Contents

Chapter 0

Intro

This is about the math for 3D positioning and rotating, and the computer code. There are examples and some tricks, but the goal is for you to know enough to figure out oddball move and spin problems for yourself.

The working examples run in the Unity3D game engine, written in C#. No special version, since no advanced features are used. You could probably read this if you don't use either – C# and Unity are fairly generic.

The examples assume you know the basics of coding, and basic Unity3D set-ups (you can place a script on a gameObject, know how to drag things into script Inspector slots). But they don't require much more.

0.1 Review of the basics

Before we start moving and rotating, it's nice to have a review of 3D basics and the choices Unity makes:

xyz axes: Different systems aim these in various directions. In Unity, y is up and down. x is the usual left/right, leaving z as forwards and backwards. A funny effect of this is a standard floor runs along x&z (not x&y as you might expect).

Positive/negative run in the obvious directions: +y is up, +x is to the right, and +z is forward (further away from you, if you're looking from the front).

For examples: 3D trees made for Unity have the y axis running from roots to crown. A 3D cow made for Unity should be facing along +z, since that's forward.

Coordinates: There aren't any special coordinate values. You can place things where-ever you like. For example:

- y less than 0 isn't "underground" or underwater. The ground, water, lava, is wherever you put it.

- There's nothing special about negative values. For example (0,0,0) can be in the center. Half the map will be have negative coordinates, which is fine.
- There are no special out-of-bounds values. The world can go as far as you want, in any direction. Edges are where-ever you place them.

It's probably better to put things somewhat near (0,0,0,) just to keep the numbers small. But you can easily move things around later.

Units: The real world meaning of units is whatever you want. In other words, the 3 in (3,0,0) can stand for any real distance. For examples:

- If your game takes place on a board, 1 unit = 1 square is nice. Or maybe 10 units per square. That way you can make "big" pieces 9x9, leaving plenty of round numbers for smaller pieces.
- For a game taking place somewhere real, you often have real measurements in actual meters or yards. In your game, use that for your units. If you have a game about mice, and know real mice measurements in inches, use inches for the units.
- The 3D models you use aren't a good cue for scale. You often have a cow in meters, a duck in inches, and a shovel in "who knows?". Pick a scale first, then adjust the models to it.
- It's fine to not know the scale. Just place things where-ever they look good. If something forces you to compute the scale later, and it works out to some funny decimal, that's not really a problem.
- Officially Unity uses 1 unit = 1 meter. But that only matters for gravity. You may not be using gravity, and it's easy and common to change gravity's value. It's completely safe to not use meters for the units.

Scale: It makes sense to scale a cow 10% larger or smaller. But for a box or a log, being able to stretch it independently along x, y, or z makes more sense. It turns out that everything can stretch/shrink on x, and y, and z. To scale the whole cow 10% larger, you have to stretch it 10% on them all (just enter 1.1 in all 3 slots).

We always have the option of making our cow twice as wide (scale 2,1,1), even though it would look fake.

Model origins: Suppose we place a cow at one specific xyz. Which exact part of the cow will be there? The tip of the nose? The center? The feet? The answer: it could be any, depending on how the cow was made.

The "placement point" is officially called the model's *origin*. It's technically the (000) used when the model was created. We can't change it. It's usually in an obvious spot, but not always.

For example, trees and cows usually have their origins on the bottom. You can place them on the ground and have the whole tree going "up". Symmetric objects often have it in the center. A shovel might have it at the center of the handle, to make placing it in a hand easier.

0.2 Slightly less basic

Rotations: 3D rotations are shockingly complicated. We'll play with them more later, you won't need to memorize this:

The Inspector has rotation slots for x, y, and z. Imagine those are rods running through the origin of the model (yes, the origin also controls how an object rotates). Essentially, y spins us like a top, x rolls us forwards and z rolls us sideways.

Strange at first, x isn't a true-forward summersault. It's uses forward for the cow, after the y-spin. z is the same way. You can spin around the real x and z using the visual rotation tool, but you'll see x,y, and z all recalculated in the Inspector.

There's more: the Left-Hand Rule says which way counts as positive rotation: grab the axis for the rotation with your left hand, thumb in the positive direction. The way your fingers curl is a positive rotation. Try this for x and it tells you +x is a forward roll, meaning -x is a tilt backwards. For z this rule tells you +z rolls left, which seems funny, but the math won't work unless it does.

Basically, 3D rotations can seem simple at first. But once you start using them, you start needing more and more rules and seeing more weirdness. The trick, which we'll see later, is not to use x, y, and z.

Common problems with real 3D models
This section is for if you want to bring in some 3D models for the rest of the examples. That can be helpful, but there can be some fixable problems:

- They can be much too big. 3D modelling programs sometimes like to use 1 meter = 100 units. When you bring in a model that large, you may not be able to see it, since it completely surrounds you.
- They can be facing the wrong way. Some programs think y is forwards, with z as up. This can be confusing when we use commands that assume +z is the front.
- The origin can be somewhere we didn't want it. Maybe it's below the feet, but we'd prefer it centered.

The parent trick
This trick can fix everything wrong with a model:

First pick out an obvious space. For a cow, make a cow-sized +z-facing cow parking space for it to stand on. Then adjust the model to fit there. Spin it to face +z if needed, fix the scale so it's your perfect cow-size, rotate so it seems to be aimed on +z, and move it to fit the space the way you want, feet on the ground. Nothing special involved yet. This step is totally obvious.

Next create an empty gameObject. Put it where you want the origin to be: center of the cow, on the ground, at the cow's nose – wherever you want. Do not change this empty's scale or rotation. Only more it to your preferred cow-origin.

Finally, in the panel with the names, drag the real cow into the empty, making it a child. The cow model is now locked into that empty and will track it as it moves. We'll never touch the real cow again. We want to freeze those adjustments we made. The parent now counts as the cow

```
cow1 <- an empty gameObject, renamed cow1
  cow <-cow model. Adjusted the way we want, then moved inside of cow1
```

To see it working, move the empty around, spin it, scale it. It looks just like we're playing with our dream cow.

Chapter 1

Vectors and offsets

Unity3D handles math with xyz points in the standard way. This first part is those rules, plus how they translate in Unity:

3D points are named `Vector3`, and have dot-x, y and z. They're structs, so using `new` is optional:

```
Vector3 p; p.x=0; p.y=6; p.z=30; // p is (0,6,30)
```

They have a constructor. `p=new Vector3(0,6,30);` is the same as the lines above.

You can add two xyz's, and it happens *pairwise*: x to x, y to y and z to z. The result is another xyz. The code below shows this:

```
Vector3 A, B, C;
A = new Vector3(2,  5, 20); // creating 2 things to add
B = new Vector3(2,  6,  7);

C = A + B; //  (4, 11, 27); // <- adding each column (pairwise)
```

Subtraction is also pairwise. C=A-B subtracts each column:

```
A = new Vector3(10, 20, 50);
B = new Vector3( 2,  6,  7);
C = A - B; //  ( 8, 14, 43);
```

The other useful shortcut is multiplying by a single float. Each part is multiplied by that number:

```
A = new Vector3(2,  5, 20);
B=A*3; //      (6, 15, 60)
```

We call the single number a **scalar** since it scales the vector by that amount.

As usual, operators can be combined and mixed, with times going before plus. `A+C*3` triples everything in C, then adds it pairwise to A. We can also use shortcuts like `A+=B;` and `A*=2;`

There are shortcuts for common Vector3's. Two handy ones are all 1's and all 0's:

```
Vector3 A = Vector3.one; // (1,1,1)
A = Vector3.zero; // (0,0,0)
```

The last is a nice way to blank something: `A=Vector3.zero;`. The first one is often used with scalars. `A=Vector3.one*7` is an easy way to make (7, 7, 7).

There are also shortcuts for 1 unit in all six directions. These use the Unity orientation, so forward is positive z. Ex's:

```
A = Vector3.right; // (1,0,0)
A = Vector3.left; // (-1,0,0);
A = Vector3.up; // (0,1,0)
A = Vector3.down; // (0,-1,0)
A = Vector3.forward; // (0,0,1)
A = Vector3.back; // (0,0,1)
```

These are nice with scalars for creating points. `Vector3.up*5` is a nice-looking way to write (0,5,0). Or we can combine them to "walk" to a point:

```
A = Vector3.back*4 + Vector3.up*9; // (0, 9, -4)
```

That's the same as `new Vector3(0,9,-4);`. But sometimes it's nicer if we can see it as "4 backwards and 9 up".

Note that all of these are merely shortcuts. `C=A+B` is a short way to write `C.x=A.x+B.x;` and the same for y and z. You may have also noticed there's no `A*B`. That's on purpose. We could easily make it – just multiply pairs – but we'll never want to use it for anything, so it's better not to have it.

1.1 Points and Offsets

Vector3's are often spots on the grid. It seems like that's all they can be, but there's a second common way we use them, as arrows or *offsets*.

Suppose we have a 2D map, and Cardiff is 2 west and 3 north of Glaston, written as (-2,3). Saying Cardiff is at (-2,3) is wrong – that's in the ocean – but also something we'd never do by mistake. "(-2,3) from Glaston" is a totally understandable use. That's what an offset is.

In our vector math, it's going to be important to keep points vs. offsets straight. We can do some useful, and easy math using offsets. Often we'll think of them as arrows. We'll also need to be careful not to use an offset as if it were a a point – not to look for Cardiff in the ocean.

A 3D offset warm-up: suppose cows wear a bluetooth in their left ears. To place it, we'll want directions from the cow's xyz position to the ear. We know the cow's origin is center bottom. That means our left ear offset is up (y) from the ground, forward (z) by 1/2 a cow then left a little. Let's say it's (0.25, 1.5, 1).

To find the left ear of any cow, we take the cow's position, plus the offset. The whole thing:

```
Vector3 toLeftEar = new Vector3(-0.25f, 1.5f, 1); // same #'s as before

Vector3 cow1pos = ... ; // could be just anywhere
Vector3 cow2pos = ... ; // just anywhere else

// final blueteeth positions for each cow:
Vector3 cow1blueTooth = cow1Pos + toLeftEar;
Vector3 cow2blueTooth = cow2Pos + toLeftEar;
```

Let's check the math. `cow1Pos + toLeftEar` says to start between cow 1's hooves and move 1.5 up, 1 forward then a little left. Sounds good – it's the math we worked out. The cool thing is how a "Pos+Offset" equation feels so natural: start at the cow's base, and follow the arrow to the ear. We don't need to think about how it's doing pairwise xyz addition. It's just as accurate, and easier, to imagine adding an arrow to a point.

To keep going: `cow2Pos + toLeftEar` figures out where the 2nd cow's bluetooth should be. A different cow, plus the same offset, gives the ear of the new cow.

Even more fun, suppose blueteeth have floating antenna, which are `toAntenna` from the actual bluetooth. `cow1+toLeftEar+toAntenna` finds it. A position plus two arrows, end-to-end. It's like following a treasure map: start at the cow, walk to the ear, then walk a little to the antenna.

The actual rule is that you have to start at any position, and can add any number of offsets. Of course, the offsets have to make sense. cow1Pos+toAntenna is junk: toAntenna has to start from a bluetooth. cow1Pos+toLeftEar+ToLeftEar would only work if one cow had another standing on its left ear.

Offsets can be scaled. It's super cool. Suppose we have a 10% larger cow. All we need to do is make `toLeftEar` 10% longer. `cow3Pos+toLeftEar*1.1f` will go to the left ear of the larger cow3.

You can think about `toLeftEar*1.1f` the math way – x, y, and z are each 10% larger. But we usually think of it as stretching an arrow to be the same direction, but 10% longer. It almost seems like cheating, but the math works, and arrows are the preferred way mathematicians think of offsets.

