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`. A nice thing is we could decide to manually make `y` wrap around at -180 and +180, even though Unity prefers 0-360.

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

Say we're at 50 degrees now and want to rotate to face 330. We should *subtract*. It's only 80 degrees backwards (50 down to 0, 30 more from 360 to 330). It turns out that 50 should spin forwards for any angle up to 230 (180+50), otherwise backwards.

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

`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. For example `MoveTowardsAngle(50,330,4)` is 46.

This code uses it. We set `targYspin` and it spins 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.

It won't snap the numbers. For 50 and 330, it moves 50 down to 0, then past to -30 (which is the same angle as 330).

9.2 Getting your y-spin

Sometimes we're free-spinning (LookAt, rotating over funny axis, or just rolling on the ground), and want to find our compass facing.

It so happens `transform.eulerAngles.y` is always the correct 0-360 y-facing. That's because of the funny way Unity stores them. Suppose we enter (170,5,0) for rotation. y=5 is basically North, but 170 on x flips us over to South. But the Inspector is lying. Unity really stores that as (10,185,180). y correctly says we're facing basically South.

Another way of getting our basic compass facing is to take the forward z-arrow of the rotation, flatten it, find the angle between it and +z, then figure out left vs. right:

```
Vector3 fwd = transform.forward; f.y=0;
float angle=Vector3.Angle(Vector3.forward, fwd); // this is 0 to 180
// if the arrow was facing left, flip degrees to negative:
if(fwd.x<0) angle*=-1;
```

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; // yikes!
```

Trying to read `transform.eulerAngles.x` doesn't work as well. For one thing, -10 is the natural way to say "10 degrees up", but Unity stores that as 350. For another, x's past 90 are fixed. If you set x to 100, it becomes 80 (and y and z flip by 180).

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;
}
```

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);
```

If we use this while walking around, we get a common problem – our rotation will tend to snap. For example, as we cross the top of a ridge. The standard trick is to compute the rotation we want, but then use `MoveRotatation` to smooth 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, we can imagine a very long selfy-stick, spinning on x&y the usual way.

When we use rotations to spin an arrow, it helps to pick a good base arrow. For a camera, we want it to start behind us. Our base arrow should go backwards. y=0 looks forward and y=180 looks backwards (the camera will be ahead of us, looking at us, which is backwards). x=0 looks flat, with x up to 90 putting the camera higher up, looking down:

```
float camYspin=0, camXspin=0; // keys (or mouse) spin these
// y can have full spin, x should be 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:

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

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 sent out to Cleveland would take a second or two to zoom back.

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;`.
If `len` was 6, this makes it length 1, then multiplies that by 6.5.

The main thing is that 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;
```

That plan makes the line go center-to-center (really, origin to origin). It will tend to go inside of them. That might be fine, or not. Suppose we want a line that goes from our eyes to the surface of a sphere. We can adjust the endpoints:

```
// arrow from our center to our eyes:
Vector3 toEyes = new Vector3(0, 1.5f, 0.3f);
Vector3 eyePos = transform.position + transform.rotation*toEyes;
// eyePos is now 1 end of the line

Vector3 toBall = ball.position-eyePos; // from eyes
// pretend we know the ball has a radius of 0.5. Shrink arrow:
float len=toBall.magnitude;
toBall=toBall*((len-0.5f)/len); // shrink line so it hits the edge
len-=0.5f;
line.position = eyePos + toBall/2; // start at eyes
```

Getting the eyes was easier, since it's a fixed point from us. Going to the edge of the sphere is more complicated because it's not one fixed spot – it's whatever part of the sphere is facing us. The best way I could do it was by shrinking the arrow.

There's one final trick. If your line is just a stretched 2D square then the z-spin will make a big difference. If the face is aimed +y then it will normally be pointing straight up. From the side we'll only see the edge, which will look bad. We'd like to z-spin the stretched square to always face us (it probably can't face us exactly, but as close as possible).

Assume the flat part is aimed up (if it's not, the parent trick can make it that way). We can use the 3rd "head this way" input to aim the face at the camera:

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

This is known as an Axis-Aligned Billboard. A square that perfectly aims only at us is a regular billboard.

9.7 Connect two blocks

Suppose we have blocks with plug-ins on the sides. Each plug-in is an empty child, rotated to show how it fits. +z is is where it plugs in and +y is how they must be rotated. Another block needs its +z facing ours, spun to have the +y's lined up.

This shows two copies of the same block, with A from the first lined up with B from the second:

```
                       z
   z                   |
   |                   A->y
y<-B                 --------
   ----              |      |
   |  | A->z    z<-B --------
   |  | |          |
   |  | y          y  same block, 90 degrees counter-clockwise
   ----
```

The problem is figuring out how to place and rotate the second block so B fits correctly into A. The first block could be anywhere, spun any way.

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

```
public Transform block1, block2; // assume we have these

void Start() {
  Transform mp1A = block1.Find("A");
  Transform mp2B = block2.Find("B");
  Vector3 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 its block. Yikes!

It took me a few tries to get right:

```
public Transform block1, block2;

void Start() {
  Transform mp1A=block1.Find("A");
  Transform mp2B=block2.Find("B");
  Vector3 mpAPos = mp1A.position; // start from here

  // block2 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 qBtoBlock2=Quaternion.Inverse(mp2B.localRotation);

  // use them to compute block2's final rotation:
  block2.rotation=mp1A.rotation*y180*qBtoBlock2;

  // now use that rotation on B's offset, backwards:
  block2.position = mpApos + block2.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 doing a little dance with parents and children. If we can set it up correctly, we'll be able to move the B connector to where it should go, letting it drag block2 with it.

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 block1, block2;

void Start() {
  Transform mpA=block1.Find("A");
  Transform mpB=block2.Find("B");

  // temporary flip so B is parent of block2:
```

```
  mpB.parent=null; block2.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 block2

  // redo B as child of block2 again:
  block2.parent=null;
  mpB.parent=block2;
}
```

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.