Chapter 9

Misc examples

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

9.1 Controlling your y-spin

Sometimes we only want a simple 360-degree spin, usually around y.

The simplest way to do that is hand-moving our own `y` variable, then setting rotation from it:

```
float ySpin; // the main control of our rotation

void Update() {
  // sample spin, slow clockwise:
  ySpin+=30*Time.deltaTime;

  transform.rotation = Quaternion.Euler(0, ySpin, 0);
}
```

This method also works if the player can use arrow keys to add and subtract from `y`. We could fix it when `y` goes outside of 0-360, but we don't have to.

If we want to spin to a certain number, there's a problem – we have to account for wrap-around.

Say we're at 50 degrees now. To spin up to 120 degrees, we add. But because of wrapping, we should *subtract* to go to 330 degrees. We could add, but we'd be going the wrong way, spinning for more than 1/2 a circle.

The solution involves a few `if`'s, but Unity has a common built-in that takes care of it:

`MoveTowardsAngle(d1, d2, 4)` adds or subtracts 4 from `d1` to move it closer to `d2`, accounting for wrap-around. Like MoveTowards, it also won't overshoot.

This code lets us set `targYspin` and spin the shortest way to it:

```
public float targYspin; // pretend someone sets this occasionally
float ySpin;

void Update() {
  // spin to target, using shortest way
  ySpin=Mathf.MoveTowardsAngle(ySpin, targYspin, 30*Time.deltaTime);

  transform.rotation = Quaternion.Euler(0, ySpin, 0);
}
```

Like MoveTowards and friends, `30*Time.deltaTime` means it spins 30 degrees/second.

Just so you know, it's made to never snap `d1`. If `ySpin` is 350 and `targYspin` is 10, this will gradually increase `ySpin` to 370. But it's safe to adjust your numbers into 0-360 whenever you want.

9.2 Getting your y-spin

When controlling spin through your variables, you always know your degrees.

But sometimes we're free-spinning (LookAt, rotating over funny axis, or just rolling on the ground), and want to find our y-spin.

It so happens `transform.eulerAngles.y` is always the correct 0-360 y-facing. It even fixes over-the-top x-spins:

Suppose `y` is 20 but `x` is tilted way back to 170. You're really aimed the opposite, at `y=200`. The system also thinks that – it flips `y` to 200 and changes `x` to 10.

For fun, this is basically the math to fix things:

```
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 between it and +z. Since the `Angle` function can't tell left from right, it's only correct if the angle is on the right side. If the arrow points left (x is negative) we flip it:

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

Here's a more trig-like version that uses `atan2`. The main use is to convince you that using real trig is very error-prone:

```
// 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;
```

Reading the `x` degrees doesn't work very well at all. The system keeps it between -90 and 90. But it stores negatives as 270 to 360. If you set `x` to -150, the system adds 180 to `y` and changes `x` into -30, then 330. Ick.

But reading `y` works fine.

9.3 Align with ground

When a game character walks over uneven ground they generally stay straight up. But sometimes we like it when things tilt with the ground: maybe it rides on treads.

The secret to applying ground tilt is getting the normal. For flat ground, the normal is up. Tilted ground's normals are mostly up, but tilted a little. The plan is the find the rotation between those two arrows. Then we can place our object for flat ground, and hit it with the ground tilt.

Getting the normal is in a previous chapter (it depends on how the ground is made). Computing the difference between arrows is the rare `FromToRotation`.

This code plants a tree tilted with the ground. The tree is first randomly spun on `y`, to make it more interesting:

```
// straight 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 to the tree:
tree.rotation = groundTilt*tree.rotation;
```

The deluxe version of this trick is when the tree is using some fancy math to spin in place. Keep the tree's "flat" rotation stored, change it however, then re-apply the ground tilt every time.

```
//  saved copy of tree's rotation on flat ground:
```

```
Quaternion baseTreeSpin;

void Update() {
  // do complicated stuff to change baseTreeSpin here,
  // maybe slowly y-spin it to face something

  // re-apply groundTilt:
  transform.rotation = groundTilt*baseTreeSpin;
}
```

The tilt of the ground feels like it's the base rotation. But this logic says we can think of it as an add-on.

Sometimes we want to partly tilt with the ground. If you remember, `groundTilt*0.3f` isn't legal, but `Quaternion.Lerp` can do it. This tilts us only 30% with the ground:

```
  // the usual line:
  Quaternion groundTilt = Quaternion.FromToRotation(transform.up, norm);

  // standard Lerp to get a percent of an angle:
  Quaternion gt30 = Quaternion.Lerp(Quaternion.identity, groundTilt, 0.3f);

  tree.rotation=gt30*Quaternion.Euler(0,ySpin,0);
```

With all of these, moving objects cause a snapping problem. Ground like /\ or V will snap the tilt back and forth. A `MoveRotatation` can smooth it out: compute the rotation, but instead of using it, smooth yourself into it:

```
  Quaternion groundTilt = Quaternion.FromToRotation(transform.up, norm);
  Quaternion wantRotation=groundTilt*myStraightUpRotation;

  // ease ourself into it:
  transform.rotation=Quaternion.MoveRotation
      (transform.rotation, wantRotation, 30*Time.deltaTime);
```

9.4 Orbit camera

A simple orbit camera spins around us in a half-sphere (the top half), always looking at us. Looking at us is the easy part – the `LookAt` shortcut will do that.

For the half-sphere part, it's like aiming a cannon with x and y. Imagine the barrel is clear, with a backwards camera glued to the end.

When we use rotations to spin an arrow, we need to pick a good base arrow. In this case, it should be pointing behind us. That's the normal place a camera

starts, so we can see where we're going. Because of that, we'll say that 0 degrees counts as behind us.

The code:

```
float camYspin=0, camXspin=0; // keys (or mouse) spin these
// y can have full spin, x is 0 to 89 only

void Update() {
  Vector3 toCam=new Vector3(0,0,-20); // long behind-us arrow

  // standard y,x euler spinning 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 arrow is how +x finally tilts up instead of down like it normally does.

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 the `qCam` arrow, to spin it with the player:

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

For real, we'd tweak this: adjust the center of the spin arrow and the LookAt point. Otherwise we might be spinning around the player's feel, aimed at the feet. But that's just simple arrow math.

Sometimes the camera might be temporarily controlled by something else. We'd like it to smoothly zoom back into the orbit spot, instead of an ugly snap.

We'll compute the camera position as normal, then use 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(theCam.position, camWantPos, mvSpd);
```

`mvSpd` should be large enough that normal spins are instant, but a camera put out in Cleveland would have a fast zoom to where it should be in the orbit.

9.5 Changing length of arrows

We can change arrow lengths pretty well with normalizing and scaling. But some things always confuse me, and they also have a shortcut.

Suppose we have a length 7 arrow and want it to be length 5. We're pretty good at cutting them in half, or making them 10% longer. But this is different.

The long solution, which we know, is to normalize it, then take it times 5. Turning 7 into 5 is hard, but turning it into 1 then 5 is easy.

A cool shortcut is doing both at once. This makes it so arrows can't be longer than 5:

```
float len=toBall.magnitude;
if(len>5) // drop down the length:
  toBall*=(5/len); // <- makes any arrow be length 5
```

Dividing by `len` normalizes it, then multiplying by 5 makes it that long. `arrow*(wantLen/currentLen);` resizes any arrow to the length you want.

A similar problem is adding or subtracting from the length of a line – making it 0.5 longer. We can do it like:
`arrow*=(len+0.5f)/len;`. That's the same trick.

The main thing is we're used to plus/minus being easier than multiplication. But for arrows, it's the other way around.

9.6 Drawing a line between

To get a line between two points, take a length one object, put it exactly between the points, aim it at the second one, and stretch it to be that distance.

The object you use should be set-up so its z forward axis is the stretchable part. For example a Unity cylinder has height 2 and runs along +y. We'd use the parent trick to aim it along +z and cut the length in half.

The code to make a line between me and a ball:

```
// halfway between, looking at the end:
Vector3 toBall = ball.position-transform.position;
line.position = transform.position + toBall/2;
line.LookAt(ball.position);

// now stretch it to cover the full distance
Vector3 lineScale = new vector3(1, 1, toBall.magnitude);
line.localScale=lineScale;
```

Placing midway works if the origin is in the center – stretching goes equally in both directions. If the origin was at the back then a z-stretch goes only for-

ward. We'd need to place it at the start.

That plan makes the line go center-to-center (really, origin to origin). If it's going between 3D models, like a duck and a frog, that won't look so nice. For fun, we can try two different ways where the line really starts and stops:

We often want the line to come from a specific part of us, maybe our eyes. Doing that is simple arrow math:

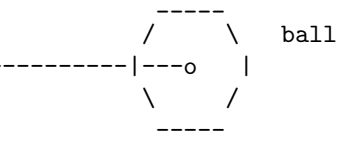
```
// arrow from our center to our eyes:
Vector3 toEyes = new Vector3(0, 1.5f, 0.3f);
Vector3 eyePos = transform.position + transform.rotation*toEyes;

// now same as before, but using eyePos:
Vector3 toBall = ball.position-eyePos; // from eyes
line.position = eyePos + toBall/2; // start at eyes
  ...
```

It's more common to mark the eyes using an empty child, named `"eyePos"` positioned right between the eyes. We'd use it like this:

```
Vector3 eyePos = transform.Find("eyePos").position;
// the rest is the same
```

Since it's a child, the system keeps it at our eyes as we spin and move, and sets it's `position` as the actual location. We only need to look it up.

For another way to draw a nicer line, pretend it ends at a see-through ball. We want it to stop right at the edge without going inside:



After getting the center-to-center line, we can subtract the radius of the sphere. Since we use the line starting from us, that shrinks the far end and it stops short:

```
Vector3 toBall= ...

// shrink by 0.5:
float lineLen=toBall.magnitude;
toBall*=(lineLen-0.5f)/lineLen;
lineLen-=0.5f;
 ...
```

Sometimes a line looks better if it goes just a little bit inside of what it hits.

One final trick involves playing with the free z-spin from a `LookAt`. Often the line is merely a stretched 2D square. To avoid seeing an edge, we'd like the z-spin to face the flat part (the top) towards the camera as much as possible:

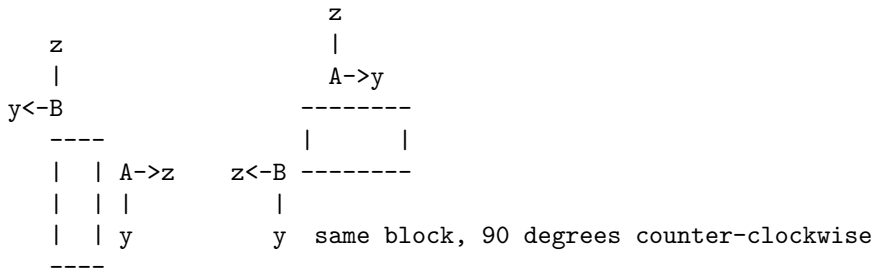
```
// arrow to camera is used for UP:
Vector3 toCam = myCam.transform.position-line.position;
line.LookAt(ball.position, toCam); // 2nd input controls z-spin
```

A square that spins to perfectly face the camera is a Billboard. This version, where it only spins on its z, is called an Axis-Aligned Billboard. It looks good from any side, but looking down the line gives a ugly edge-on view.

9.7 Connect two blocks

Suppose we have blocks with plug-ins on the sides. We usually mark them with empties: +z is straight out and +y shows the spin. Another block needs it's +z facing ours, spun to have the +y's lined up.

For example, this shows two copies of a block, with A from the first lined up with B from the second:



The problem is figuring out how to place and rotate block 2 so it plugs into block 1 using the slots we wanted.

First we need to find the two plugs: A from block 1 and B from block 2. They're children:

```
public Transform cube1, cube2; // assume we have these

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

Now the problem is a little simpler. We're only trying to get block 2 to line up with A.

Since B goes in the same spot as A, the only thing left is to follow the arrow, backwards, from B to the center of its block. But which direction? We need to account for A's rotation, flipped 180 degrees, then account for B's rotation on it's block. Yikes!

It 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, around y:
  Quaternion y180=Quaternion.Euler(0,180,0);
  // opposite of B's rotation with respect to it's block:
  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. The first testing step was to try to get block2's rotation correct, without trying to move it yet.

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

We can temporarily make B the parent of block2. Then we can place B on A and let Unity do the work figuring out where block2 is. We still have to use A's rotation, flipped 180:

```
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);
  // this also positions cube2

  // redo B as child of cube2 again:
  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.