Before the next example, let's do some clean-up. So far, the cows can't be rotated. That seems limiting, but in a few chapters we'll fix it by rotating the offset. The other funny thing is "position plus offset = position". That feels wrong, like "Apples plus Oranges gives Apples". But the math checks out, and you get used to it pretty quickly.

1.1.1 Tilted checkerboard

For this longer example we want to find the squares on a tilted checkers board (8x8 squares). We only need someone to enter the 2 bottom corners. They don't have to be on a straight line and can be any distance apart:

```
// lower-left and lower-right corners. Enter using the Inspector:
public Vector3 cornerLL, cornerLR;
```

The first step is to find the arrow from the left to right corner, and divide it into 8th's. That will give the corners of every bottom square. The math to get an arrow along the bottom is a new rule: subtract two points to get an offset from one to another:

```
Vector3 leftToRight = cornerLR - cornerLL;

// testing. Left corner plus offset:
cornerLR + leftToRight; // this is the right corner
```

It seems weird at first, but a point minus another point gives us an offset. It literally gives the xyz's between them, but thinking of it as a single arrow is accurate and easier. Notice how it's the right corner minus the left. It's End - Start to get the correct arrow.

Now that we have the arrow, we can use scalers to takes 8th's, then add to the left side. Some sample corners of bottom squares, using math we've already seen:

```
// start at lower-left corner, then walk 1/8th of the way to the right:
Vector3 sq1 = cornerLL + leftToRight/8;

// written out longer:
Vector3 acrossSq = leftToRight/8; // shortcut for "1 square over"
Vector3 sq2 = cornerLL + 2*acrossSq; // 2 squares over
Vector3 sq3 = cornerLL + 3*acrossSq; // 3 squares over
Vector3 sq4 = sq3 + acrossSq;
```

The math works out so nicely since it's on a board, with perfectly arranged same-sized squares. And remember the cool part, it works for tilted boards. A possible picture:

```
           sq1  sq2        cornerLR
            o   o  ---------->o
         o----------
   cornerLL

000 (could be anywhere)
```

To walk up the squares, I'm going to cheat and use a 10th grade math trick. If you remember, you can make a 90 degree counter-clockwise rotation by flipping x and y and taking the new x times -1 (you don't need to remember the trick for the rest to make sense). We can use that to spin `leftToRight` and get `bottomToTop`:

```
// slightly boring 90 counterclockwise spin:
Vector3 bottomToTop;
bottomToTop.y = leftToRight.y;
bottomToTop.x = -leftToRight.z;
bottomToTop.z = leftToRight.x;
```

Notice how I'm respecting Unity's coordinates. x&z counts as flat on the ground, and that's what I'm flipping, keeping y the same. Also notice how we're spinning an arrow. We can't spin positions. Arrows get to have all the fun.

We can walk to any corner of any of the 64 squares by starting at the lower-left corner and adding multiples of the across and up offsets:

```
Vector3 acrossSq = leftToRight/8; // same as before
Vector3 upSq = bottomToTop/8; // 1 square up

// corner of a middle square:
Vector3 sq34 = cornerLL + acrossSq*3 + upSq*4;
```

The last line is considered good vector math, and should feel very natural: start at the corner, walk 3 squares across, then 4 squares up. You could think of it as 7 single-square arrows, or as 2 different length across and up arrows.

To wrap up this part, we can find the other two corners:

```
// upper-left corner:
Vector3 cornerUL = cornerLL + bottomToTop;

// upper-right corner (starting from lower-left):
Vector3 cornerUR = cornerLL + bottomToTop + leftToRight;
```

Notice how the math is still Position+Offsets. In the second one, I went up then across. Across then up would be the same.

Finally, suppose we need to find the center of each square. Each center is 1/2-way across and up from the corner. We can compute this and save it in an offset:

```
Vector3 cornerToCenter = acrossSq/2 + upSq/2;
```

It's another semi-new rule: an offset plus an offset is another offset. Or, to put it another way: adding two arrows makes a single diagonal arrow.

This finds the center of the 8th square along the bottom. It moves there with the usual math, then adds the offset to the center, similar to the cow's bluetooth's antenna:

```
// center of lower-right square:
sq18 = cornerLL + acrossSq*7 + cornerToCenter;
```

The end of this chapter has a math review. But a mini review: 1) Only offsets can be scaled or added together, 2) to use an offset, you have to start from a point and add it, 3) you can make an offset by subtracting two points.

1.2 `transform.position` + offset

This section is about seeing the math working in Unity/C#. We'll have one script, placed on a cube that will be "us", then 3 sample cubes that our code will place at various nearby spots, using vector math.

I'll assume you know how to make the cubes, put a script on one, know what `Update()` does, know `transform.position` is where we are, and know how to drag objects into Inspector slots in a script. I'll call the extra cubes red, green and blue. I'll assume you know how to color them (if you want to. All we really need is to be able to tell them apart).

This will place the other cubes around us using 3 simple, slightly different types of offsets:

```
// links to the 3 other no-script cubes:
public Transform redCube, greenCube, blueCube;

void Update() {
  redCube.position = transform.position + Vector3.right*3; // 3 units right

  Vector3 greenOff = new Vector3(4, 1, 0.5f); // diagonal angle from us
  greenCube.position = transform.position + greenOff;

  // blue piggy-backs off green:
```

```
  blueCube.position = transform.position + greenOff*1.5f;
}
```

All three are simply "Position + Offset" math. They just look a little different. `transform.position` is new, sort of, but it's just a Vector3. Likewise `Vector3.right*3` is just another way of using an offset of (3,0,0). Blue used green's arrow, except 50% longer; or another way to say it – blue and green are at different spots along the same arrow.

A fun way to test this is by adding a Rigidbody and collider to us, with a floor and bounciness. The other 3 cubes will track us as we spin and roll around.

Playing with the code should do the obvious things. Substitute Vector3.forward*4. Change the greenOff numbers, tweak the multiplier on the last greenOff. Try using the style `Vector3.right*2+Vector3.up*4`. Nothing exciting, but we can visually test all of our math.

A note here: the rules say that in `transform.position + greenOff`, the second thing must be an offset. It's a Vector3, but how do we know it's an offset? The rule is that if you were thinking of "how far from something else" when you made it, it's an offset. If you were thinking "exactly where does it go" then it's a position.

For our second real example, lets pick one point a little away from us, and place the other cubes evenly spaced, using scalars:

```
// enter anything long enough:
public Vector3 cubeLine;

void Update() {
  redCube.position = transform.position + cubeLine; // end
  blueCube.position = transform.position + cubeLine*0.66f;
  greenCube.position = transform.position + cubeLine*0.33f;
}
```

This is similar to the checkerboard using 1/8th, 2/8ths and so on. Except here we're using thirds. Since it's running in Update, we can change `cubeLine` as it runs, and see the cubes snap to the new positions.

Instead of using fixed percents, what if we place one cube on the line, using a percent that gradually goes from 0 to 1? The math isn't very exciting, but the result should be really cool. It will zip along the line, over and over:

```
public Vector3 cubeLine; // same as before

float pct = 0.0f; // we'll make this go from 0 to 1

public Transform greenCube; // we only need 1 other cube
```

```
void Update() {
  greenCube.position = transform.position + cubeLine*pct;

  // make pct go from 0 to 1:
  pct+=0.01f;
  if(pct>1) pct=0;
}
```

Instead of moving along a line given to us, we might want to move to a target. In this case, we want to zoom up to the red cube. That's easy enough: use the End minus Start trick to compute the offset to it, then re-use the code from before:

```
// shoot green cube from us to the red one:
float pct = 0.0f; // 0 to 1 percent of line
public Transform redCube, greenCube; // red is target, greenMoves

void Update() {
  // arrow from us to the red cube:
  Vector3 cubeLine = redCube.position - transform.position;

  // the rest is the same:
  greenCube.position = transform.position + cubeLine*pct;

  // make pct go from 0 to 1:
  pct+=0.01f;
  if(pct>1) pct=0;
}
```

`cubeLine = redCube.position - transform.position` is the interesting part, and a chance to check our new math. transform.position and redCube.position are each positions. Our rules say we can subtract positions, giving an offset from one to the other. So that checks out.

To play with this one, we can run it and move around either cube. It will move from us to red no matter where they are. You might notice two things: 1) it moves faster as they get further apart. This is because we're moving by 1% each time. 1% of a longer line is a larger amount. And 2) the moving green cube snaps when we move either end. That's because our math locks it to being exactly on the line – not on purpose – that's just the easiest way to do it.

This next one is sort of a new rule: multiplying an arrow by a negative number flips it around.

I'd like the green cube to hide behind us, from the red cube. That seems tricky, but think of it like this: we already know how to put it 1/10th of the way towards red. We can compute that offset, times -1:

```
// In Update() for green cube to hide behind us from red cube:

  Vector3 toRed = redCube.position - transform.position;

  // new part. Notice the negative:
  greenCube.position = transform.position + toRed * -0.1f;
}
```

As before, we can move either end cube and watch the red one move to stay hidden. Since we're using 1/10th the distance it hides further behind us as the red cube moves further away. That's not intended – it just works out that way.

1.2.1 Camera vector positioning

In this section, instead of placing a cube we'll place the game-camera, which looks much cooler. It's also a neat example of how once you know vector math, you can figure out lots of things that seem very different.

To really see how this works, our object needs to looks different from different angles – like a cow, of a collection of different-sized boxes if you don't have a cow.

A problem with these is that we don't know how to aim the camera yet. Since they'll all put the camera above us, we'll need to pre-rotate it to face down.

This code puts the camera above us, and a little ahead. For fun I made the offset by adding the forward and up shortcuts, but it's the same as regular (0,20,2):

```
public Transform theCamera; // drag in a link to Main Camera

void Update() {
  Vector3 playerToCam = Vector3.forward*2 + Vector3.up*20;
  theCamera.position = transform.position + playerToCam;
}
```

By itself, this isn't very exciting – the camera snaps to a spot. The neat part is that it will track us. If we had scenery and were bouncing and spinning around, this code keeps the camera in a steady position.

We can change that code a little to get a zoom. Instead of adding all of playerToCam, we'll add a percent. That's nothing new. We'll use the arrow keys to change the percent (which is new):

```
  ...
public float zoomPct=1; // should be 0-1, as a percent

void Update() {
  Vector3 playerToCam = Vector3.forward*2 + Vector3.up*20;
  theCamera.position = transform.position + playerToCam*zoomPct;
  // " *zoomPct " is the new part

  // arrow keys zoom in/out:
  if(Input.getKey(KeyCode.UpArrow) zoomPct-=0.01f;
  if(Input.getKey(KeyCode.DownArrow) zoomPct+=0.01f;
}
```

The important line is no different than C=A+B*0.5. It's pretty slick how 0.5 is really the zoomPct variable, which the user can change. But it's still "position plus percent of an offset".

An obvious bug is that nothing keeps the zoom in range. We could add IF's to limit zoomPct between 0.2 and 1. But right now nothing stops us from a negative zoom, which takes the camera through and past us.

Another common tweak is making it so we don't zoom straight in. We might zoom to a spot slightly ahead of the player. We can change our equation to look like: C=A+B+C*pct. B is the offset to the zoom point, and C is the "zoomable" arrow:

```
Camera
  | offset#2, zooms in/out
  |
  / <- the zoom point
 / offset#1, always this size
Player
```

I worked out some different numbers, for looks, but otherwise the code is about the same (leaving out the parts that didn't change):

```
  Vector3 toZoomSpot = Vector3.up*8+Vector3.forward*3;
  Vector3 zoomArrow = Vector3.up*10;

  theCamera.position =
     transform.position + toZoomSpot + zoomArrow*zoomPct;
```

If you see it in action, it looks much nicer than a straight-in zoom.

1.2.2 Speed vectors

Instead of moving a cube along a line made between 2 points, we can give a cube a personal speed offset. In our minds that's an arrow sticking out of that cube, showing how much it moves per second.

It seems funny having it be per-second. It would be easier if it was the amount to move each update, but per-second makes lots of other things easier.

This code lets us enter a velocity vector for the green cube (the script is still on our base cube, which just sits there):

```
// We want to move at 1 arrow per second:
public Vector3 greenMv;
// enter anything. Ex: (0,0,2) is "slow forward"

public Transform greenCube;

  // in Update (assume we know it runs 60 times/second):
  greenCube.position += greenMvArrow/60;
```

I think `+=` works really well here – it's saying to change green's position by a small offset, which is a pretty good definition of moving. Dividing `greenMvArrow` by 60 is because I'm assuming Update runs 60 times/second.

For fun, let it run, then change greenMv to 000. It will stop – it's "moving" at a speed of 0. Enter -1 for x and it will slowly drift left. It won't snap-change to a new position, like it did with the old "line" movement.

For more fun, add `greenMove*=0.99f;` anywhere in Update(). It will gradually drift to a stop as the movement line shrinks.

A note on framerate: I'm assuming Update runs 60 times a second. It often won't, which means it won't run at the speed we set, but that's not a problem for us playing around. If you know about the `Time.deltaTime` trick, that's the real way to be sure.

1.2.3 Averaging points

An average of two points looks and works just like a regular average. `(A+B)*0.5f;` gives you a point exactly between A and B. It works no matter where they are.

In the game board example we can average the two bottom corners to get the bottom middle: `bottomMiddle = (cornerLL+cornerLR)/2;`.

Or, sneakier, the center of the board is the average of two diagonal corners: `Vector3 center = (cornerUL+cornerLR)*0.5f;`. That's pretty cool, since it works for tilted boards, too.

We can also find the average using our old point+arrow*percent method. This also computes the bottom middle:

```
Vector3 acrossArrow = cornerLR-cornerLL;
Vector3 bottomMiddle = cornerLL + acrossArrow*0.5f;
```

As we know, writing it out lets us use any percent. Average is just a shortcut for Position+Offset*0.5.

1.3 Math review

Above, most examples had one new rule or trick. Here they are written all together:

- A point plus an offset makes a point. Think of it as starting at the point and following the arrow.
- An offset plus an offset makes another offset. It's like putting the arrows end-to-end, then drawing one big straight-line arrow.
- A point minus another point makes an offset – an arrow from the second point to the first.
- An offset times a number, like `A*2`, is another offset – it scales the arrow.
- A point plus a point is junk. A point times a number is also junk. Averaging two points, `(p1+p2)/2`, is an exception.
- For an offset `-A` is like flipping the arrow to point the other way.
- It looks better to have the point come first: point+offset. But the order won't matter.

I think of these rules when things break. Suppose `sq.position=A+B+C;` is working wrong. Check whether A counts as a position, and B and C are offsets. That's the only way the math makes sense. If something with `B*2` works funny, check "does B count as an arrow?" and "am I wanting to double how long it is?"

Here's a picture of why adding together two points, p1 and p2, makes no sense:

```
     p1+p2(B)


A           p2
                p1              p1+p2(A)
                    B
```

Suppose A is the real (000). p1+p2 would be way over to the right. That's not a spot we'd care about. If the real (000) were at B, p1+p2 would be in a totally different place, up above, which is also not a useful spot.

In contrast, offset math is always good. o12=p2-p1 is always the arrow between them. p1+o12/2 is always halfway between them, and so on.

Another fun error is accidentally using an offset by itself, forgetting to add the starting point. That's like starting from (000), which you never want to do:

```
Vector3 offset = p2-p1;
greenBall.position = offset/2  // <- oops
// meant to write: p1 + offset/2
```

A picture of the problem, where either A or B is the origin. We should be between p1 and p2, but aren't:

```
oa
  A           p2
                  p1     ob
                           B
```

`offset` is just a short line going up and left. Since we forgot to add p1, we'll be up and left of the origin, which could be oa or ob, depending where the origin is.

If you get motion and placement going the correct way, but in the wrong spot, check that you added the starting point.

It's also easy to flip the arrow direction by mistake. B-A and A-B are the same arrows, but backwards (from A to B, or B to A). This code tries to put the green cube between p1 and p2, but accidentally starts at p1 and moves away from p2 with a backwards arrow:

```
Vector3 toP2 = p1-p2; // oops! backwards arrow

greenCube = p1 + toP2/2;
// start at p1, but moves away from p2 instead of towards it
```

Chapter 2

Local and Global coordinates

Occasionally it can be helpful to pretend an object has it's own personal coordinate system: (000) glued to the object, xyz spinning with it. We can imagine that grid overlaid on the real coordinates.

Since we have two xyz's, we'll say the real one is *global*, and the one on us is our *local* coordinate system. Obviously there's only one global, but everyone has their own personal local coords.

It turns out that if we know one, we can figure out the other. Global to Local and back. This is great. It means we can place an object using our personal xyz, if that's easier. Then convert into the real xyz. With any luck, we can make the system do that for us.

What often happens is you have a rotated object, often rotating. It could be facing any direction at any time. And the math for "go to my right" and "behind me" and so on is hard – piles of trig hard.

"Try it in local coordinates" can be a magic spell. It says to work out the problem as if you were always facing forward, and also pretend you're always at 000. Our trig problem often turns into grade-school math. Solve that and you're done. Really. The last step is to tell the computer you were using local space. It will account for it, like magic.

2.1 Local/global translate tools

Before math and coding, let's just play with local axis in the Unity3D editor. Even though they aren't real, the tools in Unity let you use them.

Get in Scene view, select any rotated cow and pick the Translate tool (on top: it's the Crossed arrows, between the Hand the the Circle arrows). Red, green and blue arrows should come out of it, aligned with the real x, y and z axes

A few buttons to the right, you should see a button that says Global. It's a toggle. Tapping it flips between Local&Global. If the object is rotated, you should see the xyz arrows snap to a new spot, then back.

On the Local setting, spinning the object spins your local arrows. As you might guess, +z is always out your front, +x is always your right side. If you turn upside-down, your local +y goes down (which is what up-side-down means).

If you drag a rotated box using the xyz local axes, you'll see it move in a nice grid. It feels like a regular xyz system. Technically, it's all fake. When you move on just 1 local axis, the Inspector shows x, y, and z all changing at once. It's re-calculating the global coords for you. The local axes aren't "real-real". But we can use it and it works great. It's real enough.

Pretty much any 3D program has options for Local/Global. Many users who aren't so hot on math have an excellent understanding of Local and Global axes. It just takes practice.

A note: the colors are standard. Everyone lines up rgb and xyz – the red arrow is always x, green is y and blue is z.

2.2 Using your local axis in code

Since they're so useful, Unity provides your local-axis arrows in code: `transform.right`, `transform.forward` and `transform.up`.

The way I remember is that `transform` means me, and holds my position and spin. So `transform.right` is my right. `Vector3.right` is global right. It never changes, but transform.right has to be calculated based on the spin.

We can use them like regular offsets. This puts the red cube 3 units to our right:

```
public Transform redCube;

void Update() {
  redCube.position = transform.position + transform.right*3;
}
```

Since this uses *our* right, spinning us will move the red cube, keeping it always 3 units right of us.

As normal, we can combine these offsets. This puts the red cube 2 in front and 4 left of us. Again, spinning shows that it's our front and left, based on our facing:

```
redCube.position = transform.position +
    transform.forward*2 + transform.right*-4;
```

Unity didn't give us transform.left, since by now everyone knows left is negative right.

We can improve our old movement code by using these new arrows. This line in Update moves us the way we're facing:

```
transform.position += transform.forward * 0.01f;
```

You can check that it always uses the our current forwards by running it and spinning a little. Notice how I was too lazy to figure out speed-per-second and then divide by 60. Instead I just guessed 0.01 for slow, but not too slow.

The rest of these are examples of how everyone has their own personal local coordinates. Our script can look at other peoples', and use them like regular offsets.
This code moves the red and green cubes along their forwards (face them in various directions to check):

```
redCube.position += redCube.forward * 0.01f;
greenCube.position += greenCube.forward * 0.01f;
```

As you might guess `transform.forward` is my forward, `redCube.forward` is her forward, and so on. You can ask any Transform, such as redCube, for any of their local axes.

This next example is basically a mistake, but legal. We're trying to move ourself forward, but accidentally used the red cube's forward, not ours:

```
transform.position += redCube.forward * 0.01f;
```

It's pretty weird. We'll move in a funny direction. Spinning the red cube, which isn't moving, will change the direction we move. it's a little like a remote-control steering wheel.

Here's another garbage example, where we position the green cube based on the red and blue forward arrows:

```
greenCube.position = transform.position + redCube.forward*3 +
  blueCube.forward*4;
```

If both are pointing in the same direction, green will be about 3+4 units away. If facing in opposite directions, they mostly cancel out. It's like the green cube is on an invisible 2-part robot arm, controlled remotely by the red and blue cubes' spin.

Here's one where it makes sense to use other people's arrows. Each cube is 2 units in front of the previous one, forming a chain:

```
 redCube.position = transform.position + transform.forward*2;
 blueCube.position = redCube.position + redCube.forward*2;
 greenCube.position = blueCube.position + blueCube.forward*2;
```

The math in all three lines uses the same cube's position and forward. "2 in front of me" makes sense. The chain is fun – if we rotate a cube, everything after it tracks the spin. That looks pretty cool if we do it while running.

Now back to movement code. We can adjust the old code where we shot out a green cube over and over. The movement line can now be our personal forward. The rest of the code is the same. As with all the others, it looks best if the object is rotated, to show how it always goes forward:

```
float pct=0; // from 0 to 1
public Transform greenCube;

void Update() {
  // the new line:
  Vector3 moveLine = transform.forward*8; // 8, since why not

  greenCube.position = transform.position + moveLine*pct;
  pct+=0.01f;
  if(pct>1) pct=0;
}
```

It starts from our origin. For a cow, that's down by the feet, which isn't so nice. We can bring it higher up, maybe out the mouth, by adding an extra transform.up offset:

```
void Update() {
  Vector3 moveLine = transform.forward*8;

  greenCube.position = transform.position +
      transform.up*2 + moveLine*pct; // <- new, adding up*2

  pct+=0.01f;
  if(pct>1) pct=0;
}
```

It's looking a little complicated, but all we're doing is adding 2 offsets. We always go 2 in our up, then go a variable amount in our forward. The zooming camera did the same thing:

```
                  transform.forward*??
                ----------------green
 transform.up*2 |
              COW
               o
```

It can feel strange not having `transform.forward` come straight out of us. But it's just an arrow representing our personal +z. Moving our up, then our forward feels natural enough.

2.3 Other Unity commands and local coords

This uses the built-in `Translate` command to move slowly in our forward direction. The inputs are x, y, and z. But see if you notice the strange part:

```
void Update() {
  transform.Translate(0, 0 , 0.01f);
}
```

The new/strange/nice part is that we didn't have to do anything funny to get our local z. Translate was written for when you want to move along your arrows. You give it the amounts on them, and it does the rest. Technically, Translate expects the inputs to be in your Local Space.

Suppose we want to move forward and a little up, at a speed of z=0.01 and y=0.002. Written both ways, the Translate one looks nicer:

```
void Update() {
  // old way:
  transform.position += transform.forward*0.01 + transform.up*0.002f;

  // the version with Translate has far less nonsense:
  transform.Translate(0, 0.002f, 0.01f);
}
```

People like to think in local coordinates, and expect to have built-in functions using them. Translate is a useful, simple example of that.

But now we have one more thing to worry about: xyz's can be in global or local. Local is so useful that it's worth being a little confusing. And in most commands it's obvious which they want.

On to the next command. Anything with a rigidbody and a collider will normally fall and bounce around. Unity also has two commands to shove, using global or local coordinates. `Relative` isn't a technical term, but it clearly means local:

```
rb.AddForce(0,0,4); // real +z, like a plunger
rb.AddRelativeForce(0,0,4); // local +z, like a thruster
```

The interesting thing is how (0,0,4) can mean Local or Global, depending on how you use it.

Similar side-by-side commands are common when either way might be useful. First decide whether the motion should be described using your personal arrows, or the real xyz. That tells you which command to use, then work out the numbers.

Back to Translate. The short version which we saw is automatically in Local. It has 2 longer versions where you can select Local/Global using the 4th input:

```
transform.Translate(0,0,1, Space.Self); // local
transform.Translate(0,0,1, Space.World); // global
```

Self and World words are more informal terms. Some game designers prefer them to Local and Global.

There's no special reason why AddForce is 2 different commands, while Translate is one with 2 options. But it doesn't really matter. The main thing is knowing there can be a local and global version. If we want to give the directions using our personal arrows, we can. We just have to find the command that wants Local Space.

2.4 Childing, `localPosition`

Suppose we have two cubes side-by-side, off somewhere. One might be at (3,0,6) and the one next to it at (4,0,6). If we go to the list of names and drag the second cube into the first, it becomes a child. It's locked to the first cube, tracking it. Nothing special so far.

But another interesting thing happens. The numbers for the second cube's position snap to (1,0,0). It doesn't move, and it's not at (1,0,0), But the Inspector shows those numbers. If we move and spin the first parent cube, the child cube moves to track it, but always displays position (1,0,0).

Moving the second cube does another strange thing. Suppose we pull it further right so it says (2,0,0). Now it tracks the parent based on that new position. The position numbers are still locked, now to (2,0,0):

```
        c2 child, at parent's local (2,0,0)
       /
   \  /
    c1 parent facing north-west
```

The rule is: objects with parents use the parent's local space. "Locked 2 spaces right of our parent" is just another way to say "(2,0,0) in our parent's local space". The Inspector is showing our local position in our parent's space, and allowing us to adjust it. The only very minor problem is how the label misleadingly still says Position.

A fun local-space parent trick is snapping one object to another. First make yourself a child of the thing you want to go to. Then enter 000 for your position.

That puts it 000 away from the parent. Neat, right? Finish up by unparenting.

Another fun child trick is sliding yourself based on another object. Say you want to slide yourself along a very tilted wall. Just make yourself a child of it. Now sliding your Inspector xyz's uses the wall's arrows (to slide a number, click just to the left when the cursor turns to a slider-symbol, then slide). x and y slide you perfectly along the surface of the wall, no matter how the wall's been rotated.

You can use the child-trick to play a fun game: get a cow and a ball, neither a child of the other, and spin the cow. The game is to place the ball exactly in front of the cow, 3 units away. Once you think you have it, check your work by making the ball a child of the cow.

The numbers you see will instantly snap to local. The closer you are to (0,0,3), the better you're lined up. If x is almost 0, you got the left/right almost perfect.

Of course, you could get it perfect by making yourself a child, typing in (0,0,3) and un-childing. But playing it like a game is educational.

In code things are different, a little nicer. `transform.position` is always your real-world position. Even children can use this as normal. If you're a child, `transform.localPosition` is where you are in your parent's local space.

Here's a crazy-looking script to run on a child. Just drag yourself into anything and tilt it. This code moves you along your parent's forward:

```
void Update() {
  transform.localPosition += Vector3.forward * 0.01f;
}
```

The key is that `localPosition` is in the parent's local coords. Adding to z is the same as adding to Inspector z, which is local in the parent. Vector3.forward "feels" global, but it's just (0,0,1). What it does depends on where we put it.

It's another case where we can choose Local or Global based on the command we choose:

```
  // [position] is real xyz:
  transform.position += Vector3.forward * 0.01f;

  // [localPosition] is parent's local xyz:
  transform.localPosition += Vector3.forward * 0.01f;
```

This next one is a little silly, showing more localPosition use. Assume the green cube and us share a parent. This puts the green cube where we are, except mirrored on our parent's x-axis:

```
// greenCube needs to have the same parent as us:
public Transform greenCube;
```

```
void Update() {
  Vector3 localPos = transform.localPosition;
  localPos.x *= -1;
  greenCube.localPosition = localPos; // our pos, with x flipped
}
```

To test, we can slide ourself around and see green mirror us – the same spot on the cow, but on the other side.

We can redo the “fire a cube out of the cow’s mouth” example using localPosition (the cube has to be a child of the cow). Instead of a percent, this will use meters going from 1 to 8. But otherwise it’s the same:

```
public Transform greenCube; // assume this is our child
float zDist=1; // in units. goes up to 8

void Update() {
  // put green 2 above and z forward:
  greenCube.localPosition = new Vector3(0, 2, zDist);

  // move zDist from 1 to 8:
  zDist+=0.02f;
  if(zDist>8) zDist=1;
}
```

`localPosition = new Vector3(0, 2, zDist);` is the fun part. It’s a clear, short way to say “2 up and z forward on my parent”.

2.5 Looking at the numbers

This section is very optional. It’s for if you aren’t happy unless you see the numbers and can check them out.

This simple program shows values for your 3 local arrows:

```
// Inspector copies of your local axes:
public Vector3 right, up, fwd;

void Update() {
  right = transform.right;
  up = transform.up;
  fwd = transform.forward;
}
```

If we have no spin, right will be (1,0,0), and so on – our local axes start out lined up perfectly with the real ones. I didn't say this before, but they're also length 1, just like the real ones.

If we spin 90 degrees clockwise on y, right will be (0,0,-1). That's still normal – our right is the real backwards. But tilting 45 degrees gives some strange numbers. `fwd` will be (0.707, 0, 0.707).

That's correct. It's length 1 going diagonal. It turns out 0.707 is the cosine of 45 degrees, and is also 1/2 of the square root of 2, and the pythagorean theorem says the whole thing is length 1.

Spinning 30 degrees makes `fwd` be (0.5, 0, 0.86). That's also length 1, and also the sin and cosine of 30 degrees.

If you tilt on y and x, you'll get even stranger values for x and y and z. But the math says they check out.

Finally, this is one more thing that might be bugging you: the manual says `transform.forward` is in *world* space. What?? We use it for local space positioning, so how can it be world space?

It's about which arrows the numbers are meant for. (0.707, 0, 0.707) is how to make a diagonal arrow, using the real x, y, and z. `transform.forward*3` is like a 2-step process: we think "3 forward local", which is (0,0,3); then convert using the arrow.

2.6 Errors

Mixing local and world axes in the same math usually gives junk, for example: `transform.right*3 + Vector3.right*2`. That's 3 to out personal right, then 2 on the real right. Depending on our spin, they could mostly cancel out, add, or anything in-between. It's not illegal, but it's almost never useful. It's usually a sign you made a mistake.

`Vector3.up` vs. `transform.up` can be an exception. Suppose you want a label above someone and a little behind. `transform.up*3 - transform.forward`. seems correct. It works fine when they spin. But if they tilt or lean, then `transform.up` tilts and pulls the label too much. The label looks better if we go straight up from the feet: `Vector3.up*3 - transform.forward`.

The opposite problem can happen. You place a hat using `Vector3.up*1.4f`. That works perfectly at first, since they rarely bend over. But when they do, the hat is floating above their feet.

The most confusing errors are double-conversions. If something expects local coordinates, you can't use `transform.forward` and its friends. This is legal but gives wrong movement:

```
transform.Translate(transform.forward);
```

`transform.forward` converts into the values we want. But Translate doesn't know that. It assumes they're unconverted locals and does it again. It's a little bit like converting feet to inches twice in a row.

Hopefully this also feels redundant. We use commands taking Local coords so we don't have to use any other tricks. We like how `Translate(0,0,1)` means our forwards.

Double-conversions can be hard to spot during testing, since the wrongness of the result can be a lot or a little, or none if you aren't rotated.

Another double-convert error which is pretty much the same: we're setting our child's localPosition, which wants Local coords:

```
// red is our child
red.localPosition = transform.forward*2; // a bad double-convert
```

To go 2 forward we should have used just (0,0,2). localPosition will handle the rest.

The same error with AddForce:

```
rb.AddRelativeForce(transform.forward*6);
```

We chose to use RelativeForce so we could use (0,0,6). Adding the transform.foward messed that up. Either of these would work correctly. Obviously the second one is nicer:

```
rb.AddForce(transform.forward*6);
rb.AddRelativeForce(new Vector3(0,0,6));
```

2.7 Longer space-fighter example

There's nothing new here, but it's fun to see a few things used at once.

As we all know, X-wing fighters fire a blast from each wing tip, angled inward a little so that they meet after 50 meters:

```
   |= laser . . .
   |             . . . .
>OOOC                   X bam!
   |             . . . .
   |= laser . . .       transform.forward*10 - transform.right*2
```

Pretend we have some tiny rigidbody balls as prefabs, ready to be used as laser bolts. As a quick review, anything with a rigidbody moves by itself. We set the `velocity` once, at the start, in units/second.

This code handles placing the balls at our wingtips:

```
public Transform ballPrefab;

void Update() {
  // the space key fires:
  if(Input.GetKeyDown(KeyCode.Space)) {
    // wing tips are 5 sideways and 2 ahead of us:
    Vector3 toRightWing = transform.right*5 + transform.forward*2;
    Vector3 toLeftWing = transform.right*-5 + transform.forward*2;

    // spawn and place them:
    Transform bRight = Instantiate(ballPrefab);
    Transform bLeft = Instantiate(ballPrefab);
    bRight.position = transform.position + toRightWing;
    bLeft.position = transform.position + toLeftWing;
```

The only tricky part, for me anyway, was the wingtip math. On my first try I flipped the entire left equation, so the ball was 5 left and 2 *backwards.*

Next we set the ball's speeds using the `velocity` built-in. They should fly forward and a little inward:

```
    // set the speed of the 2 balls we just made:

    // right ball moves fast forward and a little bit left:
    bRight.GetComponent<Rigidbody>().velocity =
      transform.forward*10 + transform.right*-2;

    // left ball has a small drift going right:
    bLeft.GetComponent<Rigidbody>().velocity =
      transform.forward*10 + transform.right*2;
  }
}
```

To test this, it looks nicer if you uncheck the balls' UseGravity box. With gravity turned on, they'll probably fall below the camera before meeting in the middle.

2.8 Intro to local space theory

You may have noticed how `transform.forward*2 + transform.right` are basically hacks. They work great, but they aren't they way we like to think about local space.

The best way to do this stuff is to think completely in local, at first. If we want to be 2 ahead and 1 right, we write (1,0,2) and remember it stands for local. Then we either find something that wants local, or convert at the end.

For example, if we want a child at (1,0,2) from us, `c1.localPosition = new Vector3(1,0,2);` is great. It says exactly what we were thinking, without being very long. `c1.position = transform.position + transform.forward*2 + transform.right` will work, but it's basically hiding "local (1,0,2)" from us.

Likewise `rb.AddRelativeForce(new Vector3(0,1,10));` is the nicest way to say that we want a local push, of this xyz local amount.

So far, there's no good shortcut for placing a non-child using our local coords. We'll get one later, but just for fun we can make it now.

We've seen the equation over-and-over: to be (1,0,2) from us, start where we are, and multiply out direction arrows. Here's a function doing that:

```
Vector3 toLocal(Transform tt, float x, float y, float z) {

  Vector3 pos = tt.position; // start where we are:

  // move x,y,z along our local arrows:
  pos += tt.right*x
  pos += tt.up*y;
  pos += tt.forward*z;

  return pos;
}
```

Using it to place the green cube (1,0,2) from us:

```
  greenCube.position = toLocal(transform, 1,0,2);
```

It almost seems like cheating, but that's what functions are for. We can see it's taking transform.right and transform.forward*2, and adding it to our position.

It's also a complete sum-up of the entire chapter. "(1,0,2) from us" is a fine way to think, the computer can do it, and it's even easy to read (once you know about the local trick).

Chapter 3

Direction & Length

Scaling a vector is a pretty neat trick. We can add half an arrow, or double an arrow, or use 0-1 to slide along the length of an arrow. But that trick can't give us distances.

If we want to travel 5 units along an arrow, or move along it at 2 units/second, we need more math.

The first trick is getting the length of an arrow. The second is getting a length 1 version of an arrow. After a bit, we'll start thinking of any arrow as really being those two parts – the direction, and how far.

3.1 Magnitude/distance

The way to find the distance between two things is to make an arrow between them, then measure the length. All distances are really measuring how long an arrow is, which is officially called its *magnitude*.

To find the distance between us and the green cube we make the offset arrow, as usual, then measure it:

```
Vector3 toGreen=greenCube.position-transform.position;
float dist = toGreen.magnitude;
```

Distances and offsets can feel similar. An offset gives us the distance over x,y,z. But not the straight line real distance. Where-as distance is a single number – the actual distance – but doesn't tell us how to get there.

Since magnitude simply measures arrows, we can check it by making an arrow out of numbers, then measuring it:

```
Vector3 A = new Vector3(3,0,4);
float dist = A.magnitude; // 5
```

You may remember that answer from school. 3 over and 4 up is a triangle with the arrow as the hypotenuse. It's length 5 by the Pythagorean theorem. That's the math the system is doing for us.

Here are some fun facts about distance and magnitude:

- Distance is always one positive number. Negative distance makes no sense. If it's 3 miles from my house to the quarry, it's 3 miles from the quarry to my house, not negative 3. Offsets can be negative – (3,0,0) to the quarry and (-3,0,0) back.
- The subtraction order doesn't matter for computing distance. `A-B` is a backwards arrow from `B-A`, but they're each the same length.
- Logically, `magnitude` is for offsets, not positions. `(transform.position).magnitude` isn't an error, but it thinks of our position as an arrow from (0,0,0). It says how far you are from (000), which isn't a helpful number.
- **magnitude** is the official mathematical term for the length of an arrow. Not just for computers, but for real math.
- Not having parens after `A.magnitude` is a (required) shortcut. It's really a function call running an equation.

A fun distance problem is getting a "flat" distance. We often want to know how far two things are apart on a map, not counting hills:

```
                   _B_
                  /
                 / hill
__A_____________    o
  |- we want this --|
```

The trick is to create the arrow we want to measure. We can find the B-A arrow, then change the y part of it to 0:

```
Vector3 arrow = B-A;
arrow.y=0; // now it's from A to the dot
float dist = arrow.magnitude;
```

Since distance is never negative, `if(dist<3)` is way to check if you're close enough. If something were 10 units behind you, the distance would be a positive 10 and the "close enough" check would fail.

Here's a basic "find nearest enemy from a list" test using a simple distance compare:

```
float shortestDist=99999;
int nearest=-1;
```

```
for(int i=0; i<Things.length; i++) {
  float dist=(Things[i]-transform.position).magnitude;
  if(dist<shortestDist) { // <- simple less-than distance check
    nearest=i;
    shortestDist=dist;
  }
}
```

3.1.1 Direction arrows

It's often good enough to have an arrow pointing at the target. We don't care how long it is, only the direction it aims. In fact, if we have an arrow pointing from A to B, we often prefer it to be length 1, no matter how far apart they are. To get to B, we'll go that direction until we get there.

`transform.forward` is a great example of a direction arrow. It's an offset, but not really. Offsets tell us how to find one specific spot. transform.forward is a way to walk forward as much as we need, in our forward Direction.

We can think of a direction vector as half of an offset. Like an offset, it has to start from a position, but it doesn't go to any particular spot yet. We have to also know the distance.

Some functions say they take a Direction as an input. They're talking about this sort of direction. They want a Vector3 which is correctly aimed, but just any length. For a direction input, (2,1,0) and (4,2,0) are the same. But (2,-1,0) or (1,2,0) aren't – they aim in a completely different direction.

Raycasts and Direction

The built-in raycast function takes a direction as an input, giving a nice example of the concept. A Raycast is a standard 3D game trick. In the game world you draw an imaginary line, as far as you want, and check whether it hits any cubes or cows. It's used to see what your laser hits, or check if you can move in a certain direction. Technically, a line going in only 1 direction, not both, is called a ray, thus the name raycast.

Here's a raycast that checks for obstacles 5 units in front of us:

```
Vector3 lookDir = transform.forward;
if(Physics.Raycast(transform.position, lookDir, 5)
  print("forward 5 is blocked");
```

Hopefully the 3 inputs are obvious: starts from us, in our forward direction, for 5 units. `transform.forward` is just the direction. It's length is 1, but that doesn't matter since it's a direction. The distance is all by itself in the next variable.

Suppose we want to look for obstacles mostly ahead, but with a small tilt right. A cheap way of getting a tilt is to think of it as a slope – 1 right for every 12 forward.

To make it more interesting, we'll to look in that direction for 7 steps, then, if it's clear, for 20 steps:

```
// the length doesn't matter, only the angle:
Vector3 lookDir = transform.forward*12 + transform.right;

if(Physics.Raycast(transform.position, lookDir, 7) {
  if(Physics.Raycast(transform.position, lookDir, 20) {
    print("lots of room");
  }
  else print("some room");
}
else print("not enough room -- need at least 7");
```

I think that looks pretty nice. We use the same direction each time, with distances of 7 and then 20. Seeing the distance to all by itself makes it more readable. And letting us make the direction however we want make that first line short and clear.

There are other ways. Sometimes the second input is the ending point. Sometimes it's a real offset going the total distance to check. But the direction + distance method is also common.

3.1.2 Normalized direction

The most useful direction vectors have exactly length 1. We've seen this with `transform.forward`. To walk 1.8 steps forward, we can use `transform.forward*1.8f`. The math term is *normalized direction*, or sometimes *unit vector* (shorthand for "a 1-unit long vector").

This is so common that when something asks for for a direction, most people check whether it needs to be length 1. Often they make it length 1 just in case. Or, to be really safe, convert all of your directions to length 1 as soon as you get them.

There's a really slick trick to take any arrow and rescale it to be length 1. All we have to do is divide by its length. We can get the length using `magnitude`, so the whole thing is too easy. This gets a length 1 arrow from us to the green cube:

```
// this part is a repeat:
Vector3 toGreen = greenCube.position - transform.position;
float dist = toGreen.magnitude;

toGreen = toGreen/dist; // toGreen is now length 1
```

Now `transform.position + toGreen` is exactly 1 away from us, towards green.

The trick works for any arrow. If the arrow is longer than 1, it shrinks. If it was shorter, it grows. It works for backwards aimed arrows, or going along only x. The only thing it won't work for is (000), since that's not a direction.

The way it works is so clever. We know any arrow times 2 is the same arrow, twice as long. An arrow divided by 2 is aimed the same way, but 1/2 as long. So, a length 4.71 arrow divided by 4.71 is going the same direction with length 4.71/4.71 = 1!

The official math term for "make a length 1 version of an arrow" is *normalizing*. If you've seen Normals used to affect lighting in 3D models, this is completely different. There's no relation at all – it's only a coincidence both terms use the same word.

Unity provides a function to normalize. All it does is compute the length and divides by it. We can do that ourselves, especially if we already needed to know the length. But doing it the official way can look nice.

In comes in the 2 standard versions: change yourself, or don't and return the result:

```
A = new Vector3(3,4,0); // sample vector

B = A.normalized; // B is (0.6, 0.8, 0), A is unchanged
A.Normalize(); // A is now (0.6, 0.8, 0)
```

We don't really need the second one, since `A=A.normalized;` does the same thing, but some people like how it looks.

Some normalizing notes:

- (1,0,0) is normalized. It's length 1.
- (0,-5,0) normalizes to (0,-1,0). The negative has to stay, or else it wouldn't be the same direction. (1,0,1) normalizes to (0.707, 0, 0.707). That's length one because of trigonometry.
- If you accidentally normalize something a few extra times, no problem. Normalizing a length 1 vector does nothing. You're dividing by 1.
- If you already know the length, `A=A/len;` is faster than `A.Normalize()`. But many people don't know (or do know but don't remember) the division trick. Writing the official commands can make code easier to read.
- The one thing you can't normalize is (0,0,0), since that's no direction. There's no possible length 1 version of that, since it's not pointing anywhere. The Unity normalize command gives you back (0,0,0). That's not correct, but it's the best it can do.

For real you often get a direction by taking B-A when they're at the same spot. There's no direction to go to B, since you don't need one – you're there.

3.2 Normalized direction + length

When we used `transform.position + transform.forward*1.5f` we were taking advantage of how transform.forward is length one. We could get a position exactly 1.5 units ahead of us. We couldn't do that with regular offsets before. Now we can, by making them into unit vectors.

Suppose we want to walk 2.5 meters towards the green cube:

```
Vector3 uToGreen = (greenCube.position - transform.position).normalized;

Vector3 pos = transform.position + uToGreen*2.5f;
```

The 2nd line looks exactly like the math we did with `transform.forward` and friends, since it is. We're scaling it by 2.5, but since it's length 1, that makes it exactly 2.5 long, towards green.

The first line is just our two old lines combined. (B-A) is the arrow, and normalized makes it be length 1. The u in `uToGreen` is for unit. Everyone knows unit means 1 unit long.

Previously we were able to put a cube behind us, hiding from green, but the best we could do for distance was 10% of the whole line. Now we can put it exactly 3 meters behind us:

```
redCube.position = transform.position - uToGreen*3;
```

Again, it should look a lot like our forward/right/up math, since it is. Starting from us, we're going 3 meters away from green.

3.2.1 Using both

The full trick is splitting the offset into 2 parts: the unit vector, and distance. Together, those are better than just the full offset. Our old code doing this, all in one place:

```
// getting ready to solve any hard offset problem with uToGreen and dist:
Vector3 toGreen = greenCube.position - transform.position;
float dist = toGreen.magnitude;
Vector3 uToGreen = toGreen.normalized;
```

Fun fact: `uToGreen*dist` is the original arrow.

With these two variables we can place the red cube 2 units past the green one (it will appear to be hiding from us, behind green):

```
// us ---------------- green -- red

redCube.position = transform.position + uToGreen*(dist+2);
```

Pretty slick, right? From us, we walk `dist` units along the arrow, which puts us on green, then 2 more. I almost hate to ruin it by saying that an easier way is to start at the green cube and walk 2 more: `greenCube.position + uToGreen*2`.

Or we can leave it between us, but almost to green – like green is using it as a shield. We can either walk `dist-3` units towards green, or start at green and walk backwards 3 units:

```
// us ------------------ red --- green

redCube.position = transform.position + uToGreen*(dist-3);

// or shorter version, start at red, walk 3 back to us:
redCube.position = greenCube.position - uToGreen*3;
```

With our new unit vectors and lengths we can make the red cube travel from us to Green, better than it did before. Now we can make it start 1 unit ahead of us, stop when it gets within 1, and move at a consistent speed:

```
void Update() {
  // pretend we have uToGreen and len

  dist+=0.05f; if(dist>len-1) dist=1; // goes from 1 to len-1
  redCube.position = transform.position + uToGreen*dist;
}
```

The first line controls the actual distance away from us. That makes it super easy to start and stop exactly as far away as we want to. The motion is now at a constant speed, which means it finally takes longer to go further.

The second line is basic *unitVector*distance* positioning. Figuring out the correct values for `dist` was all the work.

3.2.2 More throwing with rigidbodies

Using unit vectors to toss rigidbodies is just as easy. We did it before, in the spaceship, using `transform.forward`. Now we can do it with `uToGreen`.

I'll pretend the red cube has a rigidbody. The space key will put it 2 units ahead us of, then launch it towards the green cube:

```
void Update() {
if(Input.GetKeyDown(KeyCode.Space)) {
  // find the spot 2 units in my forward:
```

```
  Vector3 rPos = transform.position + transform.forward*2;

  redCube.position = rPos;

  // motion is long line from red to green, at speed 10:
  Vector3 toGreen = greenCube.position - rPos;
  Vector3 uToGreen = toGreen.normalized;
  redCube.GetComponent<Rigidbody>().velocity = uToGreen*10;
}
```

Once you know that velocity is set once, then moves us automatically, and it's in units/second, and that `uToGreen` is a unit vector, then `velocity = uToGreen*10` should be simple enough.

3.3 Looking at the numbers

If you look at the numbers for distance, they can seem funny. One way to solve this is not to look at them. But if you can't help yourself, here are some notes:

When you have a long length and a short one, the short one counts for almost nothing. For example (10,1,0), has a length of only 10.05.
To see how that makes sense, imagine the triangle: 10 over and 1 up. The diagonal is clearly just a little more than 10 long.

Even when the numbers are close together, the answer is smaller than it seems. (5,4,0) has a length of 6.4. In 3D, the numbers are even shorter. The arrow (3,4,5) has a length of only 7.1. That's the longest, 5, plus not much more for the diagonals.

If you estimate distance between two points in your head, you can think of all differences as positive. For example, comparing (10,10,10) to (2,13,9). All that matters is: 8 away, 3 away and 1 away. The distance will be 8 plus a little more.

Normalized arrows have the same funny-looking math as `transform.forward`. Any 45-degree unit vector looks like (0.71, 0.71, 0). If you normalize (1,1,0), that's what you get. A unit arrow at 30 degrees really is (0.6, 0, 0.8). Almost all unit arrows are wrong-looking numbers like that.

The actual equation for distance is the 3D pythagorean theorem:
$x^2 + y^2 + z^2 = D^2$. In computer code:

```
float dist = Mathf.Sqrt(x*x + y*y + z*z);
```

Chapter 4

Rotations

This section is about setting basic rotations: how to declare a rotation variable, and ways to make and think about them. Later, in another chapter, we'll be able to add them, take fractions and the rest.

Our cubes are no good for examining rotations since we can't tell the front from the top and sides. A cow, pointing along +z, is fine (but it can't be tilted to aim that way – its 000-rotation must be +z). Otherwise we can make something. Take one of our cubes and child something unique on top and in front. Maybe a small sphere on top, and forward-aimed cylinder in front:

```
// 3-part testing object, if we don't have a cow model:

 |       o        <- small sphere for a hat
+y       H ===   <-- cylinder as a nose
  +z ->
      Side view
```

Rotations can be imagined in lots of ways. They can be a direction arrow and a roll – like aiming a telescope, then spinning to adjust the viewfinder. Or the old xyz 0-360 method. Or a rotation can be a single diagonal line going through your origin, rotating 0-360 degrees around that one line.

Like vectors, rotations can be offsets. The simplest rotations tell us which way to face. But an offset-style rotation is meant to add to another rotation, or find how far apart two rotations are.

But the various ways to create simple direction-style rotations will be enough to fill this chapter.

4.1 Quaternions

It seems natural to think of rotations as x,y,z, 0-360. That's what the Inspector shows us, and how the 3 circles work using the rotation tool. But those are just tools giving us that view. No one has really stored or used rotations that way for decades. We use a better system called quaternions. The same way `transform.position` is a Vector3, `transform.rotation` is a quaternion.

Quaternions have only one drawback: the numbers in them don't make sense to humans. But that's not a problem since we don't need to look at the numbers. Think of quaternions as a struct with built-in functions doing everything we need.

The simplest way to use a quaternion in a program is saving and restoring a spin. We can declare a quaternion – which is a rotation-holding-variable – and copy our starting rotation into it. Later we can copy it back:

```
Quaternion savedStartFacing; // this is a rotation variable

// copy our starting rotation into a variable:
void Start() { savedStartFacing = transform.rotation; }

void Update() {
  // spin yourself by hand, then press "a" to restore:
  if(Input.GetKeyDown("a"))
    transform.rotation = savedStartFacing;  // copy it back
}
```

Quaternions looks strange, and it feels weird to copy our rotation into one, but this program is really nothing more than `Vector3 savedPos = transform.position;` and then the reverse.

A similar example, to show using quaternion variables, this switches my rotation with the red cube's. If you remember, a standard swap looks like `temp=a; a=b; b=temp;` In this, `temp` is a quaternion:

```
void Start() {
  Quaternion temp = transform.rotation; // save a copy of my rotation
  transform.rotation = redCube.rotation; // red into me
  redCube.rotation = temp; // saved old me into red
}
```

Still not very exciting, since we don't know how to create or change a rotation yet.

For more using quaternions as normal variables, let's save the rotations of red, green and blue in an array of quaternions. Then pressing 1,2 or 3 snaps us to copy that cube's rotation, using a function:

```
public Transform redCube, blueCube, greenCube; // dragged in links

Quaternion[] Spins; // will be length 3 list of colored cube's rotations

void Start() {
  // basic array creation, with quaternions!:
  Spins = new Quaternion[3]; // array of quaternion variables
  Spins[0] = redCube.rotation; // copy into each array slot
  Spins[1] = blueCube.rotation;
  Spins[2] = greenCube.rotation;
}

void copyRotation(Quaternion r) { // quaternion as an input
  transform.rotation = r;
}

// not very exciting. Keys call the function:
void Update() {
  if(Input.getKeyDown("1") copyRotation(Spins[0]);
  if(Input.getKeyDown("2") copyRotation(Spins[1]);
  if(Input.getKeyDown("3") copyRotation(Spins[2]);
}
```

The point of this is that we can use `Quaternion` like any other variable type. An array of them is fine. `Spins[0]` is the first quaternion in the array. We can pass quaternions to functions. They're regular variables.

4.2 Ways to make a rotation

The most common way to make a rotation is to say what you want to look at, and have the computer do the math. That seems like cheating, but we have a computer – why wouldn't we create that command?

After that, we'll take a look at the old xyz, 0-360 method. We don't have to set rotations that way, but it's a perfectly good option. Then there are a few oddball functions for rotation setting.

4.2.1 No rotation

Unity has one built-in for a preset rotation: `Quaternion.identity` is the 000 rotation, or no rotation. For examples:

```
transform.rotation = Quaternion.identity; // face 000 = forward
redCube.rotation = Quaternion.identity; // same

// save a spin, currently it's "no spin":
Quaternion spin1; spin1 = Quaternion.identity;
```

It's the same idea as `Vector3.zero`. It's a rotation of x=0, y=0, z=0. This is the only preset. For example, there's no Quaternion.up;

It's named `identity` instead of zero since identity is the formal math term everyone uses.

We can think of it two ways. For a tree or rubble or a dirt pile, it means to place it with no extra spin. But for a cow or a flashlight – anything which logically has a facing – it's better to say it aims you North with your head up. But only if the model was made correctly for Unity's coordinates, facing on +z.

That's why I made a big deal about facing +z and possibly fixing it with the parent trick. If your cow faces +x, then `transform.rotation=Quaternion.identity` will face it +x. It will look like the command messed up, but you just have a wrong cow.

4.2.2 Look in a direction

The simplest way of making a rotation is using a direction arrow. Recall that a direction is just any arrow where the length isn't important. Suppose you have (0,10,1), which is an arrow aimed up and a little bit forward. We could turn that into a rotation, assign it to us, and we'll be looking up and a little forward.

The command is `Quaternion.LookRotation`. It converts a direction into a rotation. This code aims us in direction (0,10,1):

```
Vector3 dir = new Vector3(0,10,1); // up, a little fwd
transform.rotation = Quaternion.LookRotation(dir);
```

It almost doesn't seem like we did anything. (0,10,1) is an arrow, and we copied it into our direction, sort of. Except arrows and rotations are different. `LookRotation(dir)` did the math to convert to a quaternion.

This next one is basically the same thing but looking 4 ahead and 1 right:

```
transform.rotation = Quaternion.LookRotation(new Vector3(1,0,4));
```

The cool thing is that there are no angles involved at all. Pretend we're on a board and actually want to look 1 space over for every 4 forward. We don't know the angle and don't need to.

This one is very sneaky. It gives us a rotation of 000:

```
transform.rotation = Quaternion.LookRotation(Vector3.forward);
```

It make us face forward, as advertised. But in Unity, forward, with no up or down, is the starting rotation, which is all 0's.

For fun, here's a hacky way to look in a random forward angle. We'll aim 10 forward and -10 to 10 sideways, which gives a random -45 to 45 degrees:

```
float xAmt = Random.Range(-10.0f, 10.0f);
Vector3 v = new Vector3(xAmt, 0, 10); // a random forward arrow
transform.rotation = Quaternion.LookRotation(v);
```

There's no special reason for using 10. It seemed like a round number, and directions don't care about the length.

Now on to the really useful part: aiming at something. It's easy, since we already know how to get the direction from us to anything else by subtracting points. This aims us at the green cube:

```
  Vector3 toGreen = greenCube.position - transform.position;
  transform.rotation = Quaternion.LookRotation(toGreen);
```

It's so slick. LookRotation takes any arrow, and we have an arrow aimed at green. So, LookRotation faces us to green.. Our aiming arrow happened to be the exact length from us to green. It didn't need to be, but it doesn't hurt. If we had a unit vector, it would work as well.

Aiming the red cube at the green one is the same idea:

```
  Vector3 redToGreen = greenCube.position - redCube.position;
  redCube.rotation = Quaternion.LookRotation(redToGreen);
```

Fun fact, if we flip the subtraction order and get a backwards arrow – green to red, then red will be facing exactly away from green.

Our old vector math tricks work here. Suppose we want to look at a spot a little above the green cube. That's just one more line getting that spot:

```
  Vector3 aimPoint = greenCube.position + Vector3.up*1.5f;
  Vector3 toGreen = aimPoint - transform.position;
  transform.rotation = Quaternion.LookRotation(toGreen);
```

Suppose the green cow can see 6 units, and we want to look at what it sees:

```
  // 6 ahead of the green cube:
  Vector3 aimPoint = greenCube.position + greenCube.forward*6;
  Vector3 toGreen = aimPoint - transform.position;
  transform.rotation = Quaternion.LookRotation(toGreen);
```

Or maybe we want to look at a spot between the red and green cubes. We can use the averaging points trick:

```
  // between 2 cubes:
  Vector3 aimPoint = (greenCube.position + redCube.position)/2;
  Vector3 toMiddle = aimPoint - transform.position;
  transform.rotation = Quaternion.LookRotation(toMiddle);
```

If the 2 cubes are on opposite sides of us, the middle will be in a not very helpful place, but we can't fix that here.

Another common look-at trick is getting a "flat" spin, only on y. For example, we want to y-spin without leaning, to face someone who might be standing on a hill or in a valley. After we get the arrow, set y to 0:

```
Vector3 toBunny = bunny.position - transform.position;
toBunny.y=0; // now it's a flat arrow
transform.rotation = Quaternion.LookRotation(toBunny);
```

I like this one since it feels like "how to rotate, but only on y", which is hard. But it's also "how to aim in a direction, but the direction can't have a change in y", which is easy.

4.2.3 Y, local X, local Z rotations

Euler angles logic

In our code we can create rotations by directly filling in the xyz 0-360 values, just as they would appear in the Inspector. To do that we'll need more details on the exact way they work, and a plan for aiming in a certain direction.

Rotations are made to go in order y,x,z. That seems funny, but it works great. To test, use the slide trick in the Inspector: move the cursor to the left of a number until it turns into a little slide icon, then click and drag left/right. The number will scroll up and down.

How the axes act and combine:

- The y-spin is your compass direction. The y-axis runs up/down, so spinning around y is like standing on the ground, turning to face North/South/East/West. Even if you spin x and z first, y is still a perfectly flat compass spin.
- x-spins are elevation, like setting the angle to fire a cannon. They never change your compass direction – only angle you up/down. Normally we want to set x between -90 and 90. Past those – over the top elevation – aims you in the opposite compass direction that y says, which is legal but confusing.
- z-spins never change the direction you point. They always just "roll" you. I like to imagine a flunky villain aiming a gun, playing around with holding it sideways or upside-down.
- Because of this, it never matters what order you change xyz's. If you side-roll on z, that has no effect on how x and y work. Changing them also keeps that z-roll, on whatever new direction we face. x is the same way – compass spinning y drags any previous x-spin with it.

To face in any direction, set y to the compass heading, and angle x to the elevation. Or go in the other order. And then z is always just for fun. Once the cow looks at what it should, you can roll in onto either side or on its back without changing the aim.

A review of the numbers:

Because Unity thinks +z is forwards, y-spins have 0=north, 90=east, and so on. It goes clockwise:

```
        0      y rotation         -90    x rotation
        ^       top view            ^      side view
        |                           |
270 <--    --> 90                     --> 0
        |                           |
        v                           v
       180                          90
```

Because of the left-hand rule, x-rotations feel backwards. +x tilts downward and -x tilts up. Also because of the left-hand rule, z-spins are counter-clockwise. +z spins to the left.

Some theory: xyz coordinate systems aren't done yet until you say the order. Unity chose to use yxz, which is the best one, but xyz, or zxy, . . . are legal. There's no way to make a system where x,y, and z all just spin you around the real axis. It always has to act like a main axis and a chain of 2 connected gears.

Put another way, it's impossible to say which direction (10,80,30) aims until you know the order. If it's in an xzy coordinate system, it will point in a different direction than Unity would.

That whole system – xyz 0-360 and an order to do them in – is now called Euler Angles. As in "the Inspector shows the Euler angle representation of the quaternion".

Euler angles in code

Because we don't store angles directly, we can't just copy xyz degrees into quaternions. But we almost can. We have a function named `Quaternion.Euler` which translates our Euler angles into the proper quaternion values.

This faces us to the right:

```
transform.rotation = Quaternion.Euler(0, 90, 0);
```

It's the same as entering those values into the Inspector. All of these are.

One thing to note is that it's an equals – it snaps our rotation to that exact angle, no matter how we were facing before. We can't add angles yet (it's not `+=`, it's more complicated than that).

This aims us straight up, with out feet facing forwards. Notice how we needed to use -90 to go up. +90 would be facing down:

```
transform.rotation = Quaternion.Euler(-90, 0, 0);
```

This aims us left, up 45 degrees, and on our back:

```
  transform.rotation = Quaternion.Euler(-45, -90, 180);
```

You need to sort of translate each value: y is clockwise with 0=north, so -90 y is facing West or Left. Negative x still goes up, so -45 x is 1/2-way tilted up. Then z has no affect on direction; 180 z merely rolls us on our back.

Also notice a cool thing about x: it isn't affected by compass facing. That's nice. If we see (-45, yy, zz), we know the cow's height angle is 45-up, no matter what yy is.

You're also allowed to use a Vector3, but it still needs to be an input to Quaternion.Euler. For example this aims South and randomly on our left or right side:

```
Vector3 spinv = Vector3.zero; // stands for a (000) rotation
spinv.y=180; // facing South
spinv.z=90; if(Random.value<0.5f) spinv.z*=-1; // left or right side
// spinv is either (0,180,90) or (0,180,-90)

transform.rotation = Quaternion.Euler(spinv);
```

It's just a shortcut. Quaternion.Euler needs an xyz, which can be 3 numbers, or one Vector3. It can also be a little confusing: a "spin" can now be a real quaternion, a Vector3 holding Eulers, or a Vector3 holding a direction (to be used in a LookRotation).

Using EulerAngles, we can have Update automatically change the angle. This makes a cow do forward summersaults (not backflips, since +x goes down, not up):

```
public float xSpin=0;

void Update() { // cow roller
  transform.rotation = Quaternion.Euler(xSpin, 0, 0);
  xSpin+=1;
}
```

Here's a trickier one. We want the cow to face right, rolling on its side (we're making a game about roasting a cow). Facing right is 90 degrees on y, and rolling forwards is +x. But this is wrong:

```
   Quaternion.Euler(xSpin, 90, 0);
```

The problem is that x spins in our personal forward. The cow is still doing summersaults. The rules about how x and z travel with y can fool us. We need to think about what rolls in our personal sideways, which is z:

```
// correct right-facing barbecue spinning:
public float zSpin=0;

void Update() { // sideways barbecue-rolling cow
  transform.rotation = Quaternion.Euler(0, 90, zSpin);
  zSpin+=1;
}
```

We can play with how x&y together aim us by having pairs of keys control them. A&D spins, W&S aims up/down:

```
Vector3 aimEulers=Vector3.zero; // aiming forward

void Update() {
  if(Input.GetKey("a")) aimEulers.y-=1;
  if(Input.GetKey("d")) aimEulers.y+=1;

  // notice how these go backwards, +1 is down
  if(Input.GetKey("s")) aimEulers.x+=1;
  if(Input.GetKey("w")) aimEulers.x-=1;

  transform.rotation = Quaternion.Euler(aimEulers);
}
```

This is a common trick. We can't just reach into the quaternion and adjust the xyz angles, since they don't exist. So we keep our own copy and send it to our rotation each time.

4.2.4 Rotate around an arbitrary axis

Sometimes we want to draw just any line through our origin, and spin ourself around that line.

An easy-to-see example, this spins us around a diagonal /-line:

```
public float degrees=0;

void Update() {
  Vector3 spinLine = new Vector3(1,0,1); // flat north-East /
  transform.rotation = Quaternion.AngleAxis(degrees, spinLine);
  degrees+=4;
}
```

If you put this on a cube, it will rotate perfectly corner-over-corner diagonally. On a cow, it will do the same thing but it will look a lot stranger (the head will tuck left, then it will be upside-down facing right, then back to normal).

After watching for a while, you should be able to "see" the diagonal line it spins around.

A semi-real example is a y-spin with a small wobble. We'll make a line *almost* straight up, leaning just a tad left, and spin around it:

```
public float degrees=0;

void Update() {
  Vector3 almostUp = new Vector3(-1,10,0); // almost up
  transform.rotation = Quaternion.AngleAxis(degrees, almostUp);
  degrees+=4;
}
```

Again, watching it should eventually help you see the almost-up line we're spinning around.

It might look nicer if the line we spin around is something we can visualize. This code snaps us to in-between the red and green cubes, then spins around that line:

```
public Transform redCube, greenCube;
float degrees=0;

void Update() {
  transform.position = (redCube.position + greenCube.position)/2;

  degrees+=3;
  Vector3 redToGreen = redCube.position - greenCube.position;
  transform.rotation = Quaternion.AngleAxis(degrees, redToGreen);
}
```

While running, moving the cubes around will change the line. In a sense, AngleAxis is very simple. It's a single spin around just one axis. If the axis is easy to visualize, AngleAxis looks like a simple spin.

4.2.5 FromToRotation

This command is an improved version of LookRotation. Instead of aiming the front, we can aim any part of us along the arrow. For example, if we're a cow, this aims our feet at the green cube:

```
Vector3 toGreen = greenCube.position - transform.position;
Quaternion feetToGreen =
   Quaternion.FromToRotation(Vector3.down, toGreen);
transform.rotation = feetToGreen;
```

The first input is the part of you to aim, as if you were standing and facing forward. `Vector3.down` always points your down-arrow at the target.

Suppose we want to snub the green cube by looking almost at it. We can do that by aiming an almost front arrow to it:

```
Vector3 almostFront=new Vector3(-1,0,20);
Quaternion aimToGreen =
   Quaternion.FromToRotation(almostFront, toGreen);
```

Since the aiming arrow was slanted a little bit left, and goes straight to green, our head will be facing a little bit to the right.

We rarely need this. Suppose we always want to aim our up arrow somewhere. We'd use the parent trick to spin Up to Forward, then aim our Forward like a normal person.

4.2.6 LookAt

Making us look at a point is so common that Unity has a shortcut command. You give it the point, and it automatically computes the offset, then uses that to aim us.

`transform.LookAt(greenCube.position);` makes us look at the green cube. This command is so useful and common that it's easy to forget that's it's only a shortcut for LookRotation:

```
// this is what LookAt does:
void LookAt(Vector3 pos) { // pos is a point, not an offset
  Vector3 dir = pos - transform.position; // get the offset
  Quaternion r = Quaternion.LookRotation(r);
  transform.rotation = r;
}
```

The drawback is that it automatically aims us. For fun, here's code to fake LookRotation using LookAt. We do a LookAt, read our current rotation as the answer, then reset our rotation to how it was. It's hilariously Rube-Goldburgy, but it works:

```
Quaternion savedOriginalSpin = transform.rotation;

transform.LookAt(greenCube.position);
Quaternion greenQ = transform.rotation; // <- the answer

transform.rotation = savedOriginalSpin; // restore my rotation
```

LookAt works with any object or any point:

```
// the red cube rotates to face me:
redCube.LookAt(transform.position);

// green cube faces red cube:
greenCube.LookAt(redCube.position);

// look at spot in front of red cube:
transform.LookAt(redCube + redCube.forward*6);

// look at the point (10,9,32):
transform.LookAt(new Vector3(10,9,32));
```

The last one is another example of how to tell a point from an offset. (10,9,32) could be an arrow – it would be forward, tilted right and a tad up. But LookAt expects a position, so (10,9,32) counts as a position here.

4.2.7 Directions vs. rotations, LookAt z-spin

A direction arrow is almost a rotation, but not quite. A rotation is made of the direction plus the "free" z-spin. This is the thing where z always goes last and merely rolls you around the direction arrow. A cow aimed at a farmer could be head up, feet up, lying in its side, or so on. Or, again, the goon playing with the cool sideways gun-aiming technique.

A way to see this is that rotations are on 3D models. They have textures, and feet sticking out. We can see them roll. But directions are on arrows. It makes no sense to roll an arrow around itself.

LookRotation turns a direction arrow into a rotation, but since directions don't have the extra z-spin, it cheats. It makes-up a z-spin of zero. In other words, `LookRotation(v)` doesn't give you *the* rotation to face that way. It gives you one possible rotation, out of many.

Later on we'll be able to turn a rotation into a direction. Going from a direction to a rotation then back is safe. It adds a z-rotation of 0, then takes it off. You get back the same arrow you started with. Going from a rotation to a direction then back destroys your z-roll. You'll still be pointed the same way, but your y will always be aimed up.

`LookAt` and `LookDirection` have an option to set your z-spin, but in an odd way. You tell it which way you'd like your +y to point. It's usually thought of as which way your head points. The z-spin will be set as best it can.

These commands will aim you at green, lying on your side with your head pointed right:

```
transform.rotation = LookRotation(toGreen, Vector3.right);
transform.LookAt(greenCube.position, Vector3.right);
```

This shows why we like the "head direction" method. Usually we don't know the z-degrees we want, but know how the top should point. In fact, we couldn't even do this by giving a z-roll. To keep its head facing right, the cow needs to switch sides when aiming forwards vs. backwards.

The thing to remember is this is only a z-roll. The first input is still the real way we aim. The `Vector3.right` means to spin on z to the most right-aiming spin we can get – pick the best out of the 0-360 z-spin.

4.3 `eulerAngles` and round trips

This is a "don't do this" section. It's explaining why, if you want to change your aim with moving xyz degrees, you need to keep your own copy.

The main issue is that when you give the system xyz Euler angles, you won't get the same ones back.

You're allowed to ask for the Euler angles with `eulerAngles`. It gives you a Vector3 with the 3 spins:

```
  transform.rotation = Quaterion.Euler(0,30,120);
  print( transform.rotation.eulerAngles ); // 0,30,120

  transform.rotation = Quaterion.Euler(-90, 0, -20);
  print( transform.rotation.eulerAngles ); // 270, 0, 340
```

The first one gave us the same numbers back, but the second one "fixed" them (for real it recomputed them differently, which feels like it fixed them). Unity prefers 0-360 for y and z. For x it uses only 0-90 and 270-360. The second range is for negative spins; -20 is stored as 340. It cuts out 90-270 since those are over-the-top x-spins.

If you need to change and move xyz Eulers, the best way is keeping your own copies, like the aiming with ASWD keys example. This spins y from 90 down to -90 over and over again:

```
Vector3 vSpin=Vector3.zero; // no rotation -- facing forward

void Update() {
  vSpin.y-=1; // from right to left, then resetting
  if(vSpin.y<-90) vSpin.y=90;

  transform.rotation=Quaternion.Euler(vSpin);
}
```

The system converts negative y's into 270-360, but we don't care since we never look at `transform.eulerAngles`. `vSpin` is like our master copy, and its y is fully controlled by us.

4.4 Details and math

These sections are all comments, background and other things that are nice to see, but aren't really important.

4.4.1 Real Quaternion values

If you look inside a quaternion, there's an x, y, z and w. I already wrote that they aren't degrees. If you know trig you might think they're radians – nope. They're totally different things called x, y and z (and w).

If you have an irresistible urge to look at the actual numbers in a quaternion, it's a simple 4-float struct. This code would show them:

```
public float x,y,z,w; // will be copied from the quaternion

void Update() {
  Quaternion q = transform.rotation;
  x=q.x; y=q.y; z=q.z; w=q.w;
}
```

For 50 on z, (0,0,50) this gives (0.4, 0, 0, 0.9). That means turning on z uses the x and w slots, with numbers that make no sense.

Some more:

```
rotation     quaternion
0,0,0         0, 0, 0, 1
0,90,0        0, 0.71, 0, 0.71
90,0,0        0.71, 0, 0, 0.71
180,0,0       1, 0, 0, 0
90,90,0       0.5, 0.5, -0.5, 0.5
```

There doesn't seem to be much of a pattern. Obviously there is. You can find the formulas. But there's nothing useful you can do with those numbers that isn't already in LookRotation and so on.

If they're so unhelpful, why aren't they `private`? That's because quaternions are real math, and actual mathematicians using Unity might want to directly play with x,y,z&w to do something obscure.

4.4.2 Multiple ways to write an angle

A crazy thing about xyz degrees is there are 2 legitimate ways to write every angle. I'm not talking about +/-360 tricks. There's another way involving an up-and-over on x.

Whenever x goes past 90, which is aiming backwards what your y says, you can make the same angle by flipping y by 180 and recomputing x. Here we can rewrite a 135 degree x-rotation as a 45-degree going the other way:

```
A                          A
 \  x=135              x=45 \
  \                          \
   o -> y=0          y=180 <- o

           side view
```

That isn't quite right, since the first way put us on our back, whereas the second keep us feet-down. Flipping z by 180 is the rest of the trick.

The two identical rotations in the picture are (-135,0,0) and (-45,180,180). The first is shorter to read, but the second makes it more obvious that we're actually facing South and are upside down.

Every rotation can be rewritten like that (add 180 to x and z, make x=180-x). Every rotation has a version with x from -90 to 90, and another with it past 90. Fun fact: (0,0,0) is the same as (180,180,180).

In case you were wondering, that's very strange, and we don't like it. But that's the way it is.

The net effect is that moving and hand-checking Euler angles is even more of a giant mess. If all you do is aim mostly forward and keep x and y both 90 or less, things are fine. But with free-movement, anything like `if(q.eulerAngles.y>180)` is doomed to failure.

4.4.3 How the Inspector shows rotations

The Inspector shows a different version of the rotation values. It's not the same as the ones `eulerAngles` computes in the code. Unity saves the starting Inspector values, and keeps what it displays close, using +/-360. It thinks you will like that.

Suppose you start a rotation at (0,0,0) – those are the values entered into the Inspector. While the program runs, the Inspector adjusts everything to between -180 and 180. If a value goes to 181, which is fine inside the program, the Inspector shows it as -179. If you started the x rotation at 90, the range for x would be -90 and 270. When your program sees 271, the Inspector shows -89.

It seems like a huge problem, but hand-checking Euler angles is already such a mess that this doesn't make it much worse. And if you keep your own xyz rotation variable, it will be fine.

4.4.4 Moving with euler xyz problems

We use quaternions because Euler angles have problems. But what are these problems? As we've seen, Euler angles are pretty good for setting a facing. So far that's all we've done, so they seem fine. It turns out that Euler's are terrible at moving between angles.

Basically y&x based rotation movement always works like an army tank aiming its gun. When targets are near the ground, we can track them pretty well. We rotate the turret with y, x is and up/down for hills and valleys. Anything higher up will give us more trouble. It takes less motion for the same degrees on y, making it easer to outrun our turning.

Suppose a flying saucer zooms up directly overhead and we angle x up to 90, tracking it. It can go a little sideways and be safe. Even though the tip of our cannon only has to move a tiny bit sideways, we'll need to spin our turret a full 90 degrees to be able to tilt left.

If you're making that game, you should use Euler Angles for rotation. Use the trick where you keep them in your own Vector3. It will give the weird gear-based gun rotation you want.

The problem happens because most things aren't army tank guns, and look terrible when the work that way. Usually we're looking at the red cow and want to smoothly turn to face the blue one. We don't want the camera to have a little extra curve, or sometimes go really slow with an extra tight funny little spin.

Unlike a 2-gear Euler system, quaternions don't have any good or bad spots. They don't have any problem areas like when a cannon aims mostly up or down. Going from any spin to any other, quaternions always gives a nice straight angle, at the same speed

If you've seen the term Gimbal Lock (the Flying Saucer example has it), quaternions don't have that.

4.4.5 Quaternion setting commands

Our commands all *compute* a quaternion, like:
`q1 = Quaternion.LookRotation(v);`. They have alternate versions where a quaternion sets itself. I've never used them, and they don't do anything new, but seeing the alternate form is a fun review.

Here's the old and new version of LookRotation:

```
Quaternion q;
```

```
q=Quaternion.LookRotation(toGreen); // old way
q.SetLookRotation(toGreen); // new way
```

The second one computes the rotation and puts it directly into `q`. It seems more efficient, but you can't use it in a formula. Not having to write `q=` isn't that much of an advantage.

You could write `transform.rotation.SetLookRotation(toGreen);` since your rotation is a quaternion. But no one ever does. `transform.LookAt(greenCube);` is a better shortcut.

Recall AngleAxis spins you around any line. Spinning around Vector3.forward rolls you sideways. Both of these tip you 5 degrees left:

```
transform.rotation = Quaternion.AngleAxis(5, Vector3.foward); // old
transform.rotation.ToAngleAxis(5, Vector3.forward); // new
```

If our cow has a unicorn horn sticking up at 45 degrees, these will aim it at the green cube:

```
Vector3 hornAngle=new Vector3(0,1,1); // 45 up and forward
Quaternion q;
q = Quaternion.FromToRotation(hornAngle, toGreen); // old
q.SetFromToRotation(hornAngle, toGreen);

transform.rotation = q;
```

The strangest one is the shortcut for setting xyz Euler Angles. It doesn't fit the pattern. It looks like an assignment statement, but it's a disguised function call:

```
q=Quaternion.Euler(0, 90, 10); // normal
q.eulerAngles = new Vector3(0,90,10); // alternate
```

Again, quaternions don't actually save the Euler angles. The second command is just running the first one.

There are also 3 shortcuts for FromToRotation – aiming a different part of you somewhere:

```
transform.forward = toRedCube; // same as LookRotation
transform.up = toRedCube; // aim your back that way
transform.right = toRedCube; // aim our right side

// same as:
transform.rotation =
  Quaternion.FromToRotation(Vector3.right, toRedCube);
```

These are more tricks with disguised functions. They actually call FromToRotation to do the work.

We can use the backwards arrow trick to assign to the other 3:
`transform.up = -toRedCube;` points your feet at the red cube, by pointing your head in the exact opposite direction.

Chapter 5

Combining rotations

Instead of thinking of a rotation as an absolute facing, you can think of it as a change – an offset. In other words, `Quaternion.Euler(0,90,0)` could mean facing east, or it could mean a quarter circle spin from how-ever you're facing now.

We can apply a rotation to an arrow, which spins the arrow. Or we can apply a rotation to another rotation, which adds them together, sort of.

Rotations are applied using a re-purposed star: `v1=q*v1;` rotates the arrow `v1` by `q`. Re-using the star is like how we re-use the `+` to mean push-together in `"cat"+"fish"`. We're not actually multiplying the rotation numbers together. We're running combine rotation equations. But mathematicians actually call it quaternion multiplication.

This is some of the hardest stuff, but if you have the basic idea, you can usually trial and error to get an equation working.

5.1 Rotating an arrow

To apply a rotation to an arrow, use `rotation*arrow`. For example, this spins a right-pointing arrow by 90 degrees:

```
Vector3 vr = new Vector3(3,0,0); // long right arrow
// standard 90 degree spin:
Quaternion y90 = Quaternion.Euler(0,90,0);

vr = y90*vr; // vr spins from right to backwards
```

We're not thinking of `y90` as an eastward facing anymore. Now it's a free-floating clockwise 90-degree y-spin. It's like a card in a game that says "turn right".

Arrow-rotation works with any kind of spin and any arrow. This finds an arrow to the red cube, then spins it 20 degrees on y:

```
Vector3 toRed = redCube.position - transform.position;
Quaternion y20 = Quaternion.Euler(0,20,0);

Vector3 almostToRed = y20*toRed;
// place a ball there to prove we did it:
theBall.position = transform.position + almostToRed;
```

Imagine the start of the arrow glued to us. The 20 degree spin moves it like a clock hand. The ball will be the same distance from us as the red cube is. If the cube was at 4 o'clock, the ball will be at 5 o'clock. It will be at the same height as the red cube.

Way back in the game board example we started with only the two bottom corners. To get the others we took the arrow along the bottom, pretended it was glued there, and spun it backwards 90 degrees to aim at the top-left corner. We had to use a cheap trick then. Now we can do a real 90 degree spin:

```
public Vector3 lowerLeft, lowerRight; