Unity Selftuition – Rotations

Introduction

In Unity you'll want to change the transforms of your objectinstances quite often. The transform of an object determines its position, rotation and scale (size) in the world. If you want to do anything beyond changing textures on a static scene, there is no way around manipulating the transform. Now, this is hardly a problem when translating (changing the position), or scaling (changing an objects size).

When rotating however, this is becomes a problem when you manipulate eulerangles, the easiest way to understand rotations. You'll have to deal with Gimbal Lock. (a phenomenon that makes small, incremental changes to rotations makes them go haywire when over a certain point) To prevent this, Unity internally represents rotations as Quaternions. But what are Quaternions?

In this selftuition, I will try to clear both my own confusion about Rotations as well as any confusion readers might have. I will focus only on 3D.

Settle in! :)

Table of Contents

Introduction........1
Method........2
Work........2
Unity functions........2
Physics........3
Transform.Rotate........3
Euler Angles........3
Gimbal lock........4
Why is this is useful........5
Transform.Rotate again........5
Vector3.RotateTowards........6
Quaternion.LookRotation........7
Quaternions........7
Real and Imaginary values........7
What is a Quaternion?........8
Rotating using Quaternions........8
More........9
Quaternion.LookRotation again........9
Quaternion.RotateTowards........9
Transform.RotateAround........9
Transform.LookAt........10
Quaternion.FromToRotation........10
Quaternion.Euler........10
Quaternion.AngleAxis........10
Quaternion.eulerAngles........10

Transform.eulerAngles......11
Transform.localEulerAngles, Transform.localrotation......11
End......11
Sources / Further material......12

Method

I will do this by looking at the possible things you can use in Unity to rotate objects the way you want them to, by exploring all terms related to rotations I might come across, and by using this knowledge to build simulations and examples that use the methods I learn.

Work

Unity functions

The first thing we're going to do is simply look at what Unity provides for us to rotate things. A quick look into the Unity manual should do this for us. A filtered list is shown below. I have omitted everything that is related to 2D, as this is not the focus of this study, and also functions relating to building your own editor tools, the GUI, basically everything that does not have anything to do with rotating the objects in your scene during runtime. A special category is for the functions relating to Physics. These are interesting and useful for what we want to achieve just as much as “regular” rotating, but I won't focus on these right now.

Transform.Rotate
Applies a rotation of eulerAngles.z degrees around the z axis, eulerAngles.x degrees around the x axis, and eulerAngles.y degre...
Vector3.RotateTowards
Rotates a vector current towards target.
Quaternion.RotateTowards
Rotates a rotation from towards to.
Transform.RotateAround
Rotates the transform about axis passing through point in world coordinates by angle degrees.
Transform.LookAt
Rotates the transform so the forward vector points at /target/'s current position.
Quaternion
Quaternions are used to represent rotations.
Quaternion.FromToRotation
Creates a rotation which rotates from fromDirection to toDirection.
Quaternion.Euler
Returns a rotation that rotates z degrees around the z axis, x degrees around the x axis, and y degrees around the y axis (in t...
Quaternion.AngleAxis
Creates a rotation which rotates angle degrees around axis.
Transform.eulerAngles
The rotation as Euler angles in degrees.
Transform.localEulerAngles

The rotation as Euler angles in degrees relative to the parent transform's rotation.
Transform.localRotation
The rotation of the transform relative to the parent transform's rotation.
Transform.rotation
The rotation of the transform in world space stored as a Quaternion.
Quaternion.eulerAngles
Returns the euler angle representation of the rotation.
Quaternion.operator *
Combines rotations lhs and rhs.

Physics

Rigidbody.MoveRotation
Rotates the rigidbody to rotation.
CharacterJoint.swingAxis
The secondary axis around which the joint can rotate.
ConfigurableJoint.targetAngularVelocity
This is a Vector3. It defines the desired angular velocity that the joint should rotate into.
ConfigurableJoint.targetRotation
This is a Quaternion. It defines the desired rotation that the joint should rotate into.
JointMotor
The JointMotor is used to motorize a joint.
Rigidbody.constraints
Controls which degrees of freedom are allowed for the simulation of this Rigidbody.
Rigidbody.rotation
The rotation of the rigdibody.
Rigidbody.inertiaTensor
The diagonal inertia tensor of mass relative to the center of mass.

Transform.Rotate

public void Rotate(Vector3 eulerAngles, Space relativeTo = Space.Self);

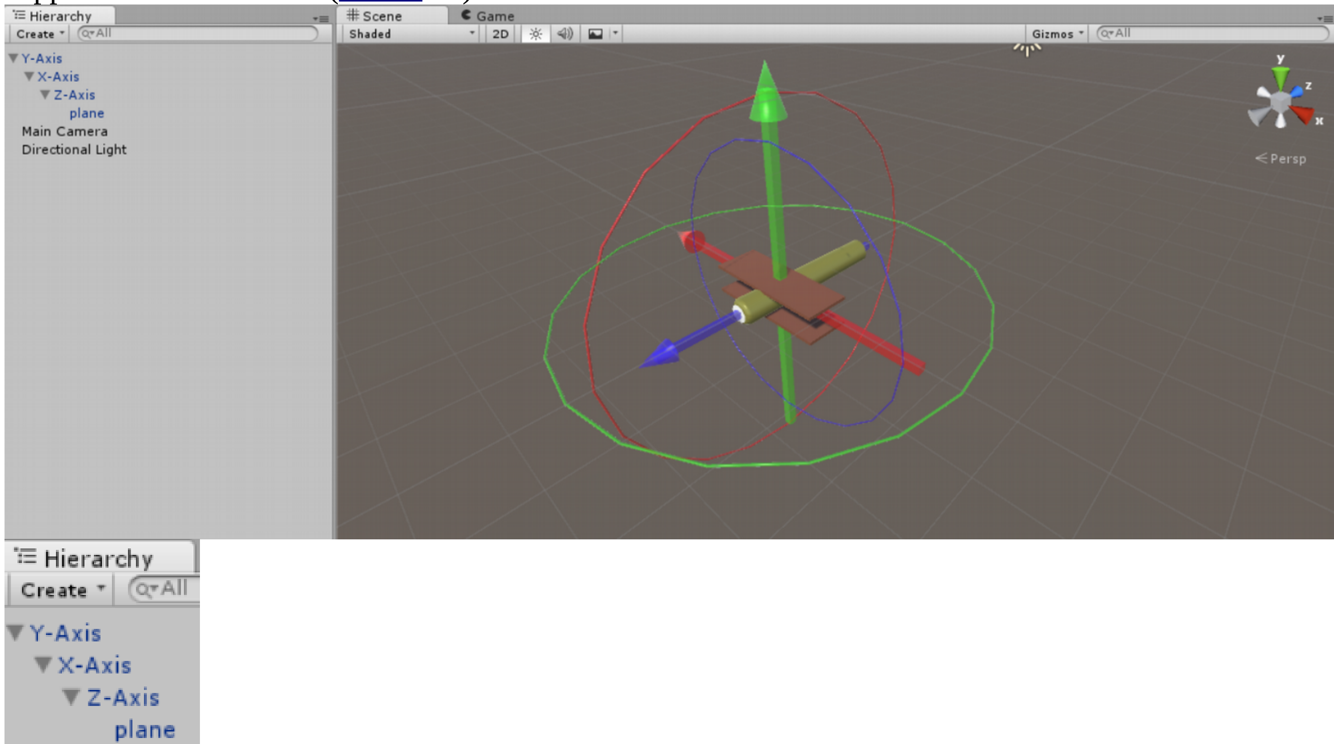
Euler Angles

So what are Euler angles? There is a neat video that explains this (source 1, see Sources)

Euler angles are a system used to represent any rotation in 3D space. This is set up by rotation around three axes, in hierarchical order. This last bit is important as it makes clear that by rotating one axis, you influence the other axes.
In Unity, the order is Z first, then X and lastly Y.

Okay, let's try this out. I'll build an airplane with three axes, in hierarchical order. What I want the plane to do is simple: I want it to be able to pitch, roll and yaw. These are three motions that are simply a rotation around the X-,Y- and Z-axis. Rolling is a rotation around the Z-axis, Pitching is one around the X-axis, and jawing is a rotation around the Y-axis. I made a video to explain what effect I'm pursueing. (source 16)

