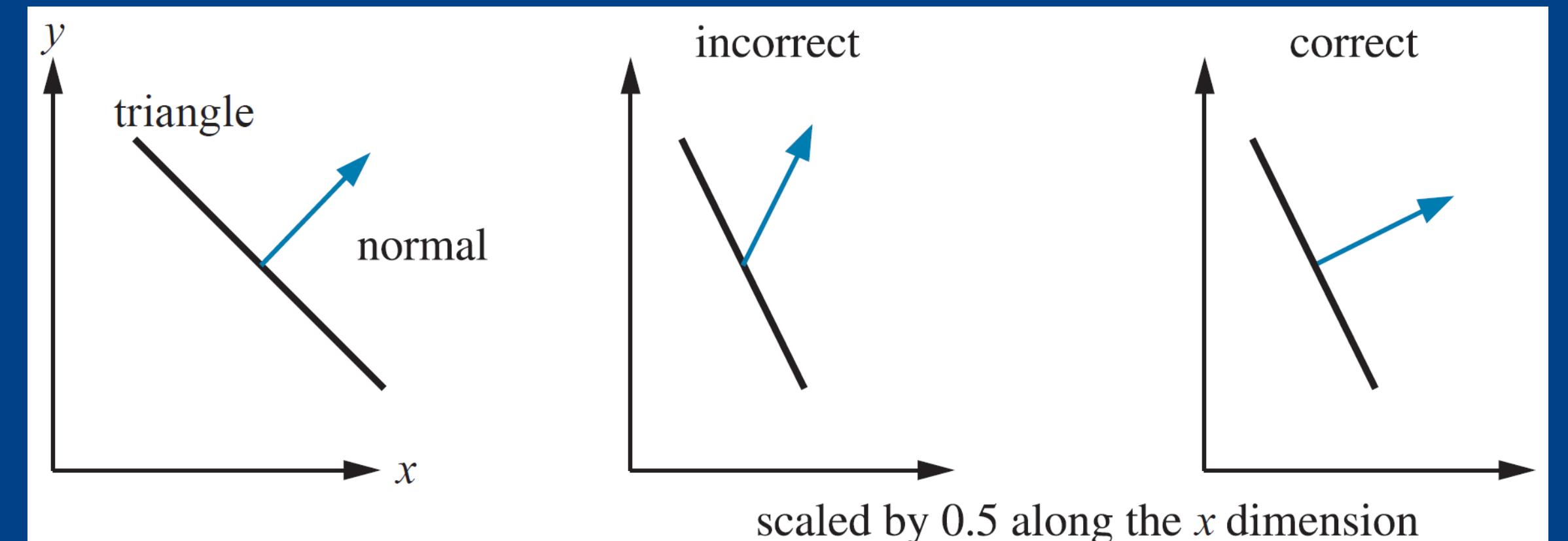
- 보통 3×3 상단만 필요 : 이동은 법선에 영향 없음

- 변환 후에는 정규화(Normalize)하는 게 일반적

- 예외 (간단해지는 경우)

- 이동 + 회전만 있는 경우 : **M**의 3×3 을 그대로 써도 OK (길이 보존)

- 균등 스케일이 추가되는 경우 : 길이만 변하니 스케일 값으로 재정규화 가능



- 역변환(Inverse) 계산 3가지 전략
 - 매개변수를 뒤집기 + 순서 반전
 - $\mathbf{T}(\mathbf{t})^{-1} = \mathbf{T}(-\mathbf{t})$, $\mathbf{R}(\theta)^{-1} = \mathbf{R}(-\theta)$, $\mathbf{S}(s)^{-1} = \mathbf{S}(1/s)$
 - $\mathbf{M} = \mathbf{TRS}$ 라면 $\mathbf{M}^{-1} = \mathbf{S}^{-1}\mathbf{R}^{-1}\mathbf{T}^{-1}$
 - 직교(Orthogonal)라면 전치
 - 회전 행렬 / 정규 직교 기저로 구성된 행렬에서 매우 빠름
 - 수치적으로도 안정된 편
 - 일반 역행렬 (가우스 소거법, LU, 수반 행렬 등)
 - 특별한 정보가 없을 때
 - 단, $\det = 0$ (특이 행렬)이면 역행렬이 존재하지 않음

