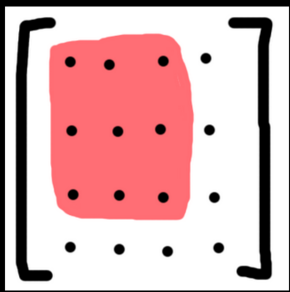
A 3×3 matrix or euler-angle would be expensive for interpolation

Matrix drift



A common homogeneous matrix uses its 3×3 part for rotation representation
the 3×3 would be exacted and scaled during rotation
then copy back

Error	Frames	Time (At 60 Hz)
0.000001	202	3 s
0.000002	437	7 s
0.000005	897	15 s
0.000010	1654	28 s
0.000020	3511	58 s
0.000050	8823	2 min
0.000100	14393	4 min
0.000200	24605	7 min
0.000500	52203	15 min
0.001000	100575	28 min

```
void set_rotation(Matrix4x4 &pose, const Quaternion &rot)
{
    Vector3 s = scale(pose);
    Matrix3x3 rotm = matrix3x3(rot);
    scale(rotm, s);
    set_3x3(pose, rotm);
}
```

This could change the scale of the obj due to float-point accuracy
The inaccuracy could grow over frames <linear growth>

How to fix

```
struct Pose {
    Vector3 translation;
    Matrix3x3 rotation;
    Vector3 scale;
};
```

Store rotation & scale separately
world-to-local transformation costs more
not practical when 4×4 solution is default

```
void set_rotation_and_scale(Matrix4x4 &pose, const Quaternion &rot, const Vector3 &s)
{
    Matrix3x3 rotm = matrix3x3(rot);
    scale(rotm, s);
    set_3x3(pose, rotm);
}
```

always set rotation / scale together
user is responsible for storing / recovering scales

quantize the scale value < the error cannot be big enough to change the scale >
user can still interpolate to make smooth animation.

Apply Gram-Schmidt algorithm

Hard for interpolation <matrix>

fix - use eular angles
→ gimbal lock

Quaternion

$v' = q v q^{-1}$ rotated vector $v = [0, v]$

ANGLE AND AXIS. Converting from angle and axis notation to quaternion notation involves two trigonometric operations, as well as several multiplies and divisions. It can be represented as

q = [cos(Q/2), sin(Q /2)v] (where Q is an angle and v is an axis)
(Eq. 4)

EULER ANGLES. Converting Euler angles into quaternions is a similar process - you just have to be careful that you perform the operations in the correct order. For example, let's say that a plane in a flight simulator first performs a yaw, then a pitch, and finally a roll. You can represent this combined quaternion rotation as

q = qyaw qpitch qroll where:
qroll = [cos (y/2), (sin(y/2), 0, 0)]
qpitch = [cos (q/2), (0, sin(q/2), 0)]
qyaw = [cos(f /2), (0, 0, sin(f /2)]
(Eq. 5)

The order in which you perform the multiplications is important. Quaternion multiplication is not commutative (due to the vector cross product that's involved). In other words, changing the order in which you rotate an object around various axes can produce different resulting orientations, and therefore, the order is important.

ROTATION MATRIX. Converting from a rotation matrix to a quaternion representation is a bit more involved, and its implementation can be seen in Listing 1.

Conversion between a unit quaternion and a rotation matrix can be specified as

| 1 - 2y^2 - 2z^2 2yz + 2wx 2xz - 2wy |
R$_m$ = | 2xy - 2wz 1 - 2x^2 - 2z^2 2yz - 2wx |
| 2xz + 2wy 2yz - 2wx 1 - 2x^2 - 2y^2|

(Eq. 6)

Table 1. Basic operations using quaternions.

Addition: q + q´ = [w + w´, v + v´]

Multiplication: qq´ = [ww´ - v · v´, v x v´ + wv´ +w´v] (· is vector dot product and x is vector cross product); Note: qq´ ? q´q

Conjugate: q* = [w, -v]

Norm: N(q) = w2 + x2 + y2 + z2

Inverse: q-1 = q* / N(q)

Unit Quaternion: q is a unit quaternion if N(q)= 1 and then q-1 = q*

Identity: [1, (0, 0, 0)] (when involving multiplication) and [0, (0, 0, 0)] (when involving addition)

for interpolation
convert quaternion to euler angle
the do rotation+1