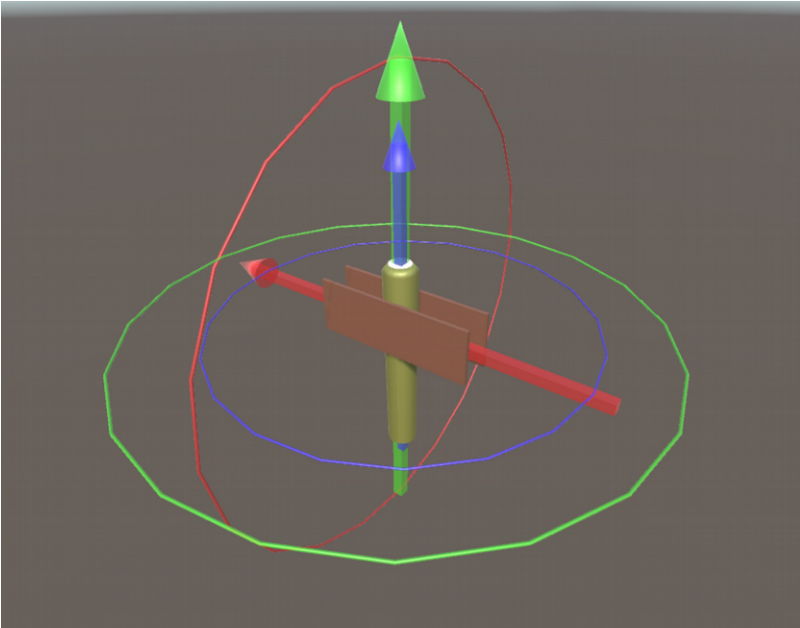
Now please look at this screenshot of the unity editor. Also, you can follow along with the unityproject supplied with this article (source 17)



As you can see, the axes are applied in a hierarchical way. This means that the rotation of the Y-axis influences the rotations of the X- and Z-axes, as well as the plane. The X-axis's rotation affects the Z-axis, and the plane, but not the Y-axis. This order of first Z, then X and then Y is not set in stone, but in Unity it happens to be in this order. The fact that it is set up in an hierarchical way is important, as it means the rotations of the axes influence eachother. This creates Gimbal lock (see below) and it also means a rotation will not behave in the way you might expect.

Gimbal lock

Gimbal lock is what happens when two of the axes align. (See the screenshot below)



Here, the Y-axis and the Z-axis are aligned. This happened by simply rotating the (red) X-axis 90 degrees. We now have a situation in which rotating around the Z-axis and rotating around the Y-axis have the exact same result. That means we've lost the possibility to rotate in three dimensions. When you are interpolating from one orientation to another, this property of using Eulerangles causes the object to rotate in an unpredictable way. Instead of interpolating smoothly between two orientations, it will follow a strange arc because at some point two of the axes align which influences the motion. A visual explanation of this phenomenon can be found under sources (source 3)

Why is this is useful

So what can we use eulerangles for? Eulerangles make it relatively simple to understand rotations as you can always see a rotations as a combination of rotation around three predefined axes. If you want to have a character that looks left and right, you can easily achieve this by applying a float value to the Y-axis rotation.

Transform.Rotate again

public void Rotate(Vector3 eulerAngles, Space relativeTo = Space.Self);

Let's head back to the Transform.Rotate function and see how we can apply this knowledge in creating a rotation we want. Let's use the plane again. We want to let it pitch, roll and yaw by using the Transform.Rotate function. (please rewatch the video to see what this means. Each motion relates to a rotation around one axis)

Transform.Rotate example

```
	public float pitchingspeed;
	public float jawingspeed;
	public float rollingspeed;
```

```
	void Update () {
		transform.Rotate(new Vector3(pitchingspeed,jawingspeed,rollingspeed));
	}
```

Please check the example to see how this behaves. As you can see, the plane is rotating around its own Y, X and Z-axis, constantly updating its rotation, and you can predict what kind of motion you'll get by altering the pitching-, jawing- or rollingspeed.

So this is an easy way to make your object rotate in a predictable way.

There is another thing you can alter here, and that is that you can pass an extra parameter saying the thing that is rotating, will do so around the world axes, instead of the local axes of the rotating object. I've included an example of that as well. In most cases, it makes the most sense for an object to rotate around its own axes. Right now I do not see a use for rotating in relation to world space, but I am sure there is one.

This is a way to make an object orbit another object as well, although it is not really a good practice. Set up two objects (object A and object B) in such a way that A is the centerobject that is being orbited around, and B an orbiting object. Parent B to A so that A becomes the parent, and rotate A using the Transform.Rotate function. This way you have a simple orbiteffect. However, this means you cannot rotate the parent without altering its child, so when using this method, you lose this possibility.

Vector3.RotateTowards

public static Vector3 RotateTowards(Vector3 current, Vector3 target, float maxRadiansDelta,
float maxMagnitudeDelta);

This function doesn't manipulate the transform of an object directly. It treats a given Vector3 as a rotation and interpolates between a target Vector3 and a current Vector3. Because the transform is set up as three Vector3's, we can still use this for rotations though. Let's see what the example this function has, does. Also, check the unity files for an example. (source 17)

Vector3.RotateTowards Unity example

```
	public Transform target;
	public float speed;
	void Update() {
		Vector3 targetDir = target.position - transform.position;
		float step = speed * Time.deltaTime;
		Vector3 newDir = Vector3.RotateTowards(transform.forward, targetDir,
step, 0.0F);
		Debug.DrawRay(transform.position, newDir, Color.red);
		transform.rotation = Quaternion.LookRotation(newDir);
	}
```

Simply put, this interpolates between two vectors in a way that makes sense for rotations. You specify a current rotation and it will nudge this in the direction of the targetdirection you've given it, and return that. This can be very useful, but it does not require much explanation. More interesting are the things they do to make their example work. Why is the direction they're rotating towards (targetDir) the same as the position of the target minus the own position? And why do they set the rotation using Quaternion.LookRotation? What does that do?

Well, the first one is quickly answerable. Let's consider just a single dimension: X. If you have a direction between two vector1s, how can you determine what that direction is? You can see the two vector1s as points on a line. Vector A (4) and Vector B (2). Now how do we determine the direction from Vector A to Vector B? We simply substract A from B.

Vector1 B minus Vector1 A
(2)-(4)=(-2)

This is -2! This is a new Vector1, which we'll call C. To get from A to B is C. But, if you use C to get a direction (in 1 dimension this is either negative or positive), it is also the direction to get from A to B.

So in three dimensions, it uses the same trick across the X, Y and Z-coordinates. Because they do not interfere with eachother while substracting, this can be done safely.

This is quite an important concept. Vectors can be used to represent a position, a way to traverse from one position to another, and at the same time they can be used to represent a direction. This can be confusing, but because a Vector3 is basically just a set of 3 real values, it (sort of) makes sense that these values can represent multiple things.

Quaternion.LookRotation

public static Quaternion LookRotation(Vector3 forward, Vector3 upwards = Vector3.up);

According to the unity documenation, this creates a rotation and returns it as a quaternion. What? So how does this actually work? What are quaternions? Alright, calm down.

Quaternions

I will start off by explaining why you would want to know anything about quaternions. A quaternion is a way to describe orientations and rotations but it is not applied in a hierarchical manner. This means that it does not suffer from Gimbal Lock. This is the main advantage of quaternions over other methods of rotating things.

A quaternion is somewhat similar to a Vector3. It is a format to hold data. Whereas a Vector3 can hold three Real values, a quaternion holds one Real value and three Imaginary values. The terms Real and Imaginary refer to the mathematical definiton of these terms.

Real and Imaginary values

When you square something, you take a number and multiply it by itself. When you take the square root of a number, you find the “original” number that was squared to get the number inside the square root.

Square and square root
x = 2
sqr(x) = 2 * 2 = 4
sqrt(4) = 2 = x

When you square something, it will always become positive. (except for zero) (Because you either multiply a positive number with that positive number, which gives a positive number, or you multiply a negative number with that negative number, which also becomes positive)

Squaring is (almost) always positive
a = 1
b = -1
sqr(a) = 1 * 1 = 1
sqr(b) = -1 * -1 = 1
sqrt(a) = 1 or -1
sqrt (b) = 1 or -1

What happens then if you try to find the square root of a negative number? You get a number that doesn't hold a relationshop to anything real anymore. When you take the square root of -1, you get the imaginary number i.

i
sqrt (-1) = i
sqr (i) = i * i = -1

Okay. So we've established that there is an imaginary number i. Similarly, other imaginary numbers exist.

What is a Quaternion?

So how is this relevant for our understanding of what a quaternion is?

Well, like I said before, you can see a quaternion as a way of formatting data. Instead of the format Vector3(real, real, real) you have the format Quaternion(real, imaginary, imaginary, imaginary). So a Quaternion is a thing that consists of a real value and three imaginary values. These three imaginary values are often labeled i, j and k. Because you have three imaginary values you can use them to rotate in three dimensions.