- Yaw / Pitch / Roll
- 짐벌락
- 행렬 분해
- 임의 축 회전

오일러 & 특수 연산

Real-time Rendering 4th Study
Chapter 4: Transformations

- 오일러(Euler) 변환 : 직관적이지만 순서에 민감

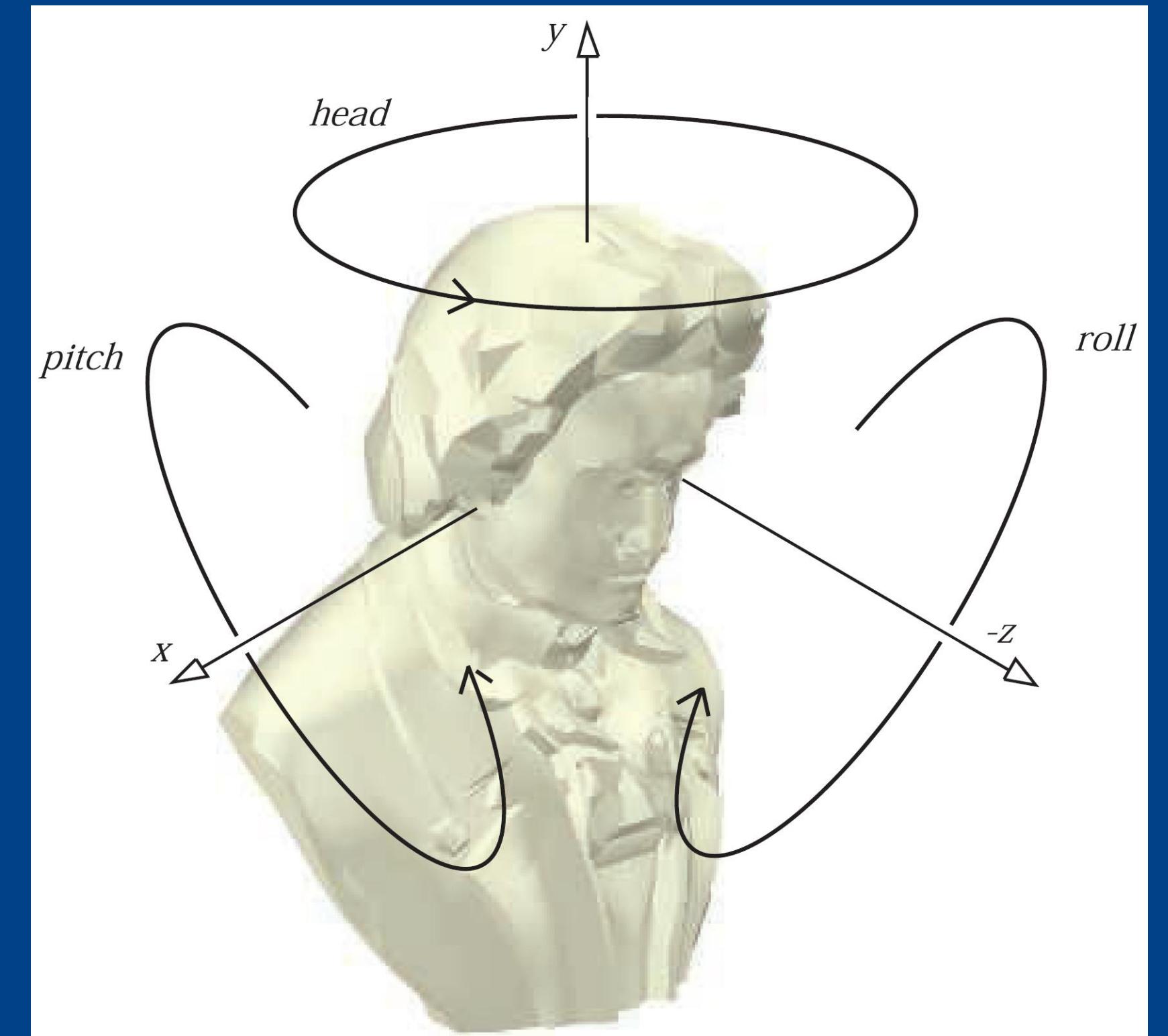
- Yaw / Pitch / Roll

- Yaw(도리) : 좌우로 고개 돌리기
- Pitch(끄덕) : 위아래로 끄덕이기
- Roll(갸웃) : 머리 기울이기

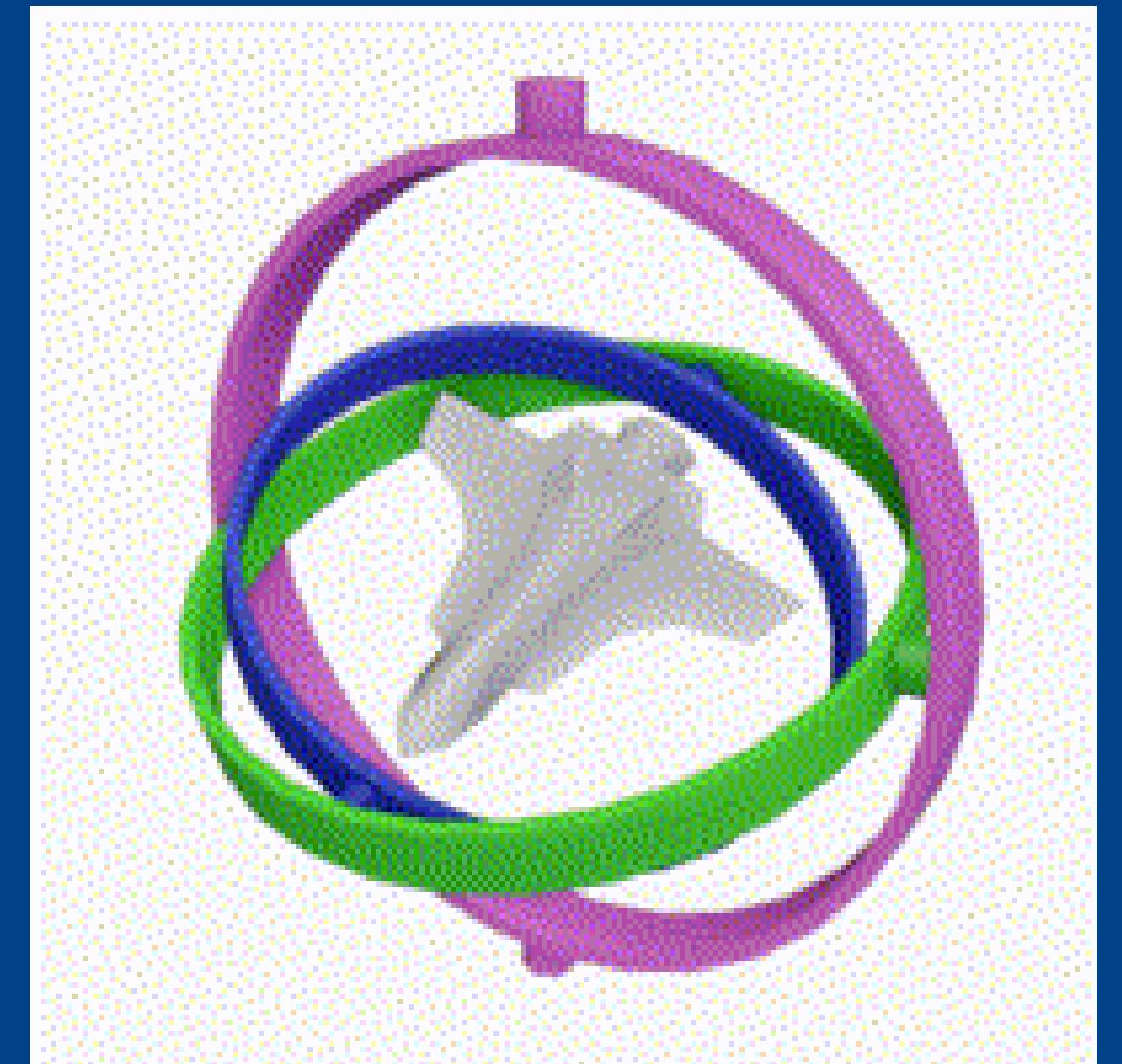
- 행렬로 표현

$$\mathbf{E}(h, p, r) = \mathbf{R}_z(r)\mathbf{R}_x(p)\mathbf{R}_y(h)$$

- 회전의 결합이므로 직교 행렬 $\rightarrow \mathbf{E}^{-1} = \mathbf{E}^T$
- 하지만 “어떤 순서로 곱하느냐”가 정의의 일부
- 시스템 / 도구마다 순서가 다를 수 있음 (애니메이션 / 물리 등)



- 짐벌락(Gimbal Lock) : 회전 자유도가 하나 “사라지는” 현상
 - 무슨 일이 생기나?
 - 3개의 회전축이 특정 각도에서 서로 겹치며 3D 회전이 2D 회전처럼 됨
 - 예 : Pitch가 90도 근처면 Yaw와 Roll이 사실상 같은 축처럼 움직임
 - 왜 문제인가?
 - 조작(컨트롤)이 갑자기 이상해짐
 - 부드러운 보간이 어렵고, 경로가 튕는 것처럼 보일 수 있음
 - 완전히 피할 수 있나?
 - 특정 표현(오일러 각도)에서는 구조적으로 발생 가능
 - 다른 표현(사원수 등)으로 “표현을 바꾸는” 것이 일반적



- 오일러 각도 추출 : 행렬 \rightarrow (Yaw, Pitch, Roll)

- 언제 필요할까?

- 엔진 내부는 행렬인데 UI / 툴은 각도로 보여줘야 할 때
- 외부 입력(트래커 / 컨트롤러)에서 온 회전을 제한하고 싶을 때

- 기본 아이디어

- 회전 행렬의 원소를 이용해 \sin / \cos 관계를 풀어낸다
- atan2 를 사용하면 사분면까지 안전하게 복원 가능

- 특수 케이스 (짐벌락 근처)

- $\cos(\mathbf{p}) = 0$ 이면 atan2 로 분리 불가 \rightarrow 한 각도를 0으로 두고 나머지로 흡수 (관례)
- 수치 오차로 $|\sin| > 1$ 이면 \arcsin 실패 \rightarrow 클램프(Clamp)가 필요할 수 있음

$$\mathbf{E}(h, p, r) = \begin{pmatrix} e_{00} & e_{01} & e_{02} \\ e_{10} & e_{11} & e_{12} \\ e_{20} & e_{21} & e_{22} \end{pmatrix} = \mathbf{R}_z(r)\mathbf{R}_x(p)\mathbf{R}_y(h)$$

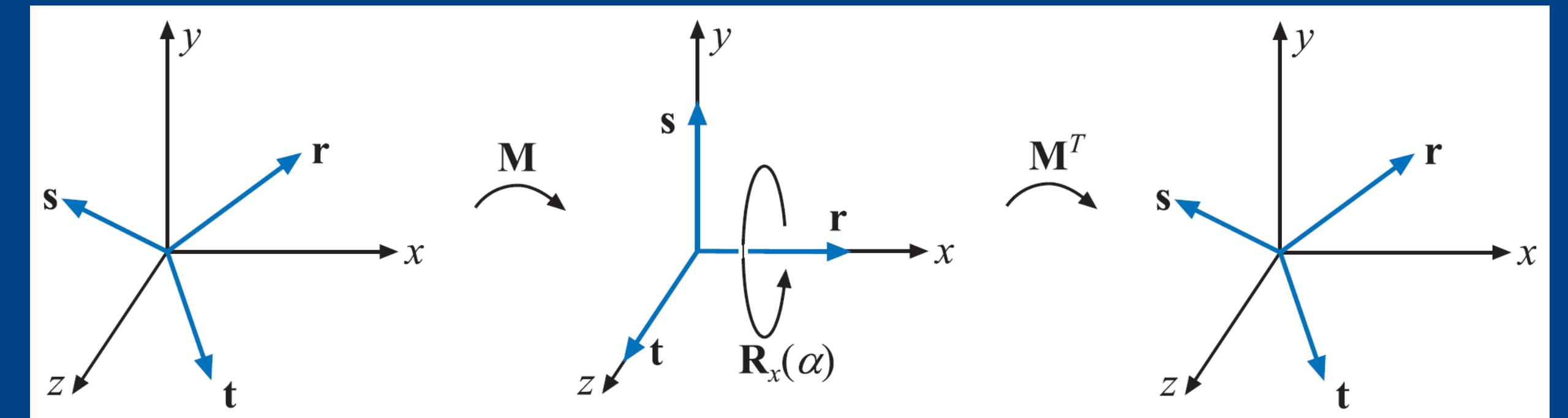
$$h = \text{atan2}(-e_{20}, e_{22}), \\ p = \arcsin(e_{21}), \\ r = \text{atan2}(-e_{01}, e_{11}).$$

- 행렬 분해 (Matrix Decomposition) : “합쳐진” 행렬에서 성분 다시 찾기
- 왜 필요한가?
 - 스케일만 뽑아내고 싶다
 - 특정 포맷 / 시스템(VRML 등)이 허용하는 변환만 필요하다
 - 강체 변환인지 판별해야 한다
 - 키프레임 사이 “행렬”만 주어질 때 보간해야 한다
- 난이도 포인트
 - 이동은 쉬움 (마지막 열)
 - 회전 / 스케일 / 전단을 분리하는 건 어렵고 수치 문제도 있음
 - 실무에서는 검증된 알고리즘 / 라이브러리 사용 추천

- 임의 축 주위 회전 : “기저(Basis)를 바꿔서 z축 회전으로 환원”

- 문제

- 정규화된 축 u 주위로 θ 만큼 회전하고 싶다
- 하지만 기본 회전은 $x/y/z$ 축 기준
- 해결 아이디어 (기저 변환)
 - u 를 새 기저의 한 축으로 삼는다
 - 새 기저에서 u 가 x 축이 되도록 정렬 $\rightarrow R_z(\theta)$ 적용
 - 마지막에 원래 기저로 되돌림
 - $M = BR_z(\theta)B^{-1}$



- 회전 표현
- 행렬 변환
- SLERP
- 벡터 → 벡터 회전

- 왜 사원수(Quaternion)를 쓰나?

- 오일러 / 행렬의 불편함
 - 오일러 : 짐벌락 + 보간이 까다로움
 - 행렬 : 저장 / 보간 시 직교성 유지가 어려울 수 있음 (수치 오차)
 - 축-각 : 결합 / 보간을 매번 다시 계산해야 함
- 사원수의 장점
 - 모든 3D 회전을 간결하게 표현 (단위 사원수)
 - 회전 결합이 곱셈으로 간단
 - SLERP로 일정한 각속도 (부드러운 회전) 보간
 - 표현이 연속적이라 애니메이션 / 카메라에 강함

- 사원수 기본 : 실수부 s + 허수부 벡터 \mathbf{v}

- 표현

- $\hat{q} = (s, \mathbf{v}) = (s, x\mathbf{i} + y\mathbf{j} + z\mathbf{k})$
- s : 실수부, $\mathbf{v} = (x, y, z)$: 허수부 (벡터처럼 다룸)
- i, j, k 는 허수 단위이며 곱셈은 비가환적

- 자주 쓰는 연산

- 곱셈 : $(s_1, \mathbf{v}_1)(s_2, \mathbf{v}_2) = (s_1s_2 - \mathbf{v}_1 \cdot \mathbf{v}_2, s_1\mathbf{v}_2 + s_2\mathbf{v}_1 + \mathbf{v}_1 \times \mathbf{v}_2)$
- 덧셈 : $(s_1, \mathbf{v}_1) + (s_2, \mathbf{v}_2) = (s_1 + s_2, \mathbf{v}_1 + \mathbf{v}_2)$
- 결레 : $(s, \mathbf{v})^* = (s, -\mathbf{v})$
- 크기 : $\|(s, \mathbf{v})\| = \sqrt{s^2 + \|\mathbf{v}\|^2}$
- 역원 : $(s, \mathbf{v})^{-1} = (s, \mathbf{v})^*/\|(s, \mathbf{v})\|^2$

