Math for 3D/Games Programmers

7. Quaternions

Table of Contents

What is a Quaternion

- **Quaternion** is an extension of complex number from 2D into 4D (yeah, 4D)
- Complex number is defined like so: $z = a + bi \qquad i^2 = -1$
- Quaternion is defined like so:

$$q = w + x\boldsymbol{i} + y\boldsymbol{j} + z\boldsymbol{k} \qquad q = (w, x, y, z)$$

$$\begin{aligned}
&w, x, y, z \in R \\
&i^2 = -1 \qquad && j^2 = -1 \qquad && k^2 = -1, \\
&ij = -ji = k \qquad && jk = -kj = i \qquad && ki = -ik = j
\end{aligned}$$

- The value w is the **scalar part**, whereas $[x, y, z]$ is the **vector part**
- Wikipedia

What is a Quaternion

- Complex number is a 2D object and enables 2D rotations
- Quaternion is a 4D object and enables 3D rotations…
  Of course we can also represent orientation with a quaternion (just like with a matrix)
- For long it was believed that rotations in 3D can be represented with:

  $$a + bi + cj$$

  It turned out to be impossible. Eventually sir William R. Hamilton proved that four numbers are required to represent rotation in 3D with a complex number
- For many reasons (which will be discussed later) Unity, as well as many other 3D engines, use quaternions for internal representation of rotation/orientation

What is a Quaternion

- **Modulus** or **length** of a quaternion can be calculated with a formula similar to that from complex numbers:

$$|q| = \sqrt{w^2 + x^2 + y^2 + z^2}$$

- **Dot product** of two quaternions is calculated like so:

$$q_1 = (w_1, x_1, y_1, z_1) \qquad q_2 = (w_2, x_2, y_2, z_2)$$

$$q_1 \circ q_2 = w_1 w_2 + x_1 x_2 + y_1 y_2 + z_1 z_2$$

- Therefore:

$$|q| = \sqrt{q \circ q}$$

What is a Quaternion

- In practice to perform rotations in 3D we need **unit quaternions**, which satisfy:

$$|q| = 1$$

- We can think of unit quaternions as of points belonging to a 4D sphere with the radius $r = 1$. Remember that such a quaternion/"4D vector" represents an **entire 3D basis**
- A non-unit quaternion can be **normalized**:

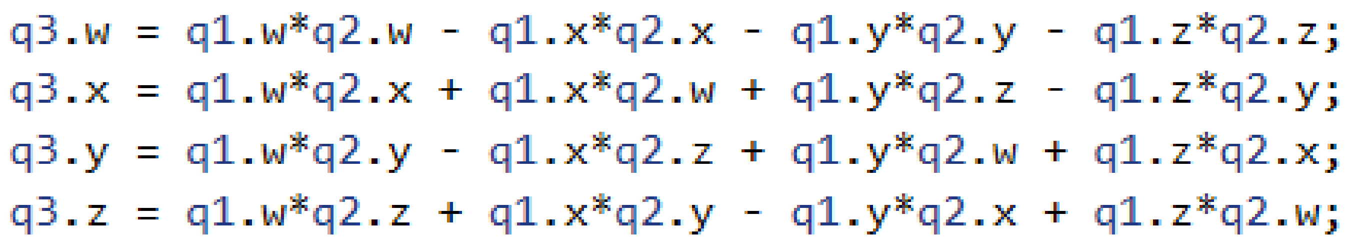
$$\frac{q}{|q|}$$

What is a Quaternion

- **Product (multiplication)** of two quaternions is calculated similarly to how we multiply two complex numbers:

$$q_1 = (w_1, x_1, y_1, z_1\ ) \qquad\qquad q_2 = (w_2, x_2, y_2, z_2)$$

$$q_1 q_2 = (w_1 + x_1 i + y_1 j + z_1 k)(w_2 + x_2 i + y_2 j + z_2 k)$$



```
q3.w = q1.w*q2.w - q1.x*q2.x - q1.y*q2.y - q1.z*q2.z;
q3.x = q1.w*q2.x + q1.x*q2.w + q1.y*q2.z - q1.z*q2.y;
q3.y = q1.w*q2.y - q1.x*q2.z + q1.y*q2.w + q1.z*q2.x;
q3.z = q1.w*q2.z + q1.x*q2.y - q1.y*q2.x + q1.z*q2.w;
```

What is a Quaternion

- One unit quaternion q can represent a rotation/orientation in 3D, just like a single matrix M of size 3x3
- When we have two **matrices** M_1 and M_2 representing two rotations, their product $M_2 M_1$ represents a transformation which first applies the rotation from the matrix M_1, followed by the rotation from the matrix M_2
- It is the same for **quaternions**. Let's say we have two quaternions q_1 and q_2. Their product $q_2 q_1$ results in a transformation where first the rotation from q_1 is applied, followed by the rotation from q_2
- Multiplication of quaternions, as opposed to complex numbers, **is not commutative**:

$$q_1 q_2 \neq q_2 q_1$$

What is a Quaternion

- Given a point p and a **matrix** M we transform p by calculating:

$$p' = Mp$$

- For **quaternion** the formula is different. Given a quaternion q the transformed/rotated point p' is calculated as:

$$p' = q * \left(0, p_x, p_y, p_z\right) * q^{-1}$$

where $\left(0, p_x, p_y, p_z\right)$ is a quaternion such that $w = 0$ and $[x, y, z] = \left[p_x, p_y, p_z\right]$

What is a Quaternion

- To rotate a point we need the formula for the quaternion's **inverse**:

$$q^{-1} = \frac{q'}{|q|^2} = \frac{q'}{q \circ q}$$

- This formula requires the **conjugate** q', which is calculated as:

$$q' = w - xi - yj - zk$$

- The above formulas are identical to those for complex numbers

What is a Quaternion

- When $|q| = 1$ then:

$$q^{-1} = q'$$

- What leads to a simpler formula for point rotation:

$$p' = q * \left(0, p_x, p_y, p_z\right) * q'$$

What is a Quaternion

- **Identity quaternion** is a quaternion which serves as 1 in the set of quaternion numbers:

  $$q = (1,0,0,0)$$

  In the context of transformations it behaves just like the identity matrix
- Surprisingly, if we have a quaternion q then quaternion $-q$ represents the same orientation. They will both affect interpolation differently though
- If we notice that an object’s rotation/orientation at some point starts to look „odd”, it is very much likely that the quaternion representing the object’s rotation/orientation is no more of length 1 and needs to be normalized

