Rotations

Wojciech Pachowiak

Table of Contents

Coordinate systems

What orientations do the following mathematical rotation representations express?

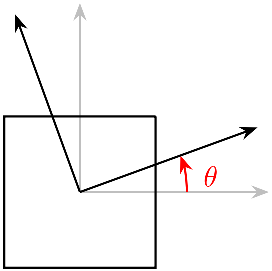
- Euler angles $(41^\circ, -18^\circ, -83^\circ)$ in XYZ extrinsic order
- Quaternion $(0.73, (0.16, -0.34, -0.57))$
- Axis angle tuple $(86^\circ, [0.24, -0.5, -0.84])$
- Axis angle vector $[0.36, -0.75, -1.26]$
- 3D rotation matrix

$$\begin{bmatrix} 0.116 & 0.725 & -0.68 \\ -0.944 & 0.3 & 0.15 \\ 0.31 & 0.624 & 0.718 \end{bmatrix}$$

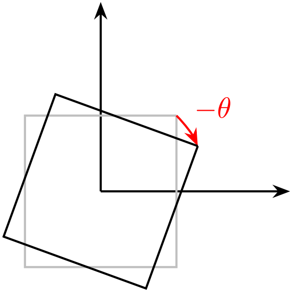
The same orientation but with respect to what coordinate system (frame of reference)?

Active and passive rotations

Which is moving? Objects within a coordinate system or the coordinate system itself?



(a) Passive rotation

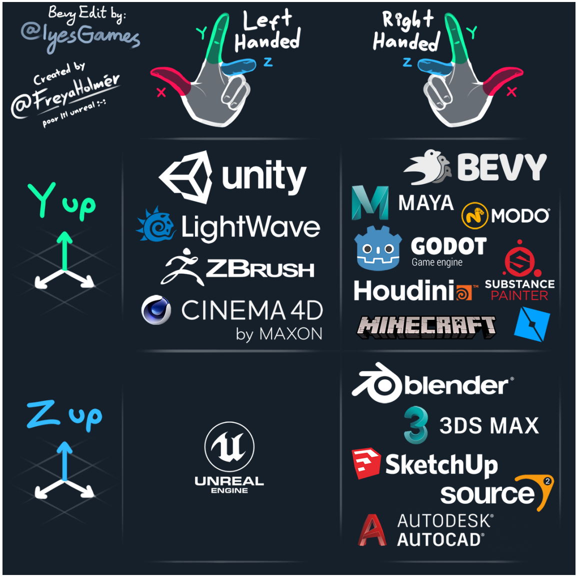


(b) Active rotation

To convert between these two perspectives, invert the rotation:

$$\mathbf{R}_{active} = \mathbf{R}_{passive}^{-1}$$

Handedness and up vector



https://bevy-cheatbook.github.io/fundamentals/coords.html

Forward, up, right vectors

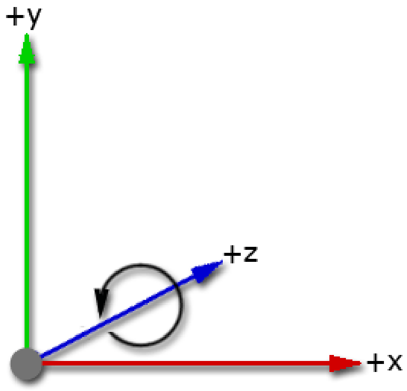
Which vectors are considered forward, upward, rightward, leftward, etc.?

This is pretty arbitrary: In 3D software it's pretty obvious which vector is considered the up vector. However, it's not obvious which vectors are the forward and right vectors.

Cameras, bones and regular geometry often have different coordinate systems from each other.

Positive angle

Is positive angle clockwise or counterclockwise when viewing from the tip of the arrow towards the origin?



`https://docs.unity3d.com/Manual/QuaternionAndEulerRotationsInUnity.html`

Intrinsic and extrinsic rotations

When it comes to Euler angles, are rotations performed around global/parent axes (**extrinsic** rotations) or some local/child axes (**intrinsic** rotations)?

See the pictures for a visual explanation:
`https://en.wikipedia.org/w/index.php?title=Davenport_chained_rotations&oldid=1222677779#Conversion_between_intrinsic_and_extrinsic_rotations`

Briefly stated, the order of axes gets reversed. For example, extrinsic XYZ order is the same thing as intrinsic ZYX order.

Matrix transposition

Are 4x4 transformation matrices (that represent the rotation, translation and scaling of a 3D object) written like this

$$\begin{bmatrix} \mathbf{RS} & \mathbf{t} \\ \mathbf{0} & 1 \end{bmatrix}$$

or like this

$$\begin{bmatrix} (\mathbf{RS})^\top & \mathbf{0}^\top \\ \mathbf{t}^\top & 1 \end{bmatrix}$$

where $\mathbf{R}$ is a 3x3 rotation matrix, $\mathbf{S}$ is a 3x3 scaling matrix, $\mathbf{t}$ is a 3x1 translation vector, and $\mathbf{0}$ is a 3x1 zero vector? In other words, are the matrices transposed or not?

This choice determines how vectors are transformed by matrices: they are either premultiplied ($\mathbf{v}' = \mathbf{R}\mathbf{v}$) or postmultiplied ($(\mathbf{v}')^\top = \mathbf{v}^\top \mathbf{R}^\top$).

Matrix transposition

$$\begin{bmatrix} \begin{bmatrix} - & - & - \\ - & - & - \\ - & - & - \end{bmatrix} & \begin{bmatrix} t_x \\ t_y \\ t_z \end{bmatrix} \\ \begin{bmatrix} 0 & 0 & 0 \end{bmatrix} & 1 \end{bmatrix} = \begin{bmatrix} R & t \\ 0 & 1 \end{bmatrix}$$

$$\Big\downarrow \ \textbf{TRANSPOSITION}$$

$$\begin{bmatrix} \begin{bmatrix} | & | & | \\ | & | & | \\ | & | & | \end{bmatrix} & \begin{bmatrix} 0 \\ 0 \\ 0 \end{bmatrix} \\ \begin{bmatrix} t_x & t_y & t_z \end{bmatrix} & 1 \end{bmatrix} = \begin{bmatrix} R^T & 0^T \\ t^T & 1 \end{bmatrix}$$

Table of Contents

2D rotations

- 2D (2x2) rotation matrices

$$\mathbf{v}' = \begin{bmatrix} \cos\theta & -\sin\theta \\ \sin\theta & \cos\theta \end{bmatrix} \mathbf{v}$$

  where θ is the angle of rotation, $\mathbf{v}$ is a vector before rotation, and $\mathbf{v}'$ is the rotated vector.

- complex numbers

$$(x' + iy') = e^{i\theta}(x + iy) = (\cos\theta + i\sin\theta)(x + iy).$$

  where θ is the angle of rotation, x and y are the coordinates of a 2D vector before rotation, and x' and y' are the rotated coordinates.

Contrary to 3D rotations, 2D rotations are **commutative**!

Euler angles

3-tuple of angles (usally expressed in degrees, not radians) that express a sequence of 3 rotations around 3 perpendicular axes.

While not common in Compute Graphics, the first and third axes can be the same as in ZYZ, ZXZ, XYX, XZX, YZY, and YXY orders.

The order is important: ZYX is not equivalent to YZX. As mentioned before, intrinsic XYZ order is not the equivalent to extrinsic XYZ order.

All in all, there are 24 possible types of Euler angles.

Euler angles $+/-$

Upsides:

- Intuitive when expressed relative to world/global coordinate system.
- Good for GUIs—only 3 numbers with no constraints (like unit-length or orthonormality constraints).
- Immune to gimbal lock, when only manipulating 2 angles.

Downsides:

- Hard to compose (element-wise addition produces incorrect result).
- Very ambiguous (one rotation corresponds to many Euler angles, even of the same order).
- Hard to visualize.
- Experience gimbal lock.
- Not self-reliant—must be converted to other representations often.
- Without explicitly stating the order, bare 3-tuple is meaningless.

3D rotation matrices: X,Y,Z rotations

To rotate around the X, Y, and Z axes, use the following matrices:

$$\mathbf{R}(\mathbf{X},\theta) = \begin{bmatrix} 1 & 0 & 0 \\ 0 & \cos\theta & -\sin\theta \\ 0 & \sin\theta & \cos\theta \end{bmatrix}$$

$$\mathbf{R}(\mathbf{Y},\theta) = \begin{bmatrix} \cos\theta & 0 & \sin\theta \\ 0 & 1 & 0 \\ -\sin\theta & 0 & \cos\theta \end{bmatrix}$$

$$\mathbf{R}(\mathbf{Z},\theta) = \begin{bmatrix} \cos\theta & -\sin\theta & 0 \\ \sin\theta & \cos\theta & 0 \\ 0 & 0 & 1 \end{bmatrix}$$

where θ is a rotation angle. Sometimes you might see them transposed (with the minus sign on the opposite side of the diagonal).

3D rotation matrices: Euler-angles-like

Extrinsic XYZ Euler angles can be represented as a rotation matrix by subsequently multiplying[1] the three rotation matrices for X, Y, and Z axes, in that order:

$$\mathbf{R}(\mathbf{Z},\alpha)\mathbf{R}(\mathbf{Y},\beta)\mathbf{R}(\mathbf{X},\theta) = \begin{bmatrix} c_\alpha c_\beta & c_\alpha s_\beta s_\theta - c_\theta s_\alpha & s_\alpha s_\theta + c_\alpha c_\theta s_\beta \\ c_\beta s_\alpha & c_\alpha c_\theta + s_\alpha s_\beta s_\theta & c_\theta s_\alpha s_\beta - c_\alpha s_\theta \\ -s_\beta & c_\beta s_\theta & c_\beta c_\theta \end{bmatrix}$$

where $c_? = \cos(?)$ and $s_? = \sin(?)$. Angles are in radians, not degrees.

There are 12 such Euler-angles-like rotation matrices. Why not 24? Because the extrinsic XYZ matrix is equivalent to the instrinsic ZYX matrix, etc.

Rotations performed by such matrices are still prone to gimbal lock!

[1]Search for "matrix multiplication" in Google Images.

3D rotation matrices: axis-angle-like

A matrix that rotates θ radians around a vector $\mathbf{n}$ is as follows:

$$\mathbf{R}(\mathbf{n},\theta)= \begin{bmatrix} c+(n_1)^2(1-c) & n_1 n_2(1-c)-sn_3 & n_1 n_3(1-c)+sn_2 \\ n_2 n_1(1-c)+sn_3 & c+(n_2)^2(1-c) & n_2 n_3(1-c)-sn_1 \\ n_3 n_1(1-c)-sn_2 & n_3 n_2(1-c)+sn_1 & c+(n_3)^2(1-c) \end{bmatrix}$$

where $c=\cos\theta$ and $s=\sin\theta$. It can be verified that $\mathbf{R}(\mathbf{n},\theta)$ is indeed a rotation matrix around $\mathbf{n}$ by confirming that

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

Note that $\mathbf{R}(\mathbf{n},\theta)=\mathbf{R}(-\mathbf{n},-\theta)$.

Rotations using this matrix are immune to gimbal lock!

3D rotation matrices: visualization

Assuming that all vectors are represented as column vectors, then 3D rotation matrices are simply containers for the X, Y and Z axes

$$\begin{bmatrix} | & | & | \\ \mathbf{X} & \mathbf{Y} & \mathbf{Z} \\ | & | & | \end{bmatrix}.$$

This is extremely useful for visualization and debugging.

3D rotation matrices: Rodriguez formula

Without any effort, $\mathbf{R}(\mathbf{n}, \theta)$ can be decomposed into the following sum:

$$\mathbf{R}(\mathbf{n}, \theta) = (\cos\theta)\mathbf{I} + (\sin\theta)\,[\mathbf{n}]_\times + (1 - \cos\theta)(\mathbf{n}\mathbf{n}^\top)$$

where $[\mathbf{n}]_\times$ is the cross product matrix of $\mathbf{n}$:

$$[\mathbf{n}]_\times = \begin{bmatrix} 0 & -n_3 & n_2 \\ n_3 & 0 & -n_1 \\ -n_2 & n_1 & 0 \end{bmatrix},$$

and $\mathbf{n}\mathbf{n}^\top$ is as follows:

$$\mathbf{n}\mathbf{n}^\top = \begin{bmatrix} n_1^2 & n_1 n_2 & n_1 n_3 \\ n_2 n_1 & n_2^2 & n_2 n_3 \\ n_3 n_1 & n_3 n_2 & n_3^2. \end{bmatrix}$$

This formula is known as the *Euler-Rodriguez formula* for rotation matrices. Equivalently, it can be reformulated as:

$$\mathbf{R}(\mathbf{n}, \theta) = I + (sin\theta)\,[\mathbf{n}]_\times + (1 - \cos\theta)\,[\mathbf{n}]_\times^2 .$$

3D rotation matrices $+/-$ (SPOILER: they are the best)

Upsides:

- Easy to visualize!
- Immune to gimbal lock[2].
- Non-ambiguous: one rotation corresponds to exactly one matrix.
- Probably benefit from CPU and GPU optimizations for matrix multiplication (speculating).

Downsides:

- Need to store 9 numbers.
- Need to reorthogonalize to prevent numerical errors from accumulating (speculating).

[2]When not performing sequential rotations around mutually prpendicular axes (like in Euler angles)!

Unit-length quaternions

Don't try to visualize them in 3D space. Just learn their algebraic rules: how to multiply them, how to invert them, how to normalize them, etc. Also, learn how to extract axis and angle from them.

Not every quaternion represents a rotation: only the unit-length ones!

Quaternion q can be represented as a sum of the scalar part and the imaginary parts or as a tuple consisting of a scalar and a vector:

$$q = q_0 + iq_1 + jq_2 + kq_3 = (q_0, \mathbf{q}) = (q_0, q_1, q_2, q_3) = (q_0, (q_1, q_2, q_3)),$$

where q_0 is the scalar part and $\mathbf{q} = (q_1, q_2, q_3)$ is the vector part. Often, q_0 is called w and (q_1, q_2, q_3) is called (x, y, z).

Unit-length quaternions: rotations

The four components of a unit quaternion can also be represented as

$$q = (\cos\tfrac{\theta}{2}, \mathbf{n}\sin\tfrac{\theta}{2}),$$

where $\mathbf{n}$ is the axis of rotation and θ its angle in radians ($\frac{\theta}{2}$ is not the angle of rotation!).
To rotate a quaternion q_1 by a quaternion q_2, simply multiply them to obtain some quaternion q_3:

$$q_3 = q_2 * q_1$$

To rotate a 3D vector $\mathbf{v}$ by a quaternion q, treat $\mathbf{v}$ as a quaternion with scalar part equal to 0 (such quaternions are called *pure quaternions*), and multiply it from both sides:

$$(0, \mathbf{v}') = q * (0, \mathbf{v}) * q^{-1}$$

The right hand side can be expanded in such a way as to reveal the Euler-Rodriguez formula.

Unit-length quaternions $+/-$

Upsides:

- Need to store only 4 numbers.
- Immune to gimbal lock[3].
- Allow rotating around an arbitrary axis.
- Straightforward to derive *SLERP* and *NLERP*.
- Normalizing quaternions after each multiplication prevents numerical errors from accummulating (speculating).

Downside:

- Ambiguous: quaternion q represents the same rotation as $-q$.
- Hard to visualize.
- Seem scary to non-maths people.
- Store only rotation whereas 4x4 transformation matrices can additionally store translation and scaling.

[3]When not performing sequential rotations around mutually prpendicular axes (like in Euler angles)!

Axis-angle

Probably the simplest way to represent a rotation around some axis **n** and an angle θ is using either a tuple

$$\left(\begin{bmatrix} n_1 \\ n_2 \\ n_3 \end{bmatrix}, \theta \right)$$

or a vector

$$\theta \begin{bmatrix} n_1 \\ n_2 \\ n_3 \end{bmatrix}.$$

For example, a 45° rotation around an axis halfway between the **X** and **Z** axes is as follows:

$$\frac{\pi}{4} \begin{bmatrix} \frac{1}{\sqrt{2}} \\ 0 \\ \frac{1}{\sqrt{2}} \end{bmatrix}$$

Axis-angle: exponential map

Axis-angle vectors and tuples can't be composed on their own: a sum or a dot product[4] produce incorrect results. To address this, we define functions exp and log.

exp converts axis-angle vectors to quaternions or rotation matrices. log converts quaternions or rotation matrices back to axis angle vectors.

They are useful for calculating derivatives[5] of functions involving rotations. There is a deep theory behind these functions, but it's not something 3D animators need to know.

[4] Search for "vector dot product" in Google.

[5] The derivative of a function measures how the function is changing (how fast, in which direction, etc.)

Axis-angle $+/-$

Upsides:

- Intuitive.
- Only 3 numbers to store.
- Cool for calculating derivatives (tbh, I don't know much about it).
- exp and log map make it easy to generalize *SLERP* to rotation matrices.

Downsides:

- Not self-reliant—must be converted to other representations often.
- $\mathbf{n}\theta$ express the same rotation as $(-\mathbf{n})(-\theta)$.
- Maybe they are necessary for some things, but I'm not advanced enough to know about it and fully appreciate it. (haha)

In SciPy

`https://docs.scipy.org/doc/scipy/reference/generated/scipy.spatial.transform.Rotation.as_quat.html`

Note the *seq* argument in *as_euler* method. Also note the *canonical* and *scalar_first* arguments in *as_quat*.

We didn't cover:

- Davenport angles — like Euler angles but generalized to nonperpendicular axes.
- Modified Rodriguez Parameters (MRP) — like axis-angle vectors but the angle of rotation is transformed.

Table of Contents

Gimbal lock

Tbh, I don't know the full theory behind gimbal lock. Most of the sources seem to only give examples of when gimbal lock happens. Don't take my word for what I'm about to say.

Basically, for some special values of Euler angles, two angles rotate around the same axis.

For the 3 perpencidular axes Euler angles (XYZ, ZXY, YZX, etc.), it happens when the middle angle is set to $\pm 90^\circ$.

For the 2 perpendicular axes Euler angles (XYX, ZYZ, YZY, etc.), it happens when the middle angle is set to 0° or $\pm 180^\circ$.

In gimbal lock, small changes of axis-angle parametrized functions result in huge changes in Euler angles.

A-to-B rotation interpolation: SLERP

You provide two keyframes at A and B; the inbetween frames are interpolated by the 3D software. Let's focus on quaternions.

SLERP (*spherical linear interpolation*) is the proper way to interpolate two quaternions: the "movement" it produces has constant angular velocity, and follows the shortest path.

$$SLERP(q_0, q_1, t) = \frac{\sin((1-t)\theta)}{\sin(\theta)} q_0 + \frac{\sin(t\theta)}{\sin(\theta)} q_1$$

Surprisingly, $SLERP(q_0, q_1, t)$ and $SLERP(-q_0, q_1, t)$ produce different results.

A-to-B rotation interpolation: NLERP

NLERP (*normalized linear interpolation*) is faster and easier to compute. It follows the same path as *SLERP*, but the angular velocity is not constant[6]. Basically, it computes *LERP* on two quaternions and normalizes the result so that the unit-length constraint is satisfied.
Simple *LERP* is defined as follows:

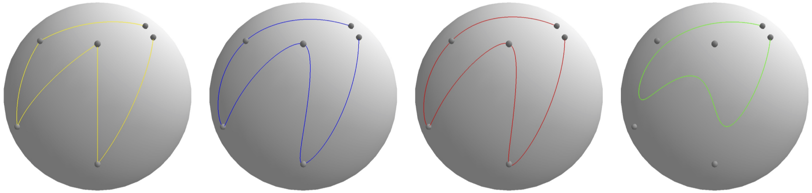
$$\begin{aligned} LERP(q,p,t) = (&q_0(1-t)+p_0t, \\ (&q_1(1-t)+p_1t, \\ &q_2(1-t)+p_2t, \\ &q_3(1-t)+p_3t)), \end{aligned}$$

where $t \in [0,1]$ and q and p are some unit-length quaternions.
Then, *NLERP* is defined as

$$NLERP(q,p,t) = \frac{LERP(q,p,t)}{\|LERP(q,p,t)\|}.$$

[6]But it's not bad. Blender uses *NLERP* by default for rotation interpolation.

A-to-C-through-B rotation interpolation



`https://www.adrian-haarbach.de/interpolation-methods/`