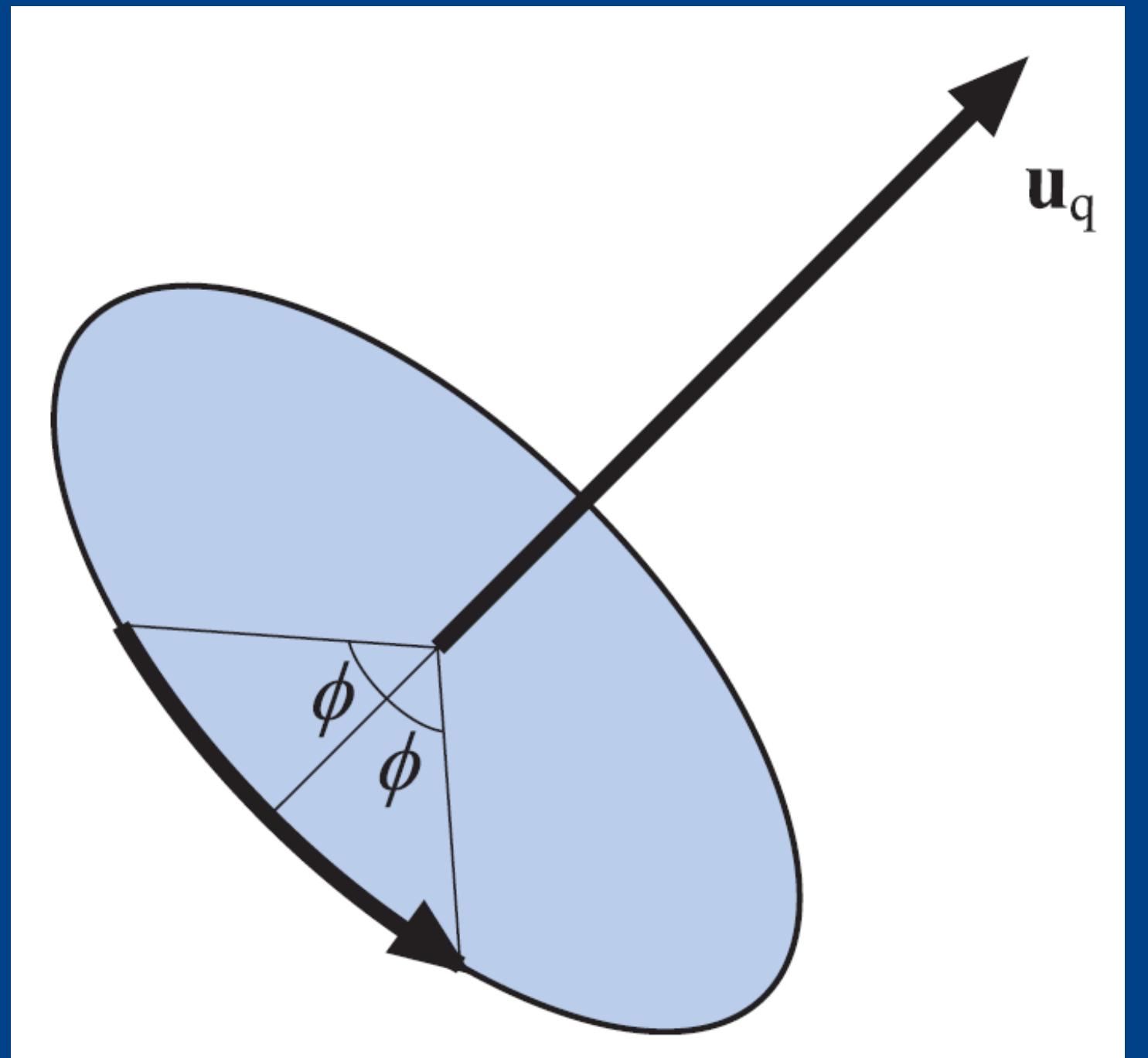
- 단위 사원수로 회전하기 : $\mathbf{p}' = \hat{\mathbf{q}}\hat{\mathbf{p}}\hat{\mathbf{q}}^{-1}$

- 회전 적용 (점 / 벡터)

- 3D 벡터 \mathbf{p} 를 $(0, \mathbf{p})$ 형태의 “순허수 사원수”로 둔다
- 단위 사원수 \mathbf{q} 로 \mathbf{p} 를 감싸듯이 곱하면 회전이 된다
- $\mathbf{p}' = \hat{\mathbf{q}}(0, \mathbf{p})\hat{\mathbf{q}}^{-1}$

- 축-각(Axis-Angle)과의 연결

- 축이 \mathbf{u} (정규화)이고, 각이 θ 라면 $\hat{\mathbf{q}} = \left(\cos \frac{\theta}{2}, \mathbf{u} \sin \frac{\theta}{2}\right)$
- 중요 성질 : \mathbf{q} 와 $-\mathbf{q}$ 는 같은 회전을 표현한다
(따라서 보간 시 “부호 맞추기” 필요)



- 사원수 결합 및 행렬 변환

- 회전 결합 : $\hat{\mathbf{q}} = \hat{\mathbf{q}}_2 \hat{\mathbf{q}}_1$

- 행렬과 마찬가지로 순서 중요 (곱셈은 비가환)
- 단위 사원수의 곱은 다시 단위 사원수 (수치 오차 제외)

- 사원수 \leftrightarrow 회전 행렬

- GPU / 엔진 내부에서 행렬이 더 편한 경우가 많아 상호 변환이 필요
- 사원수를 회전 행렬로 변환하는 방법은 비교적 단순

$$\mathbf{M}^q = \begin{pmatrix} 1 - 2(q_y^2 + q_z^2) & 2(q_x q_y - q_w q_z) & 2(q_x q_z + q_w q_y) & 0 \\ 2(q_x q_y + q_w q_z) & 1 - 2(q_x^2 + q_z^2) & 2(q_y q_z - q_w q_x) & 0 \\ 2(q_x q_z - q_w q_y) & 2(q_y q_z + q_w q_x) & 1 - 2(q_x^2 + q_y^2) & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix}$$

- 사원수 결합 및 행렬 변환

- 사원수 \leftrightarrow 회전 행렬

- 회전 행렬을 사원수로 역변환하는 방법은 복잡 (수치 안정성을 고려해야 함)

- q_w 를 알고 있다면 벡터 \mathbf{v}_q 의 값을 계산할 수 있으며, 따라서 $\hat{\mathbf{q}}$ 를 유도할 수 있다

$$\begin{aligned} m_{21}^q - m_{12}^q &= 4q_w q_x, \\ m_{02}^q - m_{20}^q &= 4q_w q_y, \\ m_{10}^q - m_{01}^q &= 4q_w q_z. \end{aligned}$$

- 행렬 \mathbf{M}^q 의 대각합은 다음과 같이 계산할 수 있다

$$\begin{aligned} \text{tr}(\mathbf{M}^q) &= 4 - 2s(q_x^2 + q_y^2 + q_z^2) = 4 \left(1 - \frac{q_x^2 + q_y^2 + q_z^2}{q_x^2 + q_y^2 + q_z^2 + q_w^2} \right) \\ &= \frac{4q_w^2}{q_x^2 + q_y^2 + q_z^2 + q_w^2} = \frac{4q_w^2}{(n(\hat{\mathbf{q}}))^2}. \end{aligned}$$

- 이를 통해 단위 사원수에 대해 다음과 같은 변환을 할 수 있다

$$\begin{aligned} q_w &= \frac{1}{2} \sqrt{\text{tr}(\mathbf{M}^q)}, & q_x &= \frac{m_{21}^q - m_{12}^q}{4q_w}, \\ q_y &= \frac{m_{02}^q - m_{20}^q}{4q_w}, & q_z &= \frac{m_{10}^q - m_{01}^q}{4q_w}. \end{aligned}$$

- 사원수 결합 및 행렬 변환

- 사원수 \leftrightarrow 회전 행렬

- 회전 행렬을 사원수로 역변환하는 방법은 복잡 (수치 안정성을 고려해야 함)

- 수치적으로 안정된 루틴을 만들려면 작은 수로 나누는 경우는 피해야 한다

따라서, 우선 $t = q_w^2 - q_x^2 - q_y^2 - q_z^2$ 로 설정하고 다음과 같은 수식을 유도한다

$$m_{00} = t + 2q_x^2,$$

$$m_{11} = t + 2q_y^2,$$

$$m_{22} = t + 2q_z^2,$$

$$u = m_{00} + m_{11} + m_{22} = t + 2q_w^2,$$

- 사원수 결합 및 행렬 변환

- 사원수 \leftrightarrow 회전 행렬

- 회전 행렬을 사원수로 역변환하는 방법은 복잡 (수치 안정성을 고려해야 함)

- 위 식을 통해 m_{00}, m_{11}, m_{22} 의 최댓값과 u 값에 의해 q_x, q_x, q_x, q_x 중에서 어떤 값이 가장 큰지가 결정된다
만약 q_x 가 최댓값이면 이전 변환식을 통해 사원수를 만들 수 있다
- 그렇지 않은 경우는 다음과 같은 조건들이 만족된다

$$4q_x^2 = +m_{00} - m_{11} - m_{22} + m_{33},$$

$$4q_y^2 = -m_{00} + m_{11} - m_{22} + m_{33},$$

$$4q_z^2 = -m_{00} - m_{11} + m_{22} + m_{33},$$

$$4q_w^2 = \text{tr}(\mathbf{M}^q).$$

- 구면 선형 보간 (Spherical Linear Interpolation; SLERP)

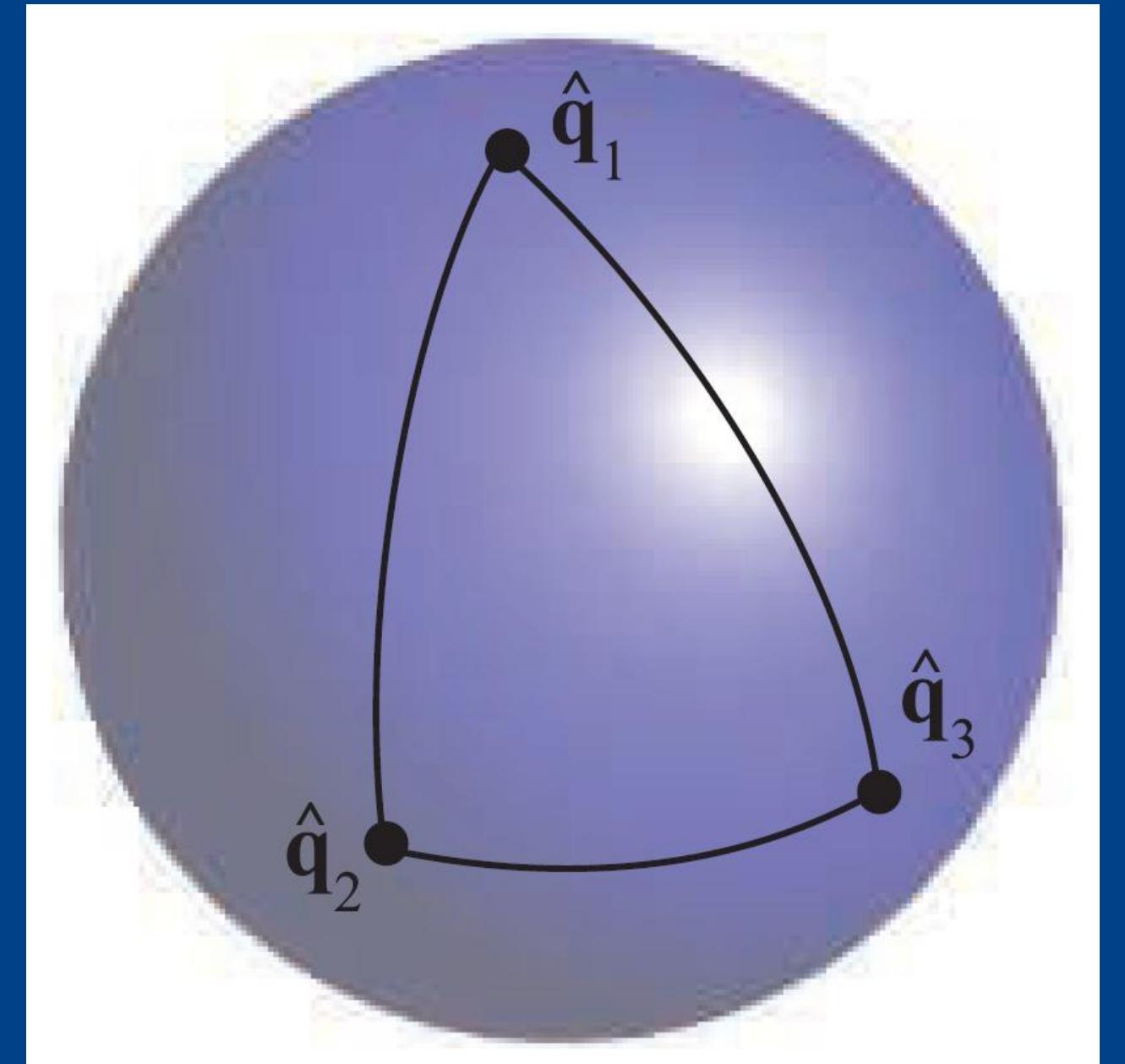
- 무엇을 하는가?