Rotation About Any Axis

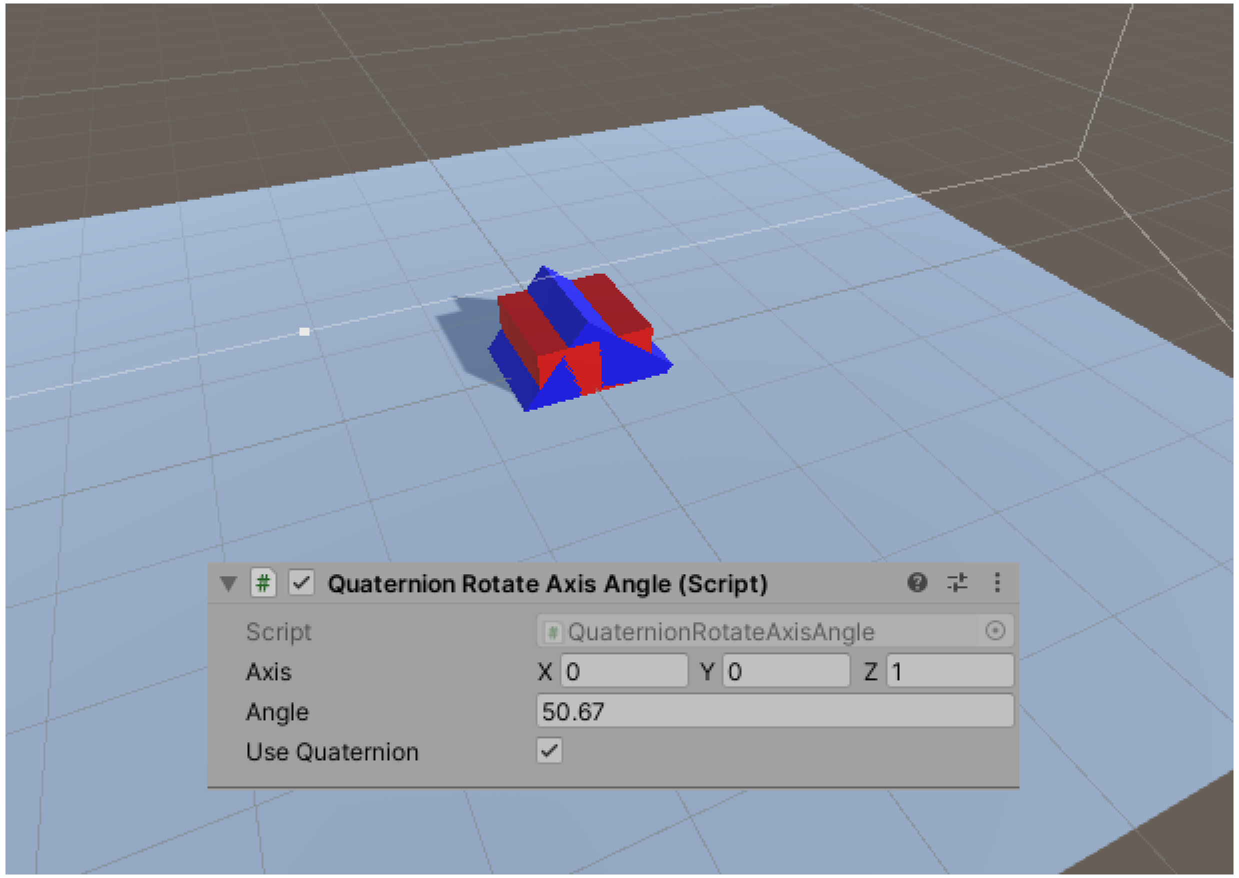
- Rotation about any axis is described by four values: the rotation axis $[x, y, z]$ and the rotation angle θ
- Creation of a quaternion which represents such a rotation is not at all as easy us just plugging those values right into the quaternion:

$$q = (\theta, x, y, z) \qquad \text{NOPE!}$$

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

- Wikipedia

Rotation About Any Axis


Quaternion Rotate Axis Angle (Script)

Script QuaternionRotateAxisAngle

Axis X 0 Y 0 Z 1

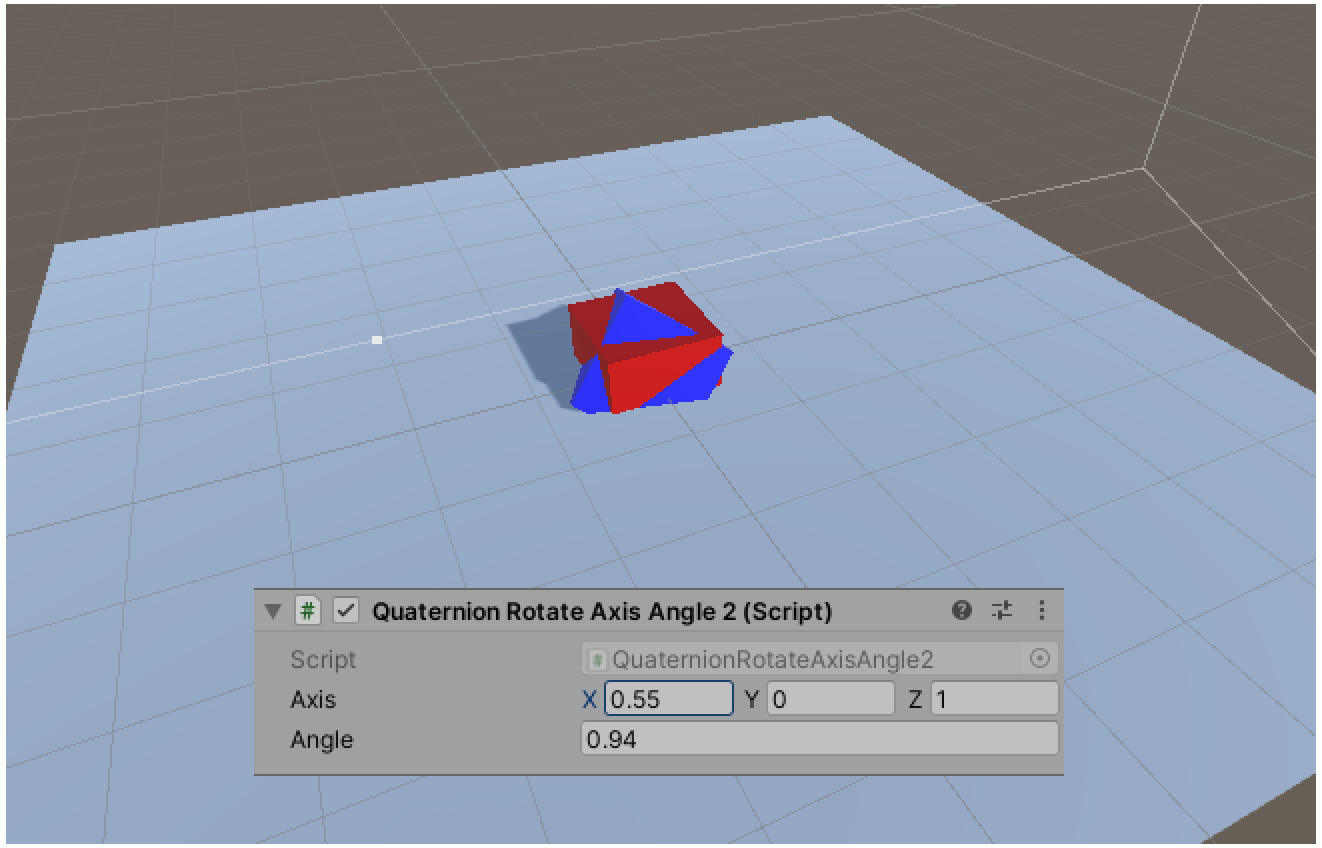
Angle 50.67

Use Quaternion ✓

Rotation About Any Axis

- Most of the functionality that pertains to quarternions is readily available in Unity and is worth using
- We will however implement the same any axis rotation using the formulas which we learned in the previous section

