Chapter 9

Misc examples

There isn't much new math in here. Just some semi-practical things we can do with it.

9.1 Getting your y-spin

It's best to set the rotation from your own variables and not try to read back from a quaternion. But sometimes you need to get the y-spin out of a rotation. `transform.eulerAngles.y` might be correct, or it might be off by 360 (like -192 when you wanted positive.) Or, because of the two ways for each problem, it can be off by 180 if x is spun over the top.

This reads the y-spin from a quaternion gets the correct 0-360:

```
float ySpin=transform.rotation.eulerAngles.y;
float xSpin=transform.rotation.eulerAngles.x;
// if the x-spin is "over the top," our y is backwards:
if(xSpin>90 && xSpin<270 || xSpin<-90&&xSpin>-270) ySpin+=180;
while(ySpin<0) ySpin+=360;
while(ySpin>=360) ySpin-=360;
```

Another way to get 0-360 y-spin is to take the forward z-arrow of the rotation, flatten it, and find the angle to due north. That's correct if the real angle is 0 to 180. If the arrow points left (x is negative) you fix it:

```
Vector3 fwd = transform.forward; f.y=0;
float angle=Vector3.Angle(Vector3.forward, fwd);
if(fwd.x<0) angle=360-angle;
```

A more trig-like version of that uses `atan2`. This seems better since it's shorter, but it's very easy to mess up:

```
// gets real math angle, 0=east, CCW, in radians:
float angRad=Mathf.Atan2(v.z, v.x);
// convert to Unity rotations:
float angDeg=90-angRad*Mathf.Rad2Deg;
```

9.2 Using a y-spin

RotateTowards and LookAt are great, but sometimes you just want to use a 0-360 y to spin something.

The first trick is to use your own `float ySpin;` as the master variable. Spin that however you want, and set your rotation from it:

```
float curYspin;

void Update() {
  // just spin us around slowly:
  curYspin+=30*Time.deltaTime;
  transform.rotation = Quaternion.Euler(0,curYspin,0);
}
```

Sometimes you want to spin to a certain rotation. Suppose you're at 350 degrees and the target rotation is 10. It looks like you have to subtract 340 degrees, but 10 is also 370. You really to want add 20 degrees.

Doing that +/-360 math isn't super hard, but Unity has a shortcut: `MoveTowardsAngle`. It pushes the first input to the second, not overshooting, using the shortest way. This uses it to always y-spin us towards where we say:

```
float curYspin;
float targYspin; // pretend someone sets this occasionally

void Update() {
  curYspin=Mathf.MoveTowardsAngle(curYspin, targYspin, 30*Time.deltaTime);
  transform.rotation = Quaternion.Euler(0,curYspin,0);
}
```

`MoveTowards` bases the result on the first number, adjusting the second. For example, `MoveTowards(20,390,move)` gradually pushes 20 to 30 (which is 390-360.) But `MoveTowards(720,-10,move)` gradually turns 720 into 710.

This is good. If you want to start at -200 degrees, you can without it jumping up by 360. And if you convert 361 into 1 in between uses, MoveToward will continue to work.

9.3 Align with ground

When a typical character walks over uneven ground they stand straight up. But sometimes we want something angled with the ground.

The way to find the angle of the ground is to get it's normal – which way is "up" for that crooked section. To angle yourself with the ground: start standing straight up, find the rotation from real up to the ground's normal, and use it to tilt yourself.

This next code plants a randomly spun tree, angled with the ground (pretend we already have the ground's normal):

```
// standing up w/random spin:
tree.rotation=Quaternion.Euler(0, Random.Range(0,360), 0);
// find tilt needed to align with ground:
Quaternion groundTilt=Quaternion.FromToRotation(Vector3.up, norm);
// apply as global rotation:
tree.rotation = groundTilt*tree.rotation;
```

Remember the order matters. In the last line we can think of a real ground tilt, then adding our y-spin around the local y.

FromToRotation does the work here. It isn't used that much, but it's pretty cool when it does.

Suppose we can't start standing straight up. For example we have a LookAt that might also tilt us. We'll use the same plan, but the ground tilt will be from our current up, `transform.up`:

```
// starting angle:
transform.LookAt(ball.position);
// align our up to ground's up:
Quaternion groundTilt = Quaternion.FromToRotation(transform.up, norm);
transform.rotation = groundTilt*transform.rotation;
```

More complicated, suppose we want to gradually spin to face some direction, always tilted with the ground.

The plan is to maintain and move a standing straight spin, and re-tilt it to the ground each frame:

```
Quaternion flatLook=Quaternion.identity; // the flat y-spin

void Update() {
  // wantLook is the standing-straight spin aimed at the ball:
  Vector3 ballPos=ball.position; ballPos.y=transform.position.y;
  Quaternion wantLook=Quaternion.LookRotation(ballPos-transform.position);

  // do the flat y-spin:
  flatLook=Quaternion.RotateTowards(flatLook, wantLook, 60*Time.deltaTime);

  // apply groundTilt:
  transform.rotation = groundTilt*flatLook;
}
```

The `RotateTowards` is sneaky here. It can rotate you in any direction, but with these it keeps your head up and y-spins you since that's the quickest.

We might want to only partly tilt with the ground. In the case, we can use `Quaternion.Lerp` to get a fraction of the angle. This gets a 30% tilt in the ground's direction:

```
  Quaternion wholeGTilt = Quaternion.FromToRotation(transform.up, norm);
  Quaternion groundTilt = Quaternion.Lerp(Quaternion.identity, wholeGTilt, 0.3f);
```

9.4 Orbit camera

A simple orbit camera spins around us in a half-circle, always looking at us. Looking at us is the easy part – the `Lookat` shortcut will do that.

For the half-circle, the plan is to re-use the code making something spin around us. The old code spun us in a ring over a single axis, but there's no reason we can't use x and y together, to aim anywhere.

Since the camera's "base" position is directly behind us, we'll have the to-camera arrow start pointing directly backwards. So y=0 looks forward, and y=180 is in front of us, looking back at us. x will go from 0 to 89.

Simple code to do that:

```
float camYspin=0, camXspin=0; // keys (or mouse) spin these

void Update() {
  Vector3 toCam=new Vector3(0,0,-20); // long behind-us arrow
  Quaternion qCam = Quaternion.Euler(camXspin, camYspin, 0);

  theCam.position = transform.position+qCam*toCam;
  theCam.LookAt(transform.position);
}
```

A neat thing about using a backwards line is +x finally goes up.

We probably also want this to spin with the player. As the player turns, a behind camera should stay a behind camera. We can do that by adding the player's rotation. A little trial and error says it goes as a global, so is in front.

This would go before we use `qCam`:

```
  // if the player is doing a y-Spin, add it to the camera rotation:
  qCam = transform.rotation*qCam;
```

For real we'd add more math. Instead of using the player's position, which might be at the feet or the center of the body, we'd compute a point near the head.

Instead of doing the LookAt to where we started, we might compute a point a little ahead of the player. As the angle swings around to the front we might

make the distance to the camera longer or shorter.

There's often a way for the camera to zoom inward if something is between it and us. When that's gone, we don't want the camera to suddenly snap out. We can do that by computing where the camera should go, then using a MoveTowards to go there

These two lines would replace the one `theCam.position=` line:

```
// prevent camera snaps:
Vector3 camWantPos = transform.position+qCam*toCam;
theCam.position = Vector3.MoveTowards(theCame.position, camWantPos, mvSpd);
```

9.5 Drawing a line between

To get a line between two objects, take a length one line (like a plane or a cylinder.) Put it exactly between your objects, aim it at the second one, and stretch it to be as long as that distance.

The object you use should be set-up so its z forward axis is the stretchable part. For example a Unity cylinder stands up, running along +y. We'd want to use the parent trick to have it aim along +z. Stretching it then gives a nice, long tube.

The code:

```
Vector3 toBall = ball.position-transform.position;
line.position = transform.position + toBall/2;
line.LookAt(ball.position);

// stretch it:
Vector3 lineScale = Vector3.one;
lineScale.z=toBall.magnitude;
line.localScale=lineScale;
```

This like goes from center to center. It might be nicer to start and stop at the edges of the shapes. Suppose the target ball has a radius of 0.5, so we want to stop that far away. And let's go nuts and assume `toWing` is the local offset to where the line starts (it starts from our wing, just because):

```
Vector3 lineStart=transform.position+transform.rotation*toWing;
Vector3 toBall=ball.position-lineStart;
toBall=toBall-toBall.normalized*0.5f; // shorten arrow by 0.5
// toBall now runs from our wing tip to the edge of the ball

// position, aim and stretch the same as before:
line.position=lineStart+toBall/2;
line.rotation=Quaternion.LookRotation(toBall);
line.localScale=new Vector3(1, 1, toBall.magnitude);
```

It the line is just a flat plane this will have it pointing straight up, since that's how lookRotation works. We'll often be seeing just an edge. A common billboard trick is to z-roll so the plane tries to face the camera.

We can do that by giving `LookAt` an UP arrow towards the camera:

```
Vector3 toCam = myCam.transform.position-line.position;
line.LookAt(ball.position, toCam); // <- using 2nd UP for lookAt
```

This is pretty much the math the built-in lineRenderer is using.

9.6 Consistant LookAt

If you use `LookAt` or `LookRotation` to track a moving object, leaning all the way back can snap your head around, which won't look very nice.

Here's a short program showing the problem. It has us look at an airplane flying straight over our head:

```
float zz=4;

void Update() {
  Vector3 lookOff=new Vector3(0,4,zz); // always 4 up, moving towards us
  transform.LookAt(transform.position + lookOff);

  zz-=2*Time.deltaTime;
  if(zz<-4) zz=4;
}
```

LookAt likes to first y-spin us towards the airplane, then lean back, which will be at most 90 degrees. When the airplane flies directly over, then goes behind us, we don't keep leaning backwards – we quickly turn around.

The fix is slippery, so I'll just write it and you can play around if you want. Pick a different UP for lookAt. Pick one where the object will never be. Suppose you only plan to look at things above you. Using `Vector3.back` for the z-spin works well:
`transform.LookAt(target, Vector3.back)`
Adding it to the code above will have you lean back more and more, with no snapping.

Another use of the "change where up is" trick is to tell `LookAt` to respect your current roll. Give it your current y-axis as UP, like this:
`transform.LookAt(marker.position, transform.up);`

That always gives a smooth rotation, but looks pretty bad – it doesn't prefer head-up at all, so will gladly have you lying on your side or back. Again, like a movie badguy trying to decide how to hold a pistol so it looks cool.

An improvement is to read our previous UP, but gradually spin it to the real up:

```
// find a slightly more up version of my current up:
Vector3 up = Vector3.RotateTowards(transform.up, Vector3.up, 30*Time.deltaTime, 0);

transform.LookAt(target, up);
```

It doesn't work for the code with `zz` above (it can't tell whether to twist left or right,) but it will if you move the airplane a little left or right, like 0.01 for `x`. It gives a nice effect.

We could also have it ease-in to the LookRotation. Suppose our target teleports and we don't want to snap – we want a quick spin to the new facing. This uses `RotateTowards` to ease-in from the current rotation to the new one:

```
void Update() {
  Quaternion wantLook = Quaternion.LookRotation(target - transform.position);
  Quaternion look;
  look=Quaternion.RotateTowards(transform.rotation, wantLook, 120*Time.deltaTime);
  transform.rotation=look;
}
```

This version will also spin to be head-up, since the target rotation is head-up.

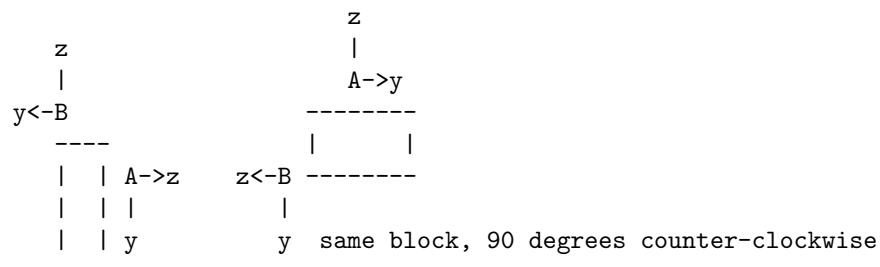
This RoateTowards trick will also help a moving object gradually tilt to the ground angle. Compute your ground-tilt angle as normal, then use the second-to-last line above to ease into it.

9.7 Connect two blocks

Suppose we have a block with certain marked spots where it can connect to other blocks. We usually mark the connection spots with an empty child . Its +z faces outward to show the "plug-in" direction. We call that a mount-point.

When we connect some other block, we find a mount point on it, face the +z's towards each other, and line up the y's (the x's will automatically be opposite.)

This picture shows two identical blocks. They have mount-points A and B, rotated as shown. The second block is rotated 90 degrees left, connecting point B with point A on the first block. Notice how the z's face each other and the y's line up:



Assume block 1 is anywhere, spun however. The problem is figuring out how to place block 2 so the proper mount points connect.

Here's just the set-up, finding the A and B mount point children:

```
public Transform cube1, cube2;

void Start() {
  Transform mp1A = cube1.Find("A");
  Transform mp2B = cube2.Find("B");
  Vector3 3 startPos = mp1A.position; // <- children have positions

}
```

The last line is cheating a little (but is legal.) Point `A` is a child of our block, and is positioned using localPosition, but we can still look up its real position (and also its real rotation.) So we'll ignore block 1 from now on and start with A.

The next part is much harder. Point B from cube2 goes in the exact same spot as point A from cube1. That's not so bad. It has the same rotation, except spun 180 on local y. Still not so bad.

The problem is we have to position and rotate cube2, based on where we know point B needs to be.

Obviously, we just need to add one thing – `mp2B.localPosition`. That's the arrow from cube2 to child B. But we want to go the other way, so have to flip it with a negative 1.

The last, hardest part is getting cube2's rotation correct. Starting from A, we: spin 180 on local y, then the *opposite* of B's localRotation (it's the rotation from cube2 to B and we're going the other way.)

This is all very tricky, and took me a few tries to get right:

```
public Transform cube1, cube2;

void Start() {
  Transform mp1A=cube1.Find("A");
  Transform mp2B=cube2.Find("B");
```

```
  Vector3 mpAPos = mp1A.position; // start from here

  // cube2 rotation in three parts. Gather parts first:
  // 180 flip from point A to B:
  Quaternion y180=Quaternion.Euler(0,180,0);
  // opposite of B's local rotation:
  Quaternion qBtoCube2=Quaternion.Inverse(mp2B.localRotation);

  cube2.rotation=mp1A.rotation*y180*qBtoCube2;

  // now use rotation on backwards B offset:
  cube2.position = mpApos + cube2.rotation*(-mp2B.localPosition);
}
```

This took a ton of testing. I finally realized I could set the rotation first, without moving cube2, and could test to see if it worked.

An alternate way of doing all of that is tricking Unity into setting things for us.

We'd like to place point B in the right spot, rotated correctly, and have the rest of Cube2 be where ever it has to be from there. If we temporarily flip it so B is cube2's parent, that will happen. Then we undo it. We still need to 180 flip on y:

```
public Transform cube1, cube2;

void Start() {
  Transform mpA=cube1.Find("A");
  Transform mpB=cube2.Find("B");

  // temporary flip so B is parent of cube2:
  mpB.parent=null; cube2.parent=mpB;
  // snap B to A, spun 180:
  mpB.parent=mpA; // child of A so we can use local coords
  mpB.localPosition=Vector3.zero;
  mpB.localRotation=Quaternion.Euler(0,180,0);

  // redo B child of cube2:
  cube2.parent=null;
  mpB.parent=cube2;
}
```

This version might be easier to visualize. It's not any faster – Unity has to do all of the math from the first version to set all the children.