- 두 단위 사원수 $\hat{\mathbf{q}}, \hat{\mathbf{r}}$ 사이를 $t \in [0, 1]$ 로 보간해 중간 방향을 만든다
- 단위 구면 위의 “최단 원호”를 따라 이동 \rightarrow 각속도가 일정

- 핵심 식

$$\hat{\mathbf{s}}(\hat{\mathbf{q}}, \hat{\mathbf{r}}, t) = \text{slerp}(\hat{\mathbf{q}}, \hat{\mathbf{r}}, t) = \frac{\sin(\phi(1-t))}{\sin \phi} \hat{\mathbf{q}} + \frac{\sin(\phi t)}{\sin \phi} \hat{\mathbf{r}}.$$

- ϕ 는 두 사원수 사이의 각 ($\cos \phi = q_x r_x + q_y r_y + q_z r_z + q_w r_w$)
- $\hat{\mathbf{q}}$ 와 $-\hat{\mathbf{q}}$ 가 같은 회전이므로, 내적이 음수면 한쪽 부호를 뒤집어 “최단 경로”를 선택



- 구면 선형 보간 (Spherical Linear Interpolation; SLERP)

- 보간을 하기 위한 더 좋은 방법 : 스플라인 (Spline)

- $\hat{\mathbf{q}}_i$ 와 $\hat{\mathbf{q}}_{i+1}$ 사이에 $\hat{\mathbf{a}}_i$ 와 $\hat{\mathbf{a}}_{i+1}$ 이라는 사원수를 도입한다

$$\hat{\mathbf{a}}_i = \hat{\mathbf{q}}_i \exp \left[-\frac{\log(\hat{\mathbf{q}}_i^{-1} \hat{\mathbf{q}}_{i-1}) + \log(\hat{\mathbf{q}}_i^{-1} \hat{\mathbf{q}}_{i+1})}{4} \right]$$

- $\hat{\mathbf{q}}_i, \hat{\mathbf{a}}_i, \hat{\mathbf{a}}_{i+1}, \hat{\mathbf{q}}_{i+1}$ 을 통해 구형 3차 보간을 정의할 수 있다
- 그리고 $\hat{\mathbf{q}}_i, \hat{\mathbf{a}}_i$ 는 부드러운 3차 스플라인을 이용해 사원수들을 구형으로 보간하는데 사용된다
(`squad` 함수는 `slerp` 함수를 이용해 반복적으로 구형 보간하는 함수다)

$$\begin{aligned} \text{squad}(\hat{\mathbf{q}}_i, \hat{\mathbf{q}}_{i+1}, \hat{\mathbf{a}}_i, \hat{\mathbf{a}}_{i+1}, t) &= \\ \text{slerp}(\text{slerp}(\hat{\mathbf{q}}_i, \hat{\mathbf{q}}_{i+1}, t), \text{slerp}(\hat{\mathbf{a}}_i, \hat{\mathbf{a}}_{i+1}, t), 2t(1-t)). \end{aligned}$$

- 한 벡터에서 다른 벡터로 : “최단 회전” 사원수 만들기
- 문제
 - 정규화된 방향 \mathbf{s} 에서 \mathbf{t} 로 돌리는 회전을 구하고 싶다
 - 카메라 방향 맞추기, 조준(Aim), 물체 정렬 등에서 빈번하게 사용
- 핵심 식

$$\hat{\mathbf{q}} = (\mathbf{q}_v, q_w) = \left(\frac{1}{\sqrt{2(1+e)}}(\mathbf{s} \times \mathbf{t}), \frac{\sqrt{2(1+e)}}{2} \right)$$

- $\mathbf{s} \cdot \mathbf{t}$ 가 1에 가까울 때(거의 같은 방향)도 안정적으로 동작
- $\mathbf{s} \cdot \mathbf{t}$ 가 -1(정반대)이면 분모가 0 → 임의의 수직 축을 선택해 180° 회전

- 한 벡터에서 다른 벡터로 : “최단 회전” 사원수 만들기
- 핵심 식
 - 때로는 \mathbf{s} 에서 \mathbf{t} 로 회전하는 작업을 행렬로 표현해야 할 필요가 있다

$$\mathbf{R}(\mathbf{s}, \mathbf{t}) = \begin{pmatrix} e + hv_x^2 & hv_xv_y - v_z & hv_xv_z + v_y & 0 \\ hv_xv_y + v_z & e + hv_y^2 & hv_yv_z - v_x & 0 \\ hv_xv_z - v_y & hv_yv_z + v_x & e + hv_z^2 & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix}$$

$$\mathbf{v} = \mathbf{s} \times \mathbf{t},$$

$$e = \cos(2\phi) = \mathbf{s} \cdot \mathbf{t},$$

$$h = \frac{1 - \cos(2\phi)}{\sin^2(2\phi)} = \frac{1 - e}{\mathbf{v} \cdot \mathbf{v}} = \frac{1}{1 + e}.$$

- 정점 혼합 (스키닝)
- 모핑 (블랜드 셰이프)

- 정점 혼합 (Vertex Blending / Skinning) : 뼈의 변환을 정점에 가중합

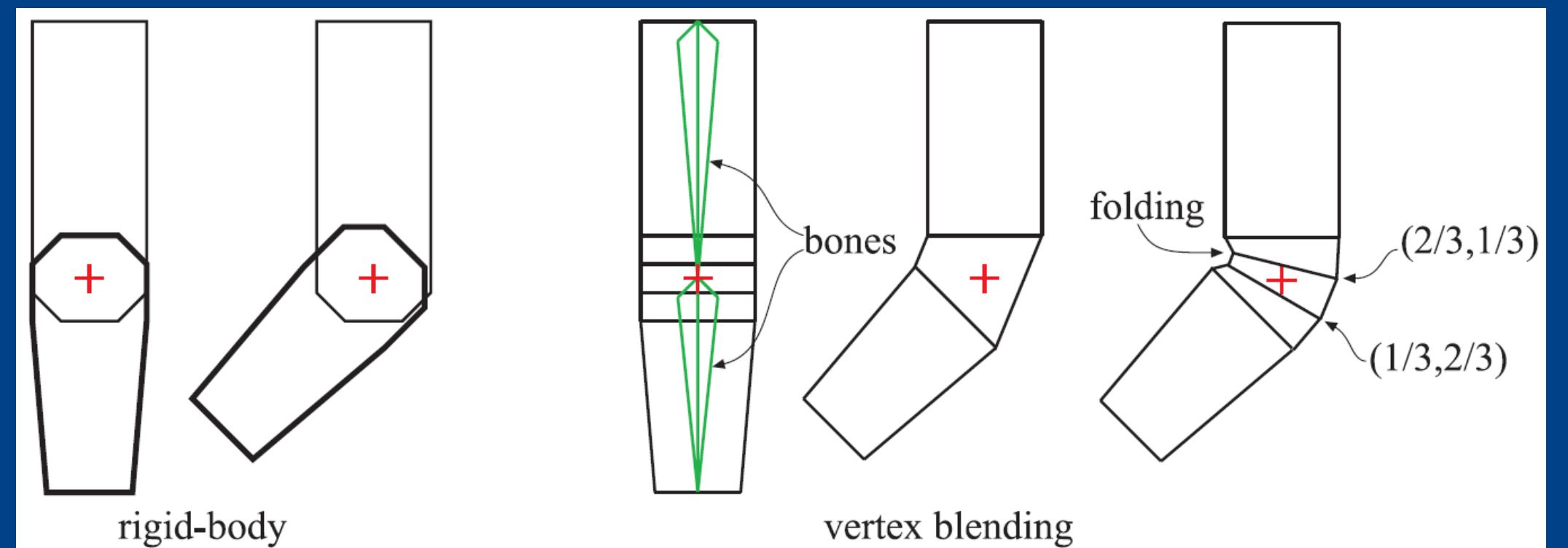
- 왜 필요할까?

- 아래팔 / 위팔을 통째로 강체 변환하면 관절이 딱딱해 보임
- 하나의 메시를 유지하면서 관절 주변 정점이 “부드럽게” 따라가야 자연스럽다

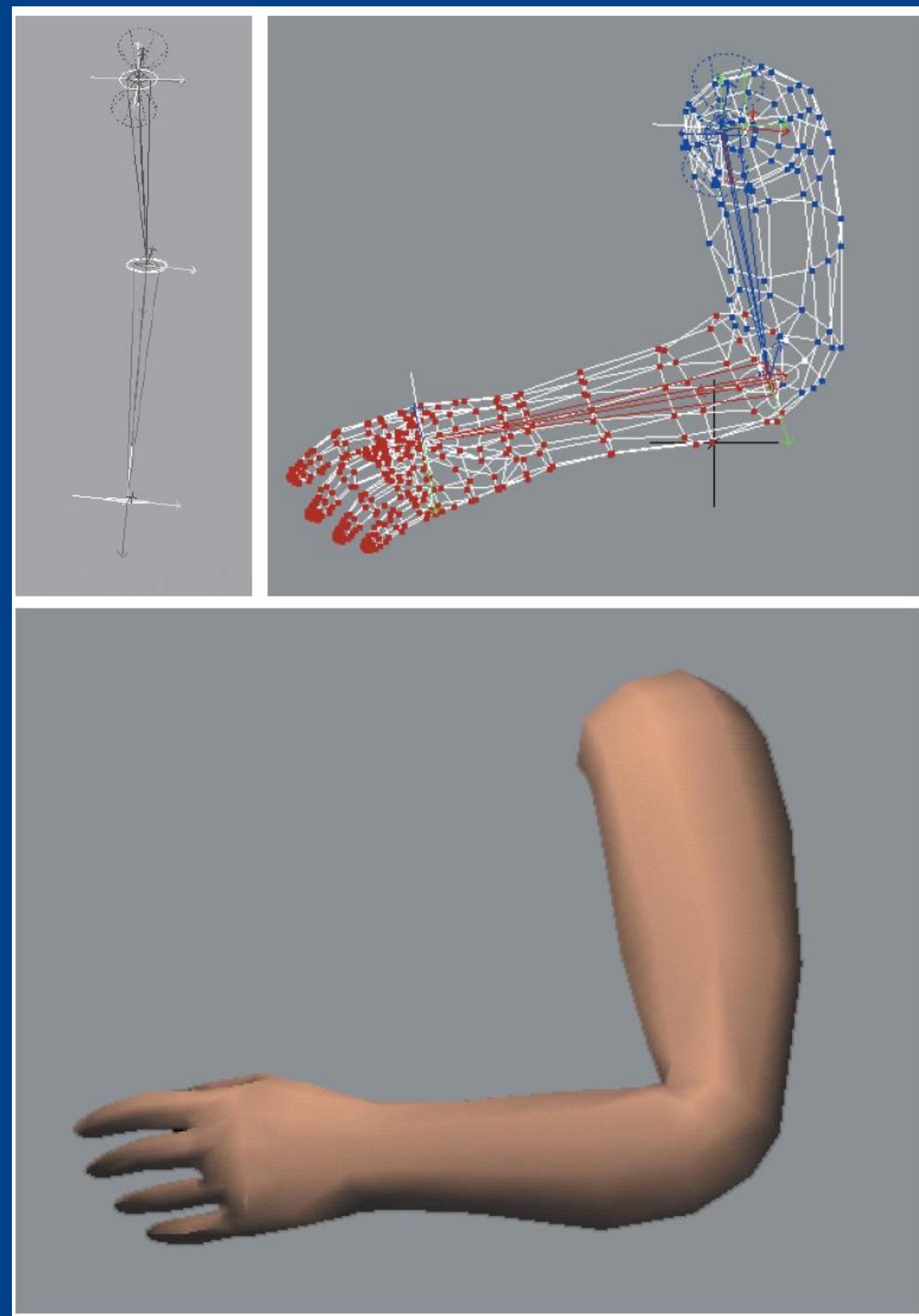
- 기본 공식 (선형 혼합 스키닝)

$$\mathbf{u}(t) = \sum_{i=0}^{n-1} w_i \mathbf{B}_i(t) \mathbf{M}_i^{-1} \mathbf{p}, \quad \text{where} \quad \sum_{i=0}^{n-1} w_i = 1, \quad w_i \geq 0.$$

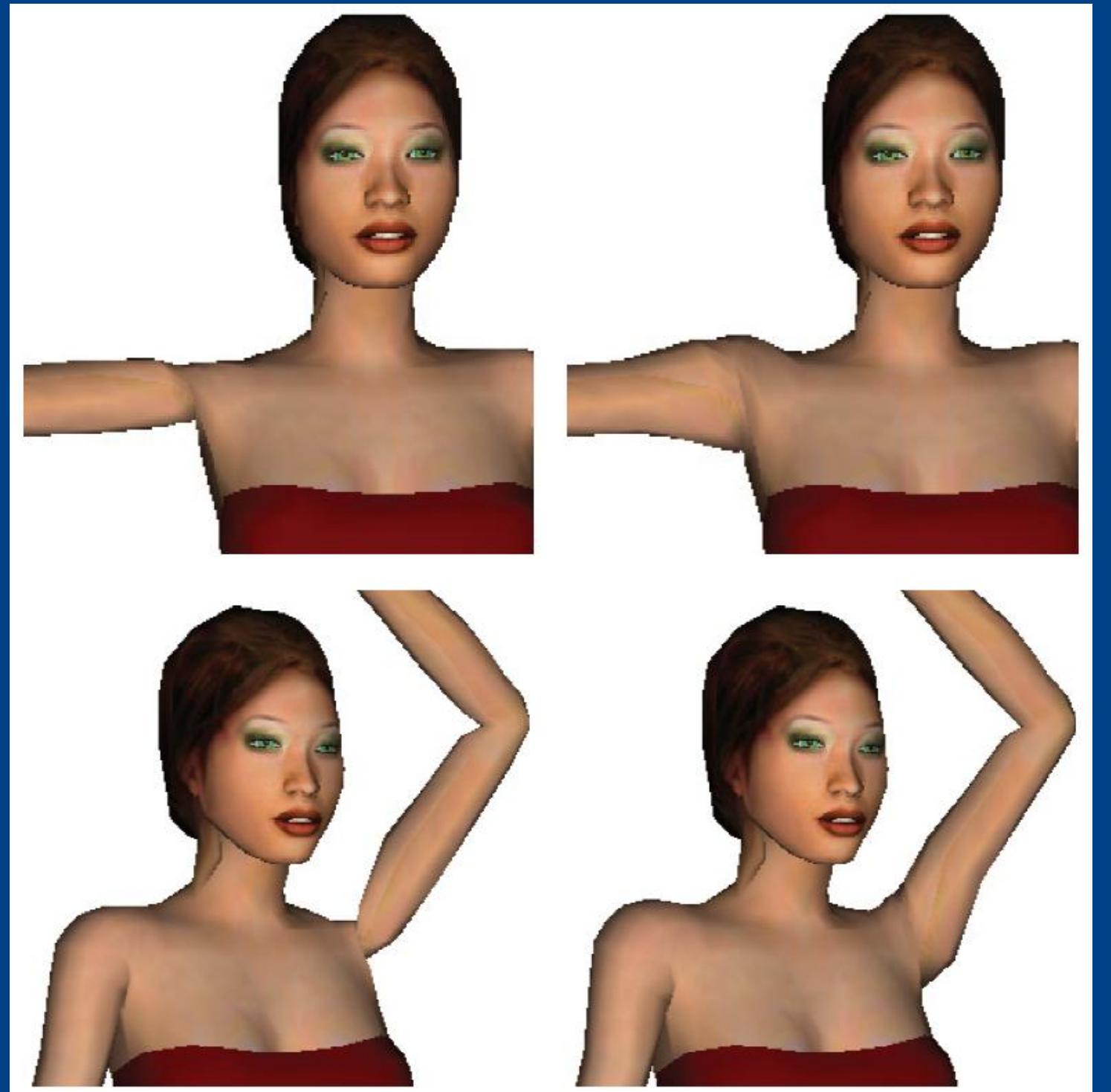
- GPU 정점 셰이더에 매우 적합
(뼈 행렬만 매 프레임 업데이트)



- 정점 혼합 (Vertex Blending / Skinning) : 뼈의 변화를 정점에 가중합

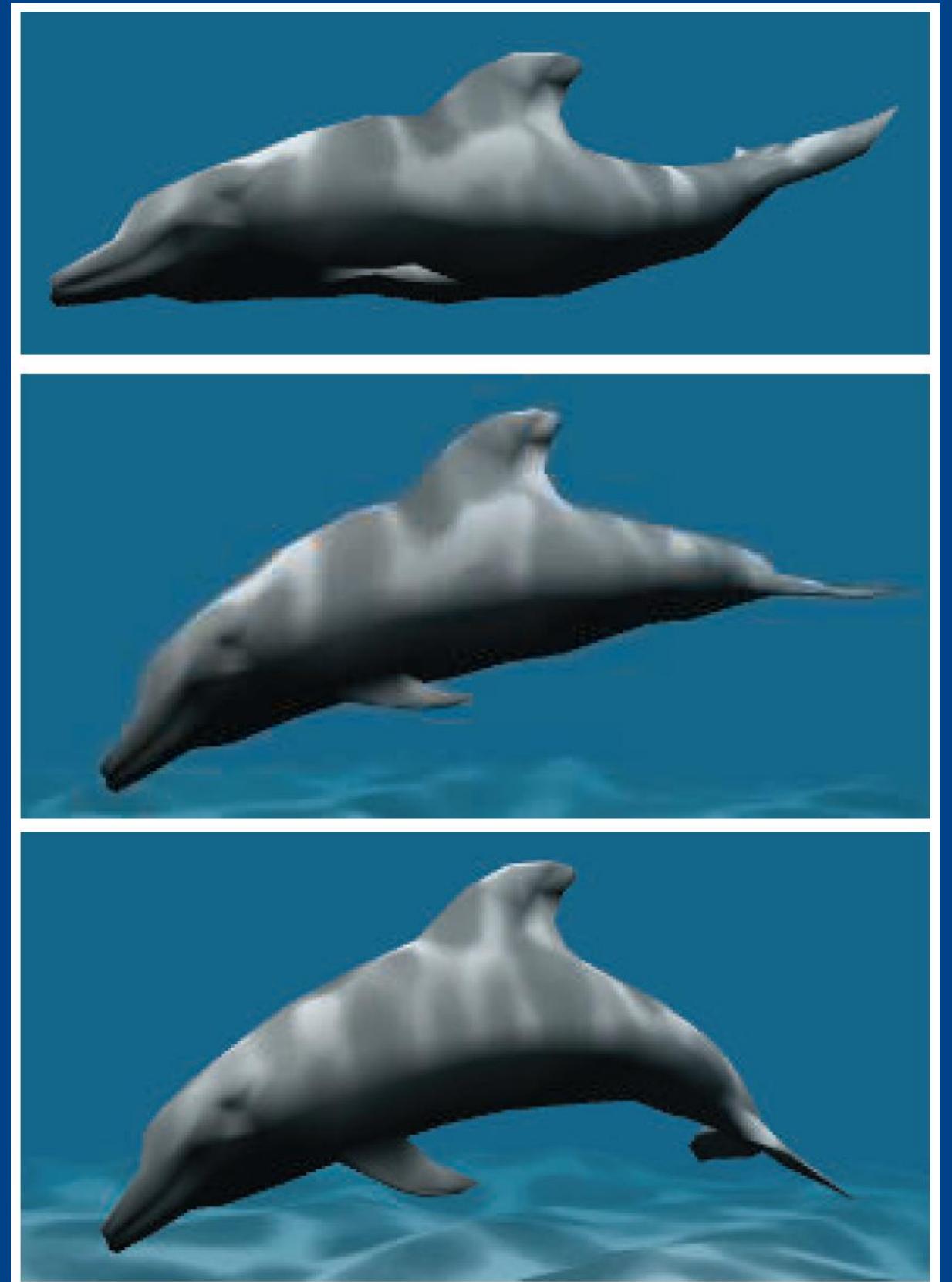


- 스키닝의 단점과 개선
 - 선형 혼합 스키닝(LBS)의 단점
 - 관절 안쪽이 “접하는” 현상
 - 비틀림에서 사탕 포장지처럼 찌그러짐
 - 자기 교차 등 원치 않는 변형이 발생할 수 있음
 - 개선 방향
 - 가중치 / 뼈 추가로 완화 가능하지만 한계가 있음
 - 이중 사원수(Dual Quaternion) 스키닝이 대표적 해결책
 - LBS 대비 비용이 약간 늘지만 품질이 크게 개선되는 경우가 많음



- 모핑(Morphing) & 블랜드 셰이프(Blend Shape)

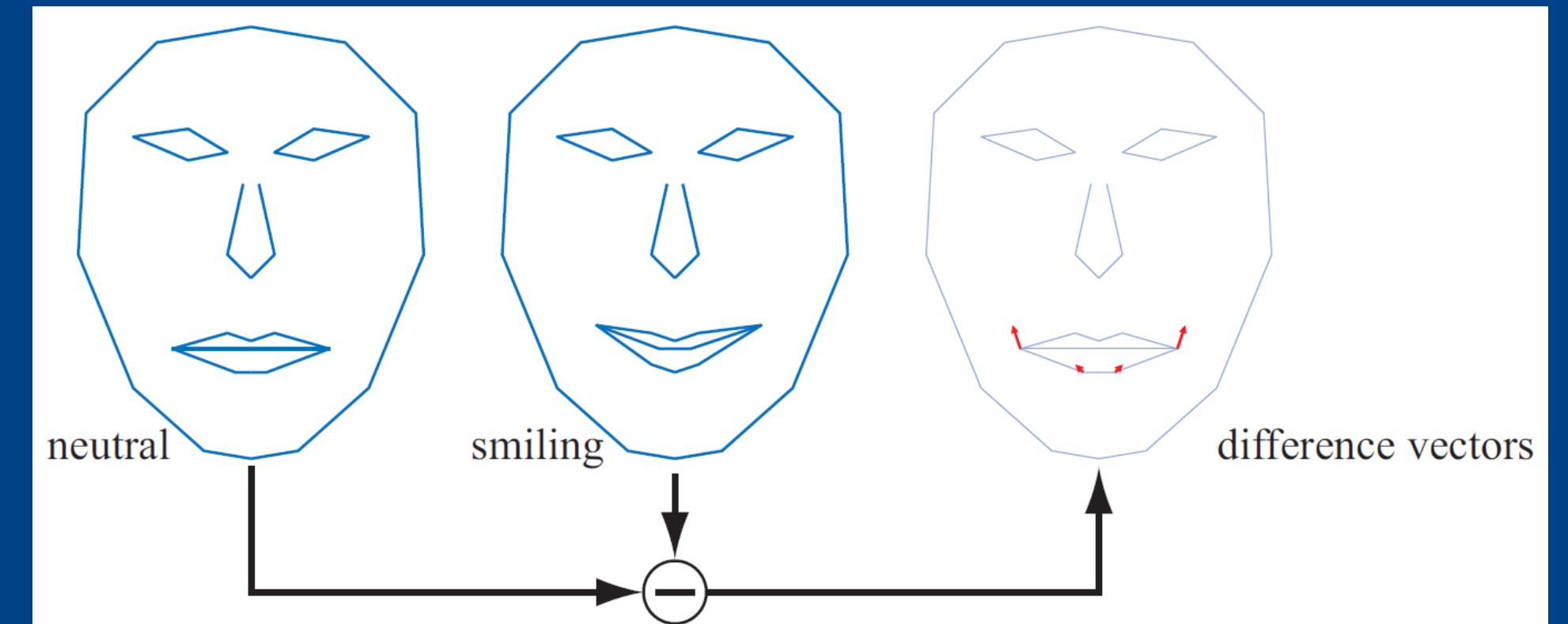
- 모핑 = 두 모델 사이를 보간
 - 정점 대응(같은 정점 번호가 서로 대응)이 있으면 간단해짐
 - 프레임마다 선형 보간으로 모핑된 정점을 만들 수 있다
 - 시각 $t \in [t_0, t_1]$ 에 대해 모핑된 정점을 계산하기 위해, 먼저 $s = (t - t_0)/(t_1 - t_0)$ 를 계산한다
 - 이제 $\mathbf{m} = (1 - s)\mathbf{p}_0 + s\mathbf{p}_1$ 를 통해 선형 정점 혼합을 수행한다



- 모핑(Morphing) & 블랜드 셰이프(Blend Shape)

- 블랜드 셰이프 = 여러 표정 / 포즈를 조합

- 일반적으로 $\mathcal{P}_i, i \in [1, \dots, k]$ 인 $k \geq 1$ 개의 다른 포즈들이 있다
- 전처리 단계로, “얼굴 차이”는 $\mathcal{D}_i = \mathcal{P}_i - \mathcal{N}$ 으로 계산된다
- 이제 감정을 드러내지 않는 포즈 \mathcal{N} 과 다른 포즈들의 집합 \mathcal{D}_i 를 가지고 있다. 모핑된 \mathcal{M} 은 다음 식을 이용해 구할 수 있다



$$\mathcal{M} = \mathcal{N} + \sum_{i=0}^k w_i \mathcal{D}_i$$

- 가중치 w_i 를 사용해 감정을 드러내지 않는 포즈에서 원하는 다른 포즈들의 특징들을 추가한다
- GPU에서 정점 셰이더로 처리 가능

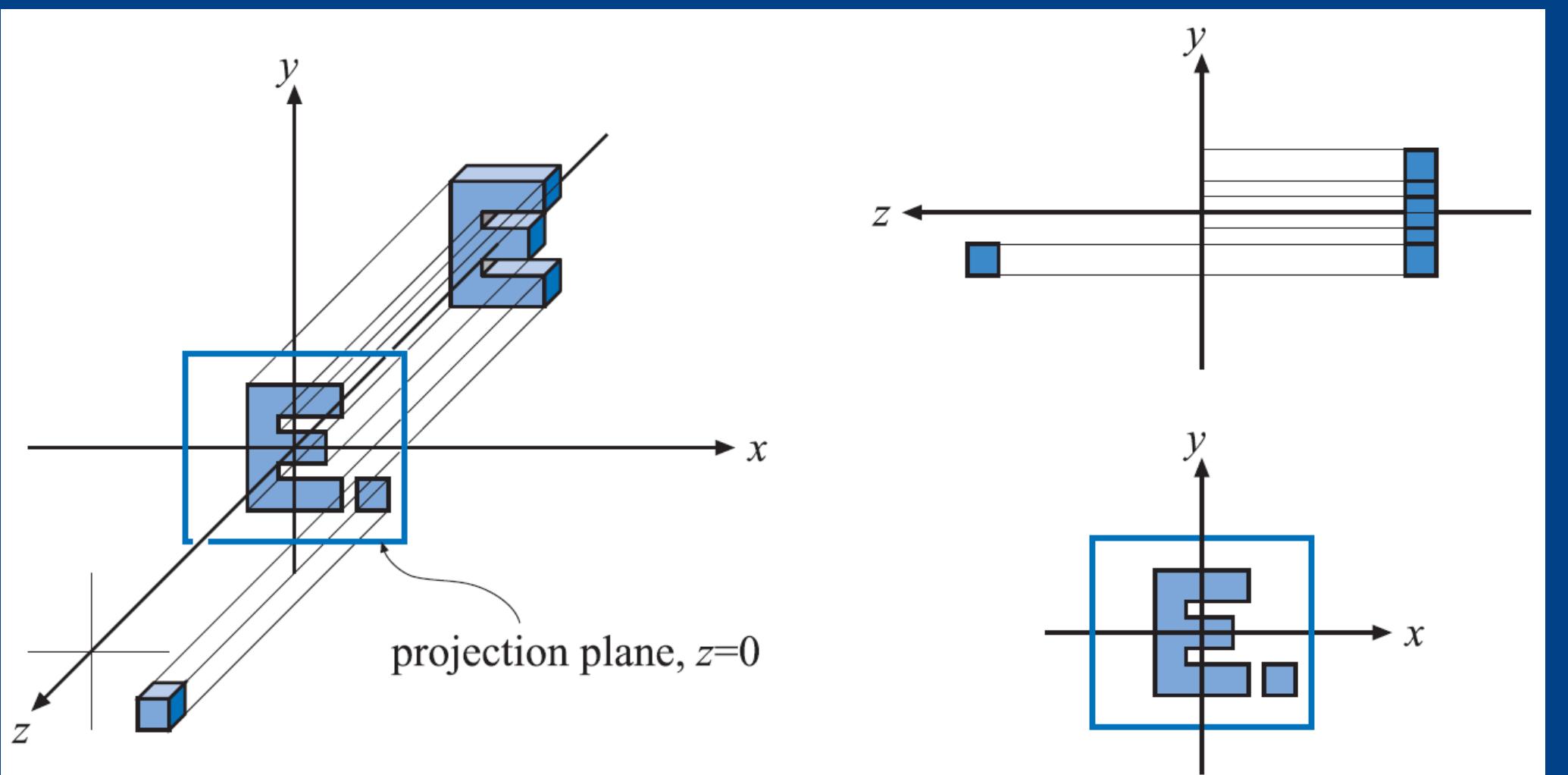
- 직교 투영
- 원근 투영
- NDC
- 깊이 정밀도

- 투영 전후의 흐름 : View Space → Clip Space → NDC → Screen
 - 파이프라인
 - 모델 (로컬) → 월드 → 뷰 (카메라) → 투영 (클립 공간)
 - 클리핑(정규 시야 영역) 후 동차화(나눗셈) → NDC
 - 뷰포트 변환으로 픽셀 좌표로 맵핑
 - 동차화 (Homogenization)
 - 투영 행렬은 w 를 1로 유지하지 않는다
 - 클립 공간에서 $(x, y, z, w) \rightarrow (x/w, y/w, z/w)$ 로 나눠 NDC를 얻는다

- 직교(Orthographic) 투영 : 평행선이 그대로 평행

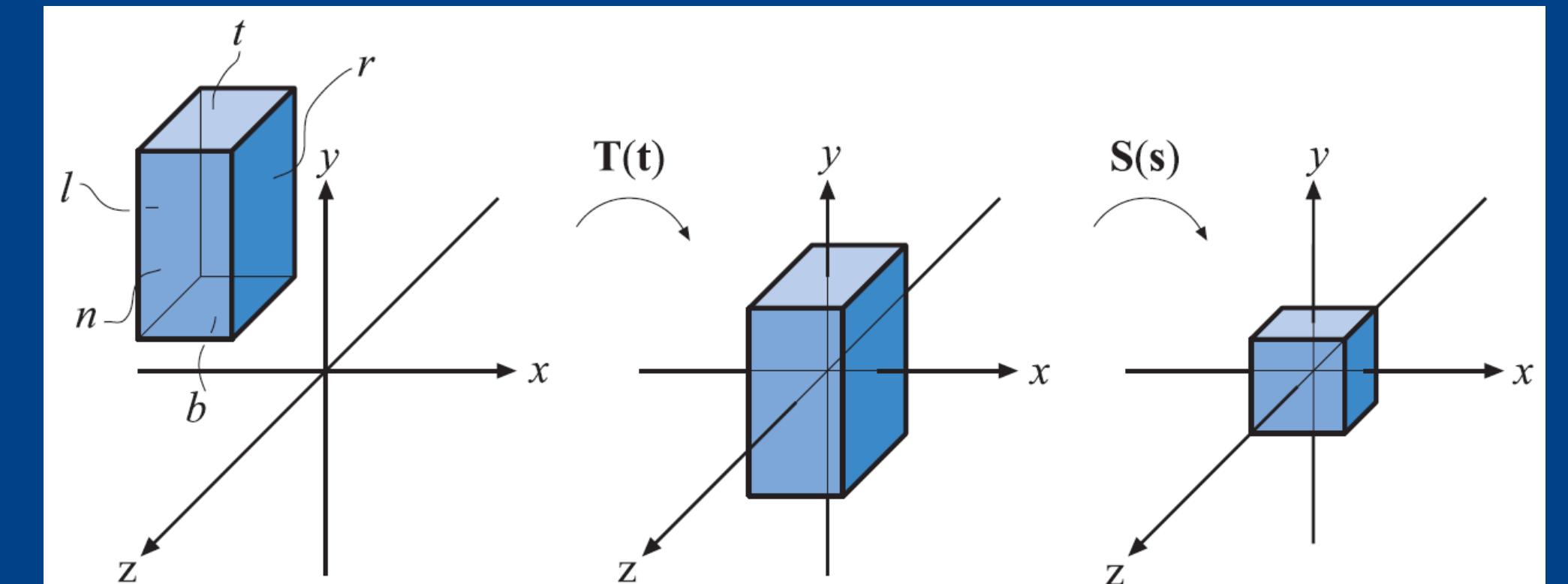
- 특징

- 원근감이 없다 (거리와 무관)
- CAD / 에디터 / 미니맵 / 2D UI 등에서 유용
- $z = 0$ 인 평면 상에 수직으로 투영한다 (역행렬 존재하지 않음)