Rotation About Any Axis



Quaternion Rotate Axis Angle 2 (Script)

Script QuaternionRotateAxisAngle2

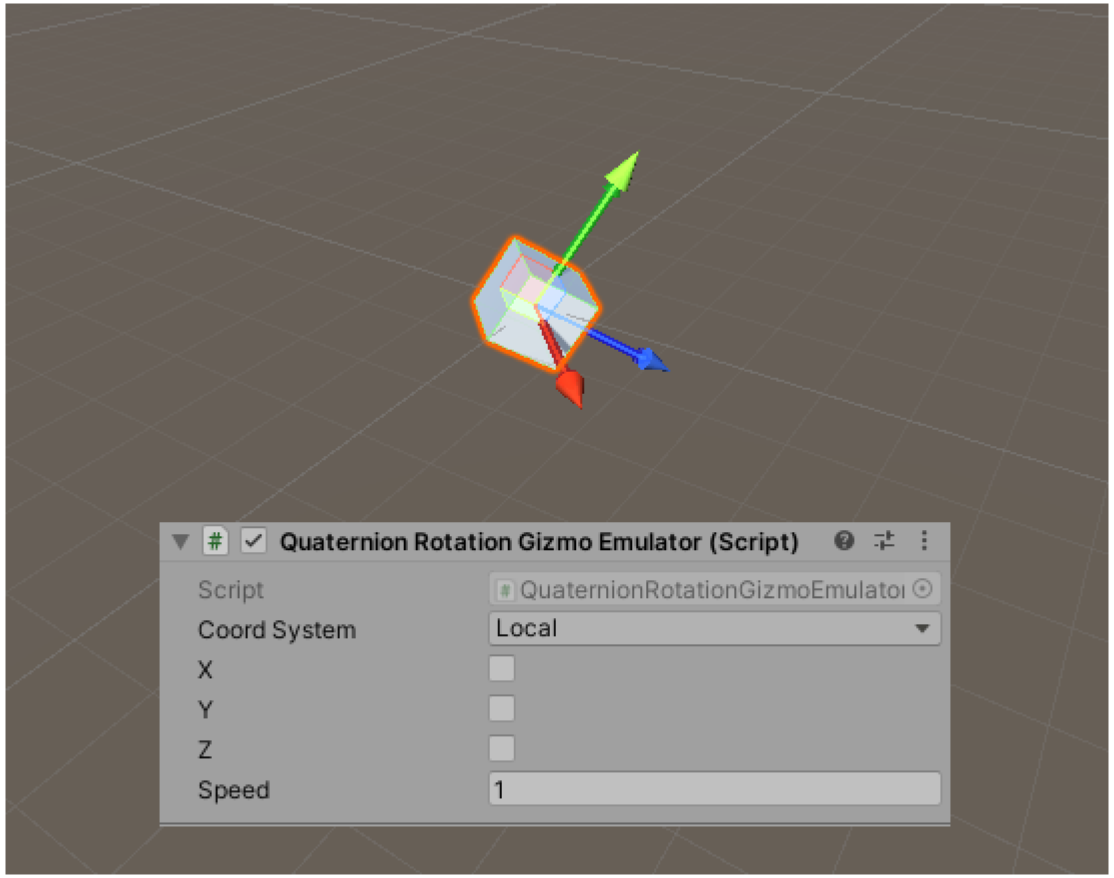
Axis X 0.55 Y 0 Z 1

Angle 0.94

Rotation About Any Axis

- In the last chapter, using matrices, we created the rotation gizmo emulator (RotationGizmoEmulator)
- Knowing how to create an any axis rotation quaternion we can write a similar program which emulates the gizmo, but this time around using a quaternion
- While on it we will also learn how to extract the local space from a quaternion

Rotation About Any Axis



Quaternion Rotation Gizmo Emulator (Script)

Script	QuaternionRotationGizmoEmulator
Coord System	Local
X	☐
Y	☐
Z	☐
Speed	1

Rotation About Any Axis

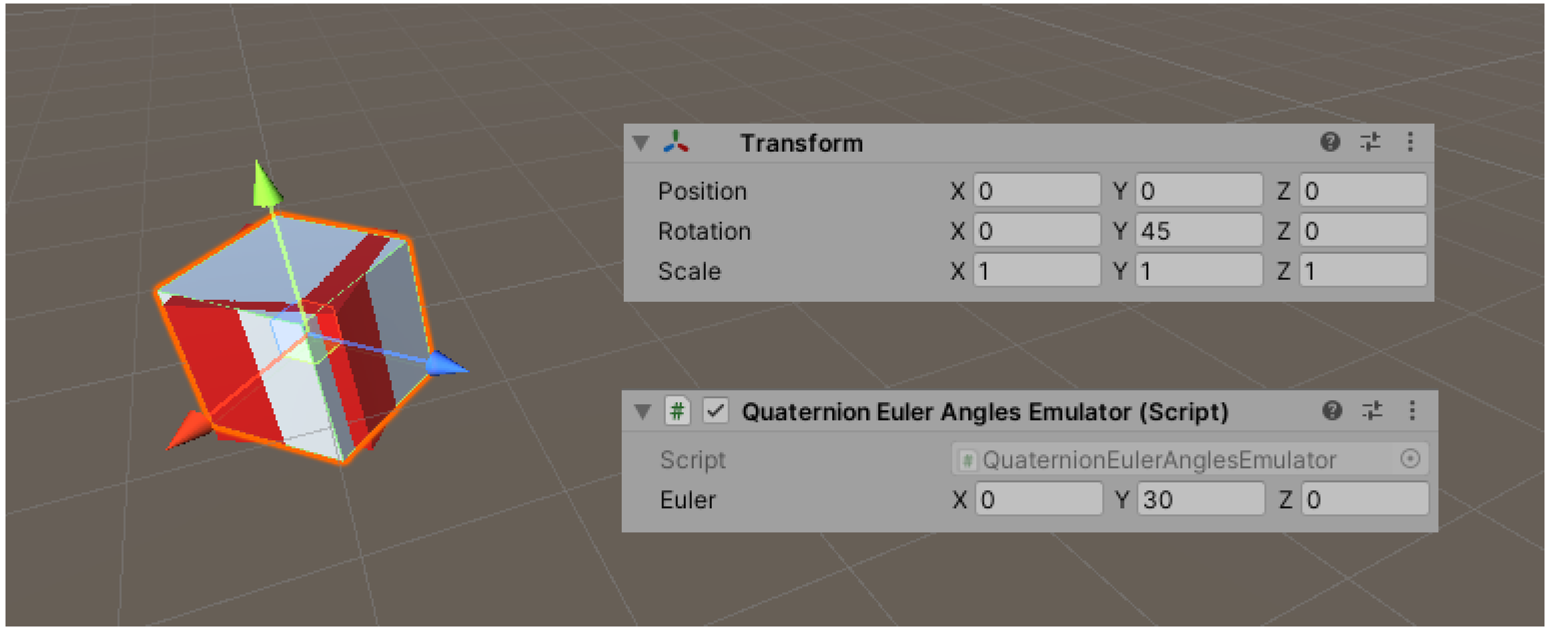
- In the last chapter we said that the *Transform* component, from the source code level, has access to the fields `localPosition` / `localEulerAngles` / `localScale`
- In reality `localEulerAngles` is a short for `localRotation.eulerAngles`
- Internally, Unity – for each *Transform* component – stores position as `Vector3`, scale as `Vector3` and orientation as `Quaternion`
- Euler angles serve only as an auxiliary since they are the most convenient to work with for humans