However, the relationship between the three imaginary values and the real value is not straightforward, ie. you cannot directly convert each imaginary value to a x, y or z-value. This is also why Unity advises you to not manipulate any of the components of a Quaternion directly and instead supplies you with a few functions to use them.

When you have a quaternion, you can create another quaternion you can use to rotate the first quaternion. This is similar to vectors where a Vector3 is a point in 3D space, but can also be a translation or a direction. A quaternion is a point in a sort of bizarre 4D space, and another quaternion can be used to rotate this quaternion.

Rotating using Quaternions

Simply put, you can create a quaternion that describes a rotation, and that rotation can be ANY rotation in 3D space.

When you are using quaternions for rotations, you are using the properties of the imaginary part of quaternions.

You can see the imaginary part of a quaternion as a Vector in 3D space. However, keep in mind that it is actually part of the 4D quaternion! This is just a simplification to make it easier to visualize.

You can choose a rotation quaternion to get the rotation you want. This means you'll choose a quaternion of which the imaginary part is perpendicular to the imaginary part of the first quaternion. There are an infinite amount of quaternions that are perpendicular, however. This second quaternion's imaginary part will be your rotation axis. This means you have an infinite amount of rotation axes to choose from.

Then, you can input an angle that rotates a certain amount around that rotation axis, and you can use the value of that angle to calculate what your rotation quaternion should be to get the desired rotation.

More

That is about as far as I will go in this article. If you are interested in a more indepth explanation of the mathematical nature of quaternions and why they can be used for rotations, please refer to the excellent article “Understanding Quaternions”, linked under Sources. (source 11)

Quaternion.LookRotation again

public static Quaternion LookRotation(Vector3 forward, Vector3 upwards = Vector3.up);

So how can we apply this knowledge about quaternions in Unity? Unity provides an example of how you could use Quaternion.LookRotation to get a similar behaviour to the LookAt function.

Quaternion.LookRotation Unity Example

```
Vector3 LookVector = target.position - transform.position;
Quaternion Rotation = Quaternion.LookRotation(LookVector);
transform.rotation = Rotation;
```

Okay, the first line of code does the same as the Vector3 example, it specifies a direction to look in.

Next, it creates a quaternion that describes the orientation you would get when you were looking in that direction. Finally, it sets the orientation of the transform to that exact orientation, resulting in the thing you're rotating facing the target. You can see this at work in the example Unity project provided with this article. (source 17)

However you can use this orientation to overwrite the orientation of other transforms as well. For example, you might want a row of soldiers to all look to the right, you could specify that orientation with one LookRotation and then pass it to all soldiers. An example of something like this is included in the Unity project.

Quaternion.RotateTowards

public static Quaternion RotateTowards(Quaternion from, Quaternion to, float maxDegreesDelta);
Rotates a rotation from towards to.

This function is similar to the Vector3.Rotatetowards functions, but directly uses quaternions instead of first using vector3's.

Transform.RotateAround

public void RotateAround(Vector3 point, Vector3 axis, float angle);

Another vector function, this allows you to specify a rotation axis and a point to rotate around. It works pretty much like you'd expect. The rotation axis is in world space, so that can give some strange results you might not want.

Transform.LookAt

public void LookAt(Transform target, Vector3 worldUp = Vector3.up);

This function can be used in the specific case you want the transform of the object the script is attached to, to change its orientation directly so it is facing a target you have specified. So it is like applying Quaternion.LookRotation but has a more limited use. However, it is easier to use because you needn't bother with Quaternions or axes.

Quaternion.FromToRotation

public static Quaternion FromToRotation(Vector3 fromDirection, Vector3 toDirection);

A way to get a rotation that describes how to get from a certain direction to another direction.
This differs from Quaternion.LookRotation where you would set the orientation directly. This function gives a rotation quaternion which describes how to get from orientation A to orientation B. This gives the following code:

Quaternion.FromToRotation

```
Vector3 LookVector = Target.position - transform.position;
```

```
	Quaternion Rotation = 
Quaternion.FromToRotation(transform.forward,LookVector);
	transform.rotation = Rotation * transform.rotation;
```

As you can see in the last line of code, you apply the rotation to the original orientation of the object, instead of setting the rotation directly. You could use this to rotate twice I think.

Quaternion.Euler

public static Quaternion Euler(float x, float y, float z);

Using this function you can create a Quaternion rotation using the Euler system. Please pay attention to the order the rotation are performed in.

Quaternion.AngleAxis

public static Quaternion AngleAxis(float angle, Vector3 axis);

This is pretty straightforward.

Quaternion.eulerAngles

public Vector3 eulerAngles;

This allows you to manipulate quaternion rotations by inputting the eulerangles that that same rotation would have. Example:

Quaternion.eulerAngles example

```
Quaternion q = Quaternion.identity;
q.eulerAngles = new Vector3(xfloat, yfloat, zfloat);
```

Quaternion.identity is the quaternion equivalent of vector3.zero

Transform.eulerAngles

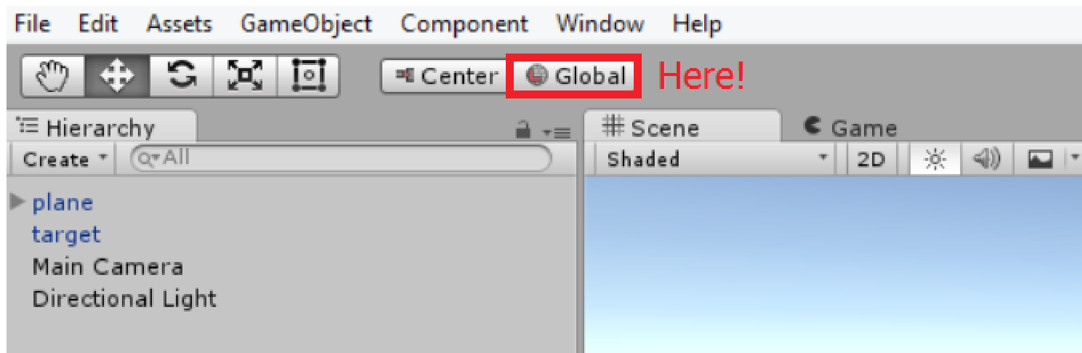
public Vector3 eulerAngles;

The short way of writing transform.rotation.eulerAngles

Transform.localEulerAngles, Transform.localrotation

So far when we're setting transform.rotation, we are setting the world rotation. You have rotation relative to the world axis system, and rotation relative to the parent of the object you're inspecting. If

this confuses you, please look it up or play with the world and local button in Unity itself to get a feel for it. It is probably best to try and move an object instead of rotating it to understand global and local space.



Anyway. The local-methods are for setting local rotation, so it takes the parent's rotation into account.

End

I hope I have supplied you with enough information to understand rotations, as well as some practical examples you can copy into your own projects. As mentioned in the Introduction, I was really struggling with rotations, not knowing how to properly use them. While writing this article I gained a lot of knowledge on how you use rotations and I hope I have succeeded in passing some of that knowledge on. If you feel like you need some more information or maybe hear it in a different way, please check out the sources linked below.

Good luck with your own Unity endeavors! :)

Sources / Further material

Recommended

Eulerangles
1 https://www.youtube.com/watch?v=q0jgqeS_ACM
2 https://en.wikipedia.org/wiki/Euler_angles

Gimbal Lock
3 https://www.youtube.com/watch?v=zc8b2Jo7mno

Quaternions
4 http://unity3d.com/learn/tutorials/modules/intermediate/scripting/quaternions
5 https://www.youtube.com/watch?v=RmElJeJjtWE
6 https://www.youtube.com/watch?v=KdW9ALJMk7s
7 https://www.youtube.com/watch?v=Eh8BU5HUJtY
8 http://developerblog.myo.com/quaternions/

9 http://mathworld.wolfram.com/Quaternion.html
10 http://www.gamasutra.com/view/feature/131686/rotating_objects_using_quaternions.php
11 http://www.3dgep.com/understanding-quaternions/

Rotate functions
12 http://docs.unity3d.com/ScriptReference/30_search.html?q=rotate

Matrices are a thing

Cross Product
13 https://www.mathsisfun.com/algebra/vectors-cross-product.html

unit vector cross-product
14 http://betterexplained.com/articles/cross-product/

similar article
15 http://blog.preoccupiedgames.com/quaternions-not-satan/

PitchRollYaw video
16 https://www.youtube.com/watch?v=F9yZyDrsTds

Unity files
17 http://studenthome.hku.nl/~niels.dejong/files/unity_zelfstudie/Selftuition_rotations.zip