- 직교(Orthographic) 투영 : 평행선이 그대로 평행
- 일반적인 직교 투영 (박스 → NDC)
 - 왼쪽(l), 오른쪽(r), 아래(b), 위(t), 근평면(n), 원평면(f)
 - AABB를 중심 원점의 정규 시야 영역으로 이동 + 스케일
 - 투영은 “보는 방향(z)”을 기준으로 근/원 평면으로 잘라(Clipping) 정규화한다

$$\begin{aligned}
 P_o = S(s)T(t) &= \begin{pmatrix} \frac{2}{r-l} & 0 & 0 & 0 \\ 0 & \frac{2}{t-b} & 0 & 0 \\ 0 & 0 & \frac{2}{f-n} & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix} \begin{pmatrix} 1 & 0 & 0 & -\frac{l+r}{2} \\ 0 & 1 & 0 & -\frac{t+b}{2} \\ 0 & 0 & 1 & -\frac{f+n}{2} \\ 0 & 0 & 0 & 1 \end{pmatrix} \\
 &= \begin{pmatrix} \frac{2}{r-l} & 0 & 0 & -\frac{r+l}{r-l} \\ 0 & \frac{2}{t-b} & 0 & -\frac{t+b}{t-b} \\ 0 & 0 & \frac{2}{f-n} & -\frac{f+n}{f-n} \\ 0 & 0 & 0 & 1 \end{pmatrix}.
 \end{aligned}$$



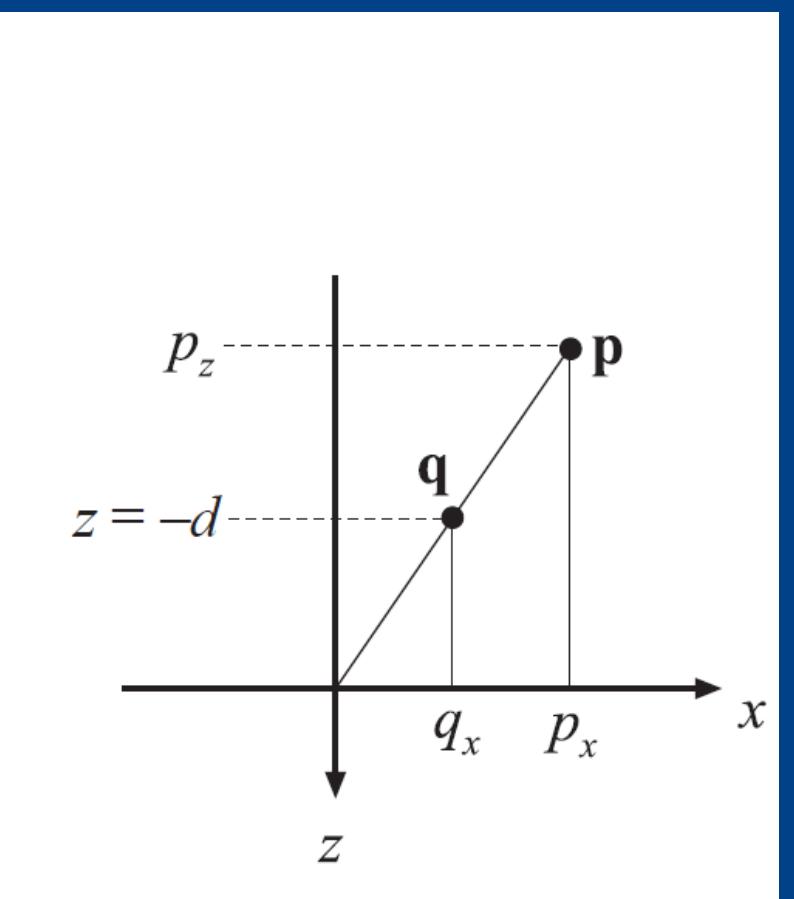
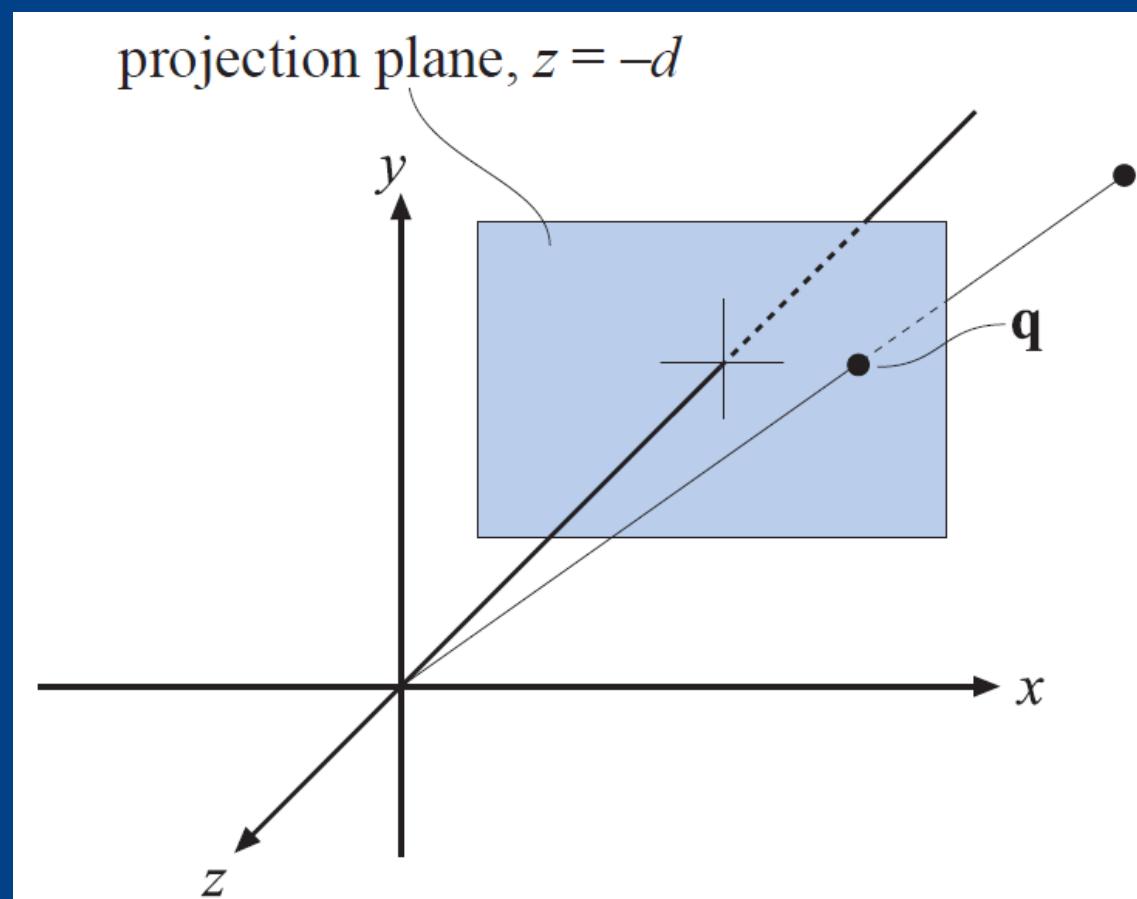
- 원근(Perspective) 투영 : 멀수록 작게 보이는 이유를 행렬로

- 핵심 직관

- 투영 평면까지의 거리 $-d$ 가 고정이면 $q_x = -dp_x/p_z, q_y = -dp_y/p_z$ 처럼 “ p_z 로 나누는” 형태가 된다
- 이 나눗셈이 동차 좌표(w)로 자연스럽게 구현됨

- 간단한 투영 (평면 $z = -d$)

$$\mathbf{P}_p = \begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & -1/d & 0 \end{pmatrix}$$



- 클립 후 (x, y, z, w) 에서 $w = z^0$ 이므로, 나누면 $q_x = -dp_x/p_z$
- 원근감은 동차화($\div w$)에서 생긴다

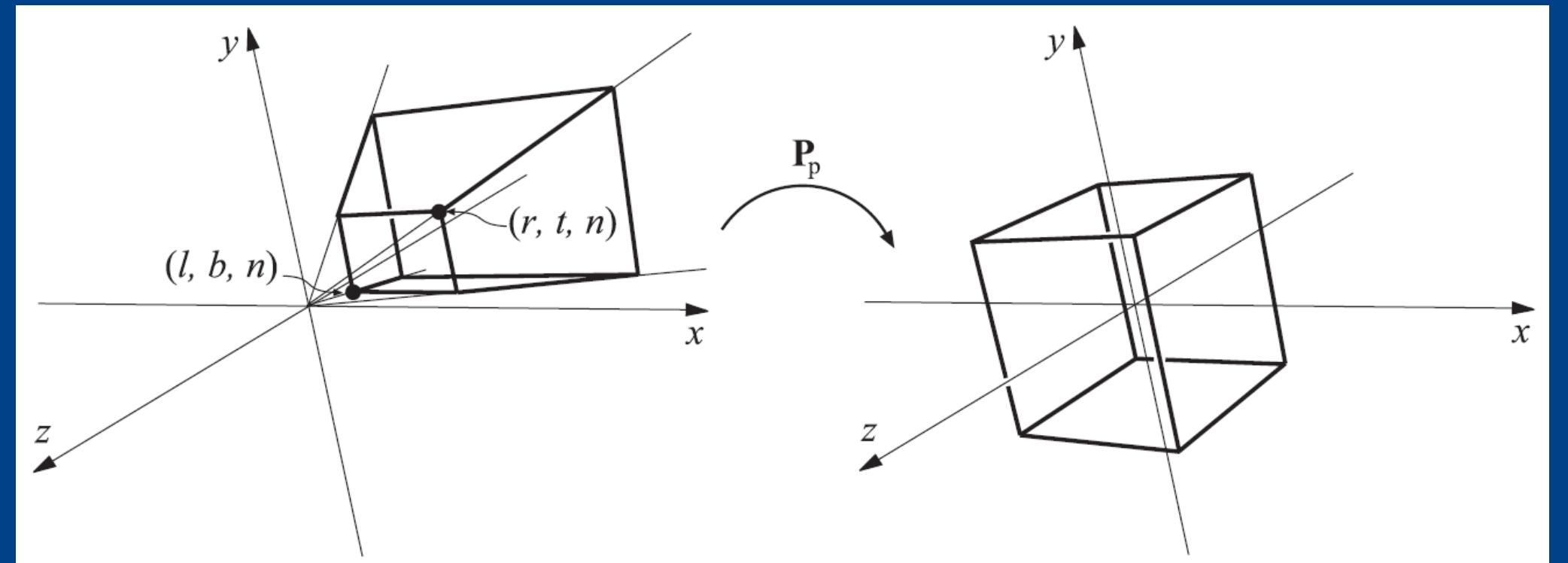
- 원근 투영 행렬의 목표 : 절두체 → NDC 정육면체

- 시각 절두체(Frustum)

- 카메라 시야각(FOV)과 화면비(Aspect), 근 / 원평면으로 정의
- 클리핑은 이 절두체 안쪽에서 수행

- 정규 시야 영역 (NDC)

- OpenGL 기준 $x, y, z \in [-1, 1]$ 인 정육면체로 정규화
- 정규화 후에는 단순 비교로 클리핑 가능 → 하드웨어에 유리



$$\mathbf{P}_{\text{OpenGL}} = \begin{pmatrix} \frac{2n'}{r-l} & 0 & \frac{r+l}{r-l} & 0 \\ 0 & \frac{2n'}{t-b} & \frac{t+b}{t-b} & 0 \\ 0 & 0 & -\frac{f'+n'}{f'-n'} & -\frac{2f'n'}{f'-n'} \\ 0 & 0 & -1 & 0 \end{pmatrix}$$

- 원근 투영 행렬의 목표 : 절두체 → NDC 정육면체
 - 정규 시야 영역 (NDC)
 - DirectX와 같은 일부 API들은 근평면을 $z = 0$ 으로 매핑하고, 원평면을 $z = 1$ 로 매핑한다
 - 또한 DirectX는 투영 행렬을 정의할 때 왼손 좌표계를 사용한다
이 말은 DirectX가 양의 z축 방향으로 바라보며, 근점값과 원점값을 양수로 표현함을 의미한다

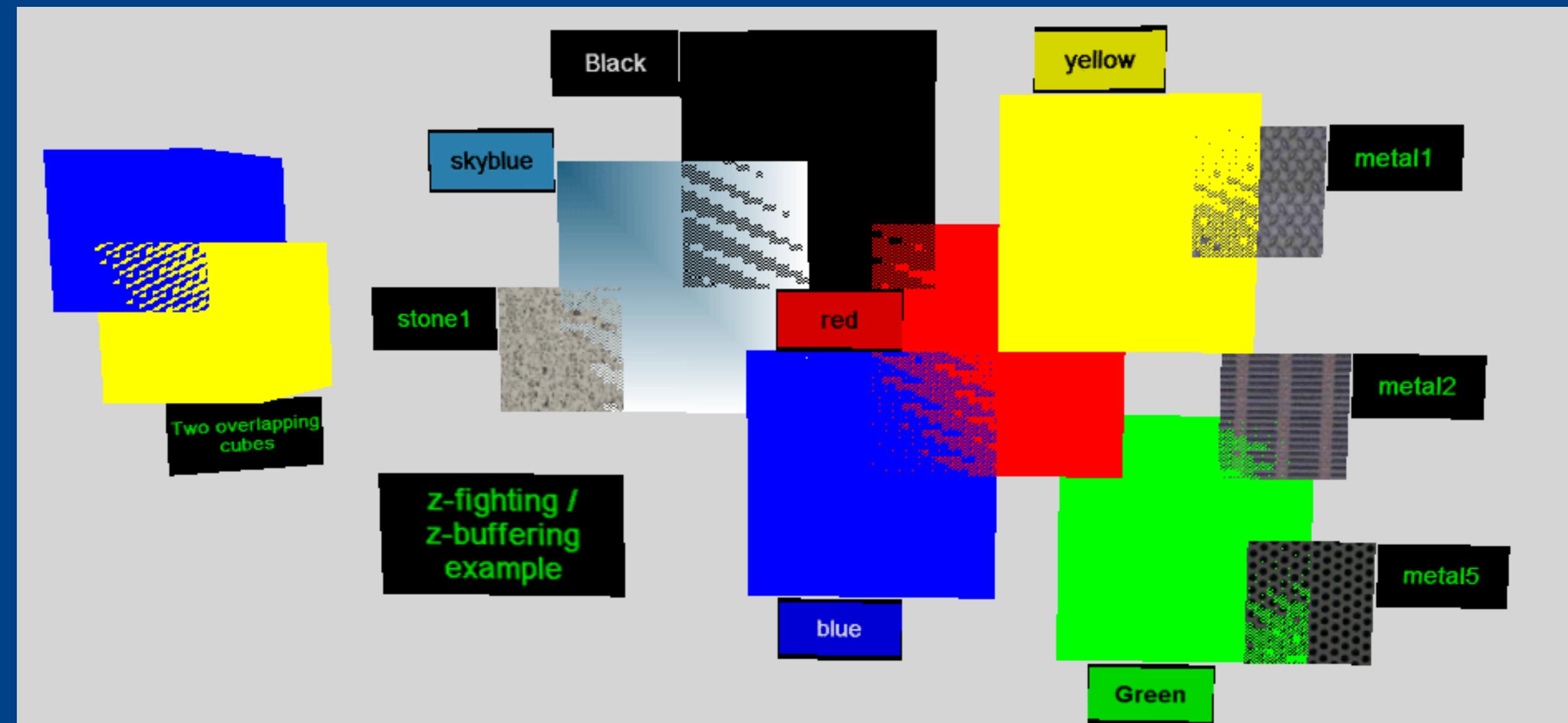
$$\mathbf{P}_{p[0,1]} = \begin{pmatrix} \frac{2n'}{r-l} & 0 & -\frac{r+l}{r-l} & 0 \\ 0 & \frac{2n'}{t-b} & -\frac{t+b}{t-b} & 0 \\ 0 & 0 & \frac{f'}{f'-n'} & -\frac{f'n'}{f'-n'} \\ 0 & 0 & 1 & 0 \end{pmatrix}$$

- FOV(시야각)은 “몰입감/왜곡”을 결정한다
 - 물리적 관계 (수평 시야각)
 - $\phi = 2 \arctan(w/(2d))$
 - ϕ 는 수평 시야, w 는 시선에 수직인 물체의 폭, d 는 물체까지의 거리
 - FOV가 넓을수록 더 많이 보이지만 왜곡도 커짐
 - 동일한 FOV라도 화면비에 따라 수평 / 수직 시야가 달라짐
 - 실전 팁
 - 너무 좁으면 “줌”처럼 평평해 보임
 - 너무 넓으면 모서리 왜곡(광각)과 가까운 물체가 과장됨
 - 플랫폼 / 거리에 맞게 조정이 중요

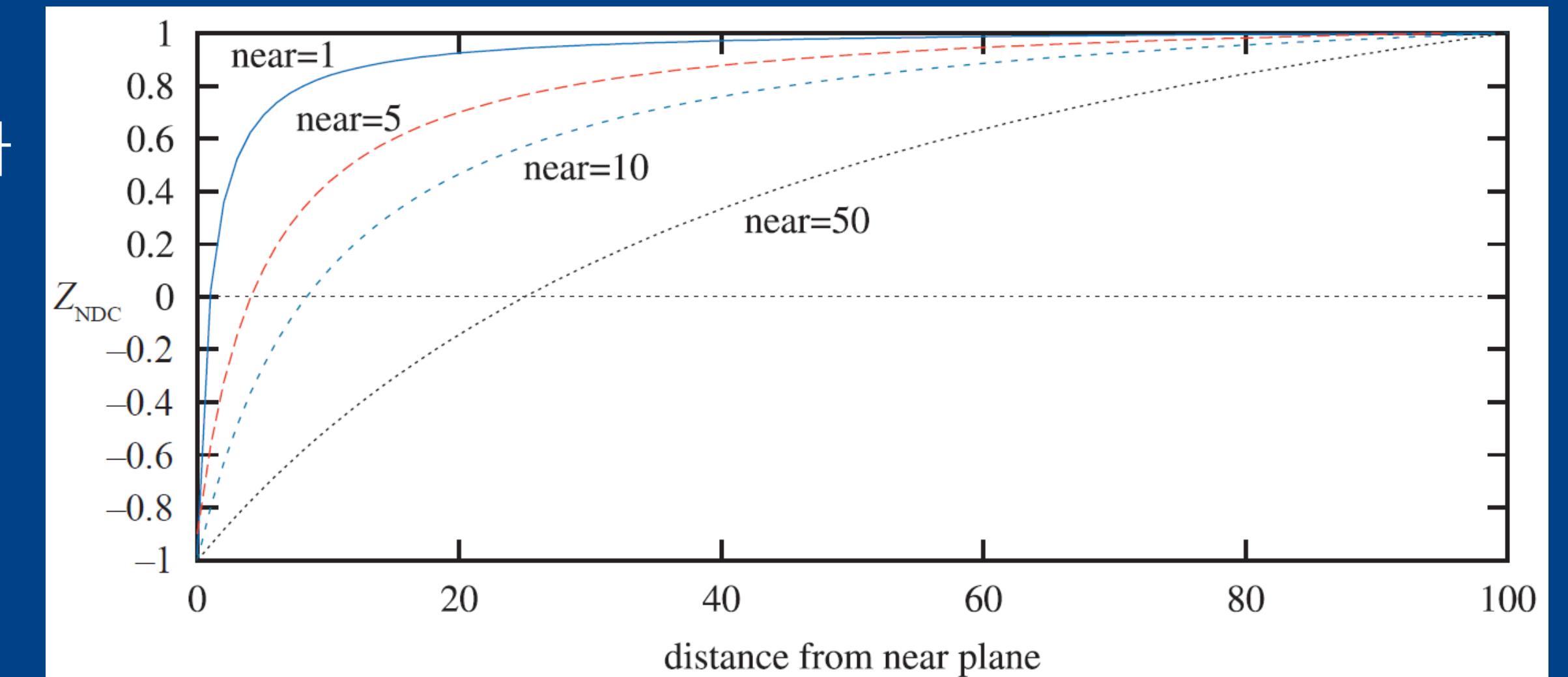
- 깊이 값은 선형이 아니다 : near/far가 정밀도를 좌우

- Z-fighting

- 움직이지 않는 2개 이상의 오브젝트가 출력 우선순위가 계속 변해서 깜빡이는 것처럼 보이는 현상
- 해당 위치 pixel의 Z-depth 값이 실수 연산의 오차 때문에 발생 (값이 거의 유사하거나 동일하거나)



- 깊이 값은 선형이 아니다 : near/far가 정밀도를 좌우
 - 왜 멀리서 Z-fighting이 생길까?
 - 원근 투영 후 깊이(z)는 거리와 선형 관계가 아니다
 - 가까운 near 근처에 정밀도가 몰리고, far 쪽은 압축된다
 - 권장사항
 - near를 너무 작게 두지 않기
 - far/near 비율을 줄이기
 - 필요하면 reversed-Z / 로그 깊이 등을 사용



감사합니다.

utilForever@gmail.com

<https://github.com/utilForever>

Facebook, Twitter: @utilForever