Quaternion, Matrix and Euler Angles

- At this point we might be a little confused when it comes to 3D rotations
- We’ve said a lot before about matrices and Euler angles, and now quaternions came into play
- The core notion: each of those forms allows us to represent rotation/orientation in 3D
- In a moment we will explain how and why these three concepts work together

- But before that we will see the Euler angles emulator using quaternions

Quaternion, Matrix and Euler Angles



Transform

Position	X 0	Y 0	Z 0
Rotation	X 0	Y 45	Z 0
Scale	X 1	Y 1	Z 1

Quaternion Euler Angles Emulator (Script)

Script	QuaternionEulerAnglesEmulator		
Euler	X 0	Y 30	Z 0

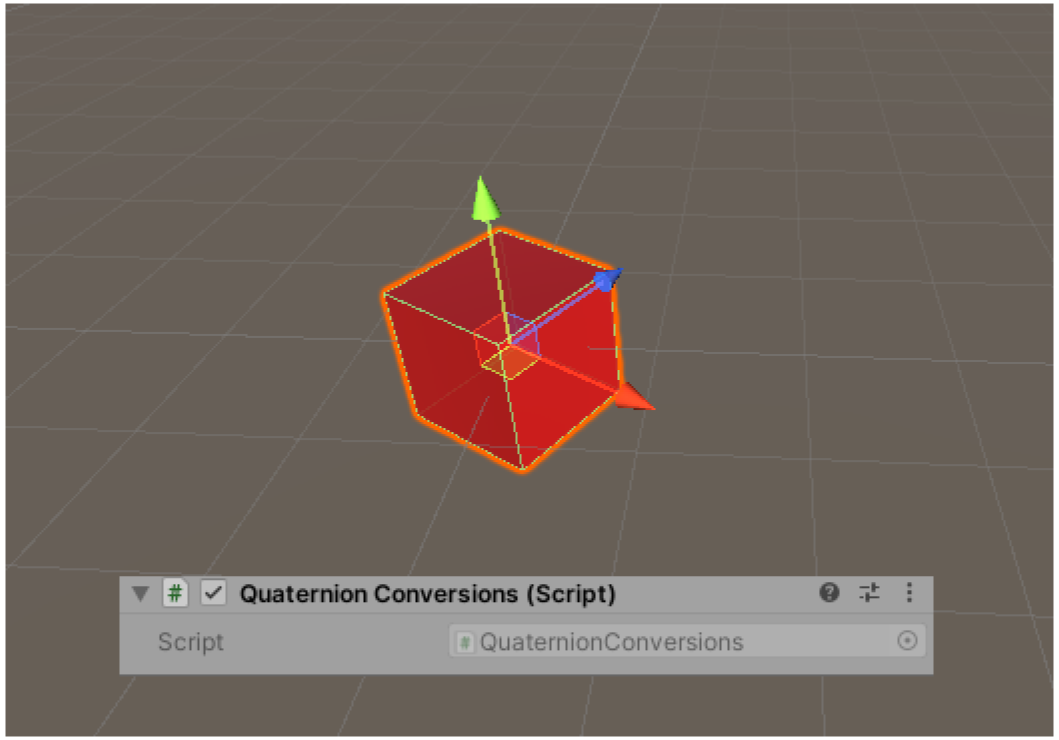
Quaternion, Matrix and Euler Angles

- Note that for the actual rotation calculations we use either a matrix or a quaternion
- Euler angles „don’t live in a vacuum” and they are just a helper interface when creating a rotation out of three independent ZXY rotations (all of which are constructed using matrices or quaternions)
- The idea of Euler angles exists because it is by far the easiest way of working with rotations for humans. Easier than filling up a matrix or a quaternion by hand
- The generic rotation about the given axis and angle (which for example is used by the rotation gizmo tool) is usually implemented either via a matrix or quaternion

Quaternion, Matrix and Euler Angles

- Internally Unity does not use Euler angles anywhere
- A scene file written to disk stores rotation in quaternion form (*Transform* component stores rotation in `Quaternion` type)
- In general on the gameplay/scripts/simulation/physics side we usually work with quaternions
- On the rendering side we usually work with matrices

Quaternion, Matrix and Euler Angles



Quaternion, Matrix and Euler Angles

- We know from the last chapter, from the sub-chapter about bases, that in order to go from one base to another we can use the formula for the „difference” of bases:

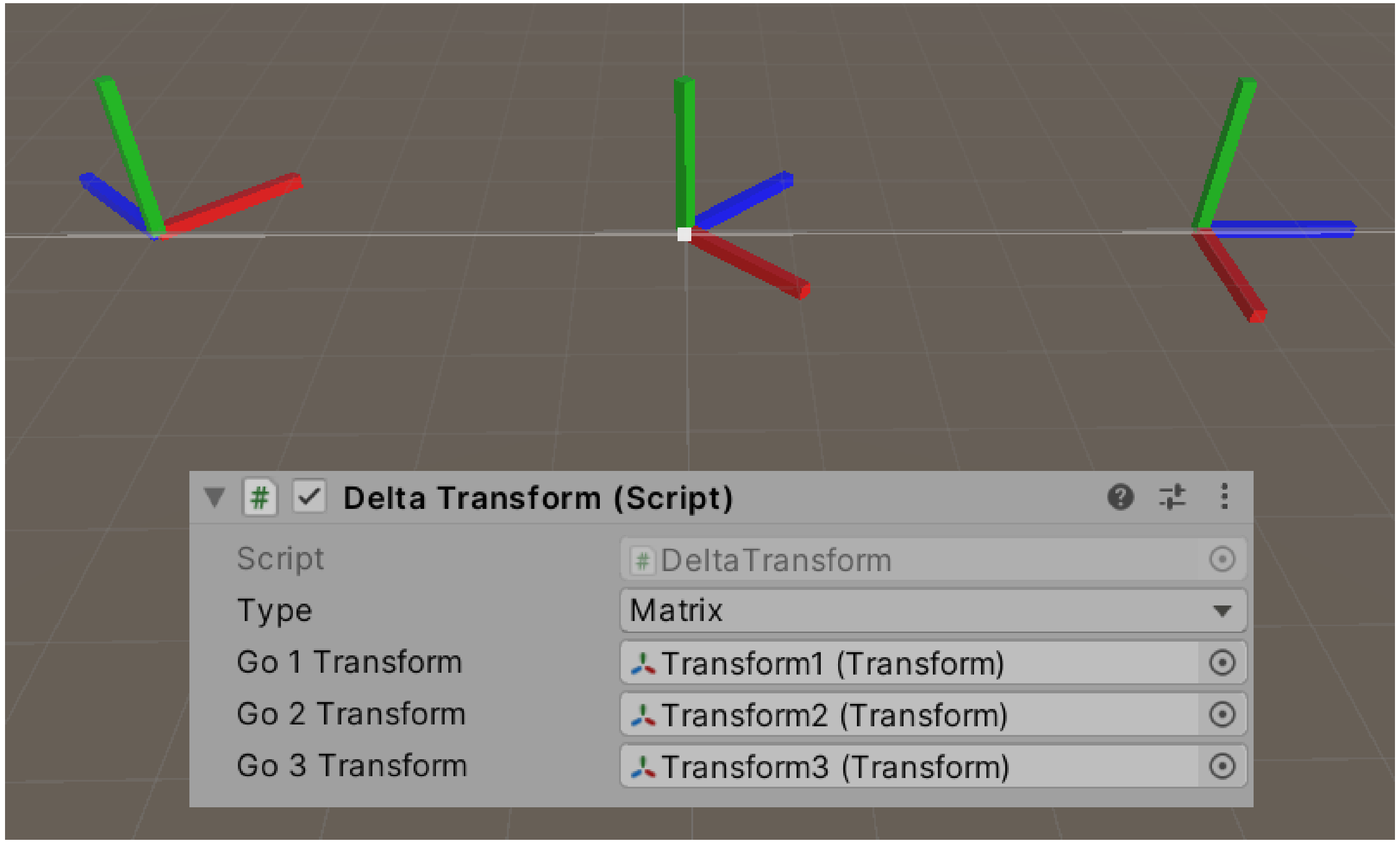
$$(M_2)\left({M_1}^{-1}\right)$$

- The same formula works for quaternions:

$$(q_2)({q_1}^{-1}) \qquad \text{or} \qquad (q_2)({q_1}') \qquad |q_1| = 1$$

- For Euler angles a „ similar” approach will not work

Quaternion, Matrix and Euler Angles


Delta Transform (Script)

Script	DeltaTransform
Type	Matrix
Go 1 Transform	Transform1 (Transform)
Go 2 Transform	Transform2 (Transform)
Go 3 Transform	Transform3 (Transform)

Quaternion, Matrix and Euler Angles

- Euler angles pros:
  - intuitive for humans
  - only 3 numbers that represent rotation/orientation
  - each 3 numbers give us a correct rotation/orientation (easy generation)
  - interpolation between two sets of angles gives acceptable results

- Euler angles cons:
  - can't transform points/vectors with them (need to convert first)
  - there are many triplets representing the same orientation
    try (90, -90, 90) and (90, 90, -90) in Unity (works for ZXY)
  - gimbal lock

Quaternion, Matrix and Euler Angles

- Matrices pros:
  - can transform points/vectors
  - can compose transformations/rotations
  - fast inverse calculation – just a transpose
  - direct access to the basis
  - „natively" used by the GPU

- Matrices cons:
  - as many as 9 numbers for representation (we talk 3x3 matrices here)
  - unintuitive for humans
  - a matrix can be „broken"; we need to make sure it contains valid data
  - interpolation possible but sometimes gives so-so results

Quaternion, Matrix and Euler Angles

- Quaternions pros:
  - can transform points/vectors (less efficiently than with matrices)
  - can compose transforms/rotations (more efficiently than with matrices)
  - 4 numbers for representation; good
  - easy to generate orientations (thanks to normalization)
  - best possible interpolation – slerp

- Quaternions cons:
  - unintuitive for humans
  - a quaternion can be „broken”; we need to make sure it contains valid data
  - to compose with other types of transformations we need to convert to a matrix first

Quaternion, Matrix and Euler Angles

- Euler angles are useful when we want to expose rotation/orientation to the user
- When we go into the „calculations land” – rotations composition and interpolation in particular – quaternions are the most convenient tool
- When we operate beyond rotations (i.e. other types of transforms come into play) or when we work with a graphics API then matrices are necessary

- Usually in the pipeline this is the order that those three forms appear in:

Euler angles (editor/scripts) → Quaternions (scripts/physics) → Matrices (scripts/rendering)

Quaternion, Matrix and Euler Angles

- The full description of an object in 3D space is determined by position, scale and rotation
- Position requires 3 values, scale (arbitrary/non-uniform) also 3
- Orientation with a quaternion is 4 values; when using a 3x3 matrix it is 9
- Overall this gives us 10 (3+3+4) values vs 15 (3+3+9)

- Not quite because a 3x3 rotation matrix also includes scale
- The matrix variant is thus 3 + 9 = 12 values (assuming non-uniform scale)

- Assuming uniform scale: 8 (3+1+4) vs 12 (3+9)

Quaternion, Matrix and Euler Angles

- There is one more representation that you may come across: axis-angle
- The good thing about it is that it requires only 4 values
- The bad thing is that it can’t be used right away: it has to be converted to a quaternion or a matrix first (like Euler angles)
- It is also less understandable for humans than Euler angles
- A great book that discusses and compares all four representations: 3D Math Primer for Graphics and Game Development

Slerp – Spherical Linear Interpolation

- One of the biggest advantages of quaternions is the ability to perform smooth interpolation, called **spherical linear interpolation – slerp**
- None of the other forms can give us such a perfect result (although that might not be needed)
- Slerp can be calculated for both quaternions and vectors

Slerp – Spherical Linear Interpolation

- For formality and context let’s bring up the classical formula for linear interpolation:

$$\mathrm{Lerp}(a, b, t) \;=\; a(1-t) + bt \;=\; a + (b-a)t$$

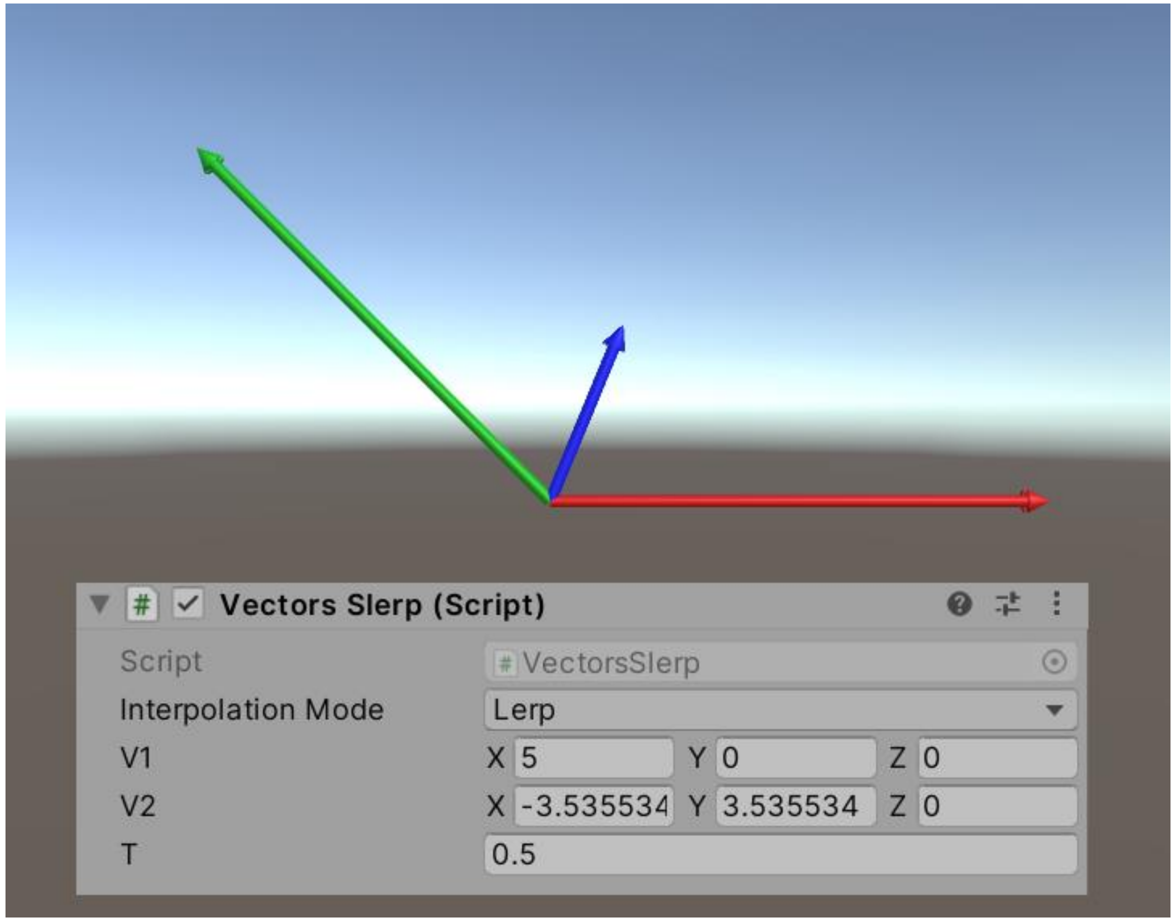
Slerp – Spherical Linear Interpolation

- The formula for spherical linear interpolation:

$$\text{Slerp}(a, b, t) = \frac{\sin\big((1-t)\theta\big)}{\sin(\theta)} a + \frac{\sin(t\theta)}{\sin(\theta)} b \qquad \theta = \cos^{-1}\left(\frac{a \circ b}{|a||b|}\right)$$

- a and b can both be vectors and quaternions. In both formulas
- Note that in case of a quaternion we are talking about a 4D object, which represents an entire basis in 3D (three 3D vectors).
  Interpolation using both Lerp (with normalization) and Slerp will interpolate the whole basis and retain its orthogonality!

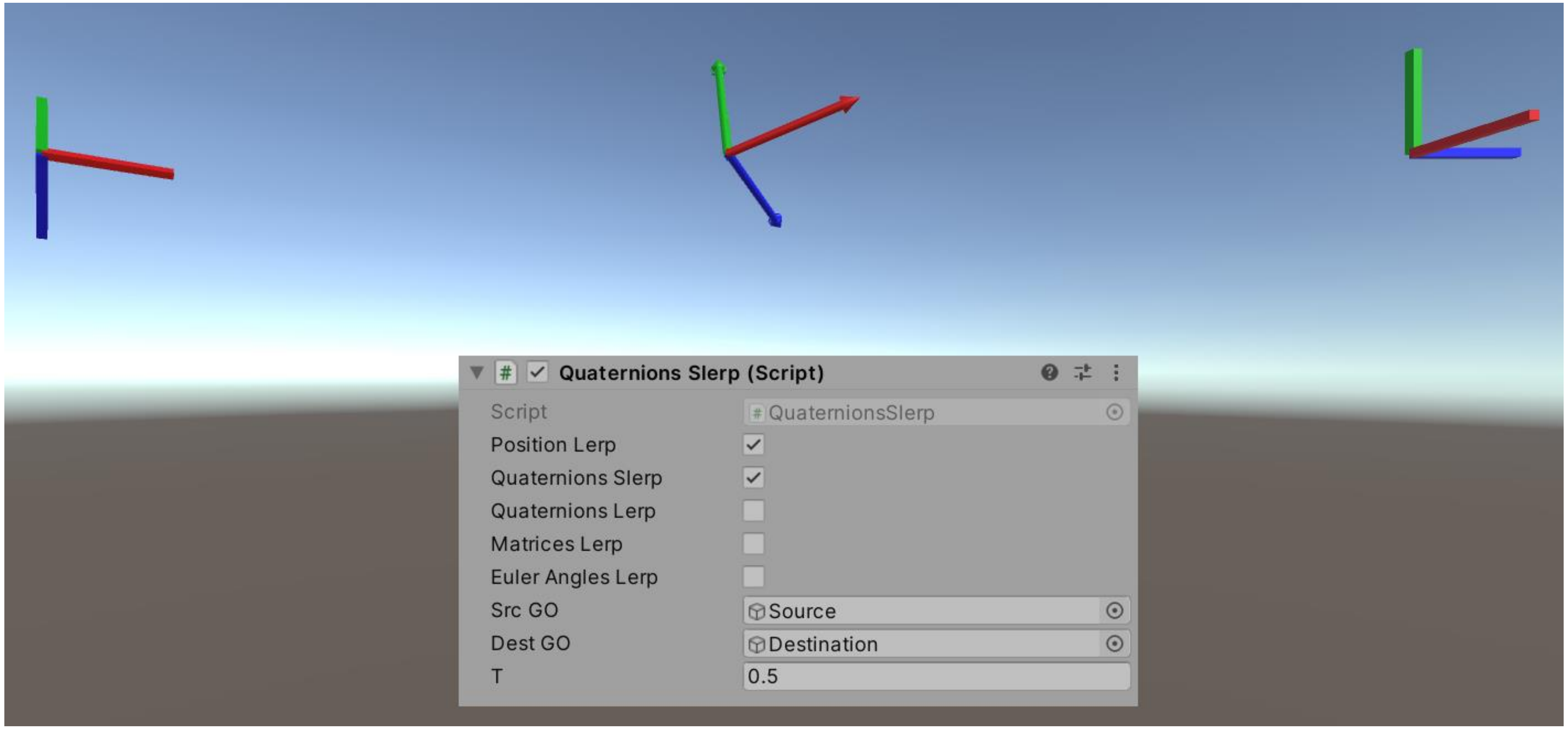
Slerp – Spherical Linear Interpolation



Vectors Slerp (Script)

Script	VectorsSlerp
Interpolation Mode	Lerp
V1	X 5 Y 0 Z 0
V2	X -3.535534 Y 3.535534 Z 0
T	0.5

Slerp – Spherical Linear Interpolation



Quaternions Slerp (Script)

Script	QuaternionsSlerp
Position Lerp	✓
Quaternions Slerp	✓
Quaternions Lerp	
Matrices Lerp	
Euler Angles Lerp	
Src GO	Source
Dest GO	Destination
T	0.5

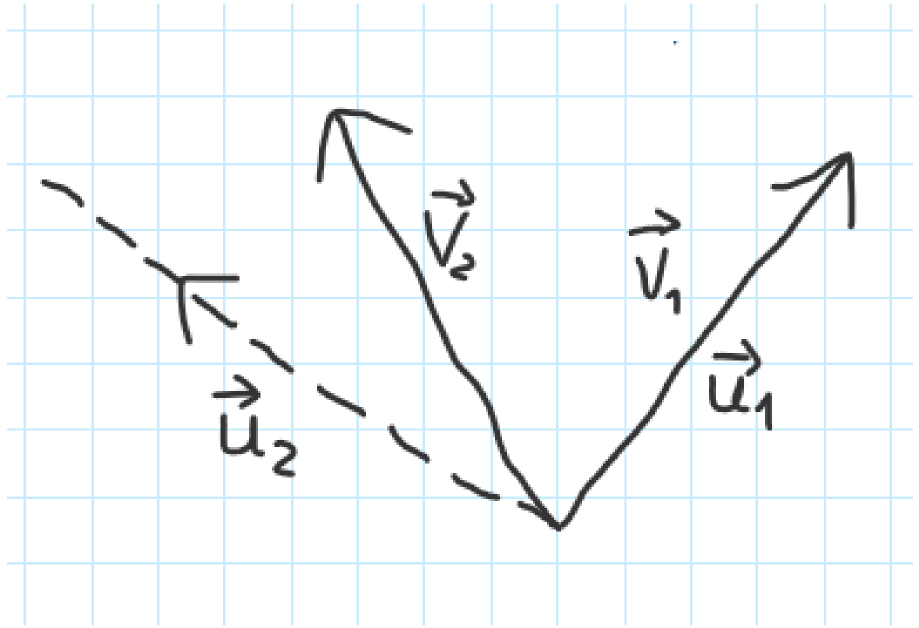
Slerp – Spherical Linear Interpolation

- Quaternion Slerp gives the best results and offers the smoothest interpolation
- Quaternion Lerp usually gives results very similar to Slerp while being more performant (but make sure the resulting quaternion is of length 1; `Quaternion.Lerp` normalizes already)
- Lerp of Matrix and Euler angles also often give acceptable results
- Matrix Lerp requires Gram-Schmidt orthogonalization

- Another idea: Slerp of the basis vectors followed by Gram-Schmidt orthogonalization.
  Quality of interpolation slightly better than matrix Lerp.
  If instead of Slerp we use Lerp the result will be identical to the matrix Lerp

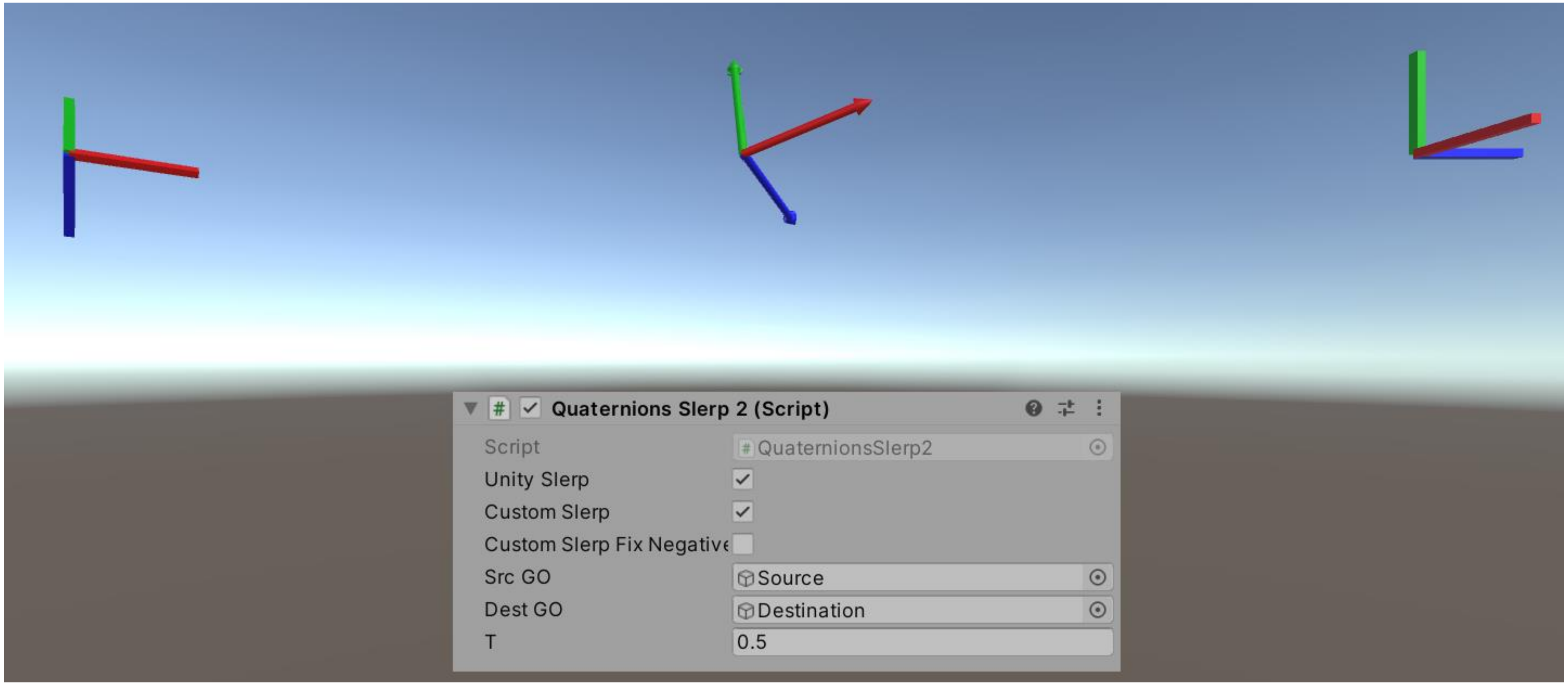
Slerp – Spherical Linear Interpolation

- Gram-Schmidt orthogonalization illustration:



- Fun fact: using this algorithm to construct the basis in program PositionAndOrientation2 (previous chapter) gives there the same result

Slerp – Spherical Linear Interpolation



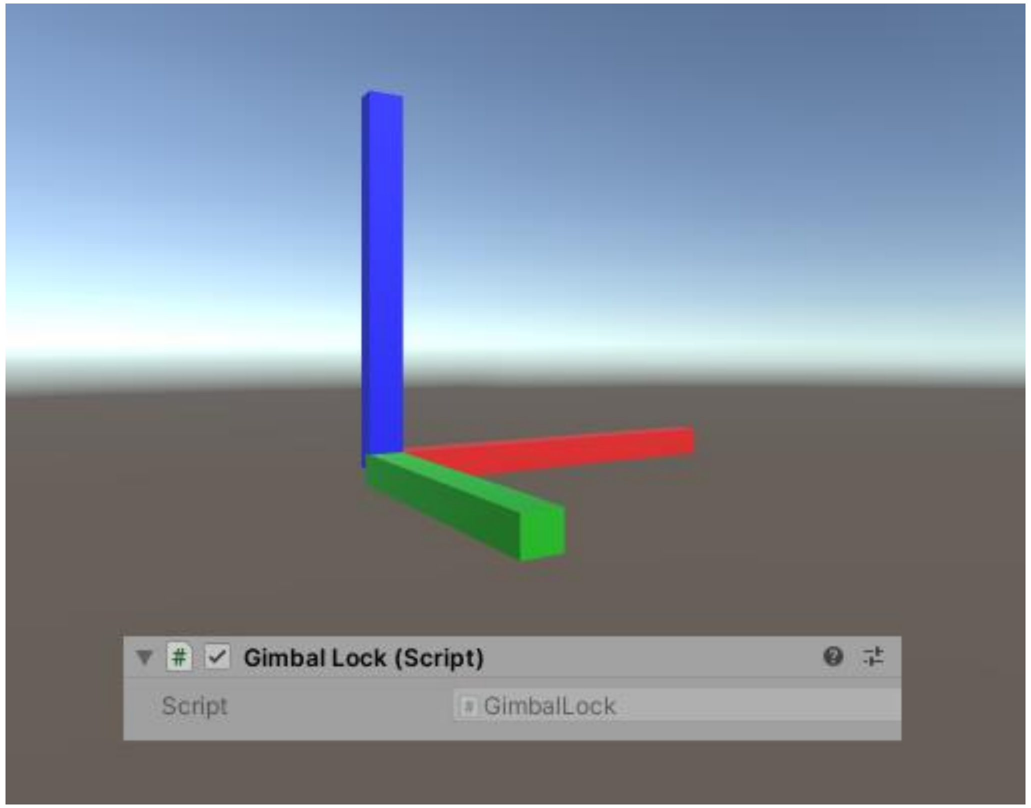
Quaternions Slerp 2 (Script)

Script	QuaternionsSlerp2
Unity Slerp	✓
Custom Slerp	✓
Custom Slerp Fix Negative	
Src GO	Source
Dest GO	Destination
T	0.5

Gimbal Lock

- **Gimbal lock** is a burdensome problem that affects Euler angles
- This problem leads to a situation where we loose the ability to rotate about one of the axes

Gimbal Lock



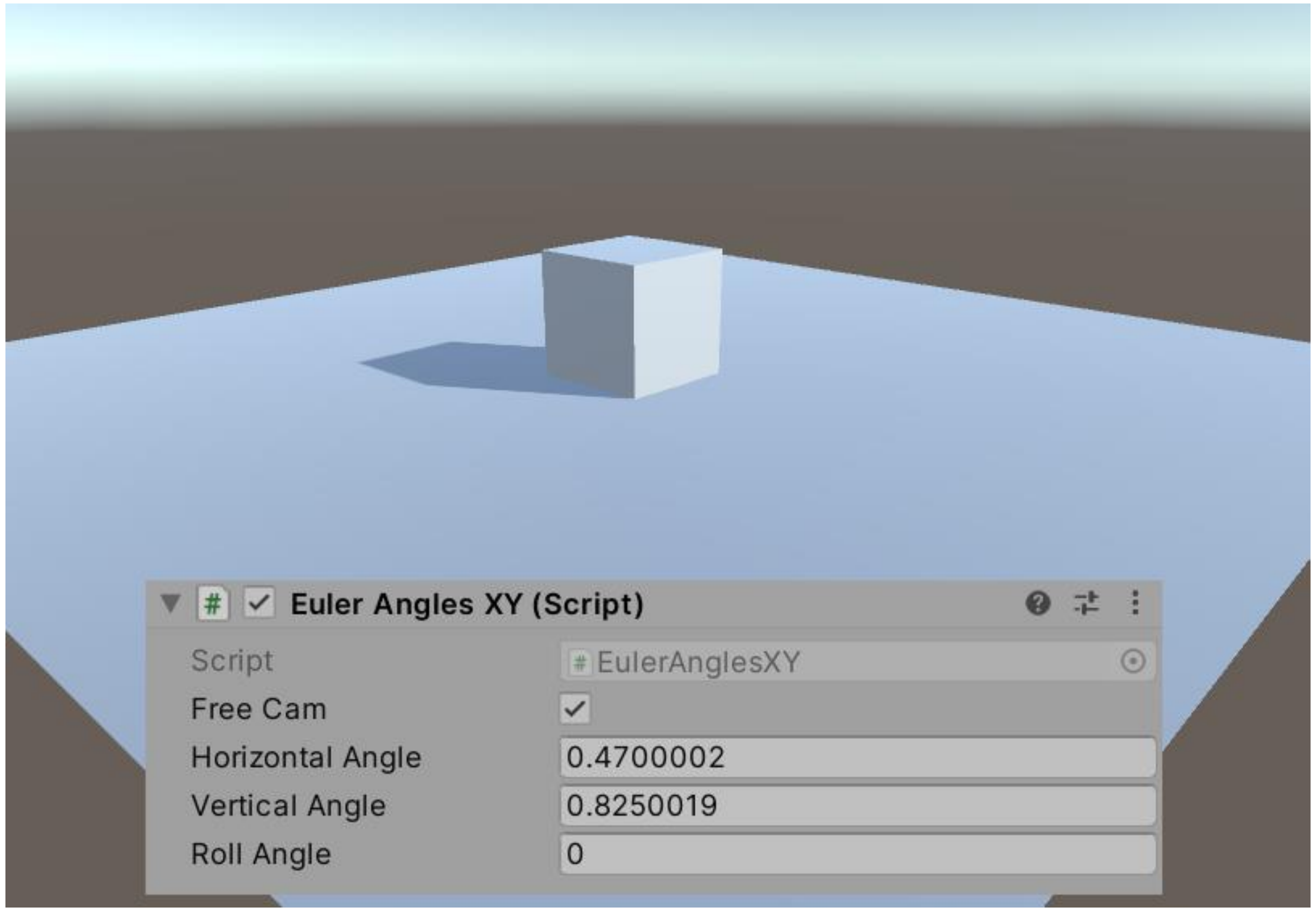
Gimbal Lock

- In many sources we can read that the solution to the gimbal lock problem is to use quaternions – that is not true!
- Gimbal lock is a problem that stems from the fact that the final rotation is a result of composition of three rotations (all around the global axes), where each subsequent rotation affects the previous ones.
  It does not matter if that final rotation has been calculated as a composition of three matrices or three quaternions
- The ZXY sequence of rotations implies that the Z rotation will rotate around the **local** Z axis, the Y rotation will rotate around the **global** Y axis.
  The X rotation will be unpredictable.
  When the local Z axis aligns with the global Y axis we have gimbal lock

Gimbal Lock

- A very good video that demonstrates the problem: What are Euler Rotations in Blender? | How to Avoid Gimbal Lock (2021)

Gimbal Lock


Euler Angles XY (Script)

Script: EulerAnglesXY

Free Cam: ✓

Horizontal Angle: 0.4700002

Vertical Angle: 0.8250019

Roll Angle: 0

Exercises

1. Write a program which rotates a cube around any axis, about an arbitrary point r.
   Implement the rotation using a quaternion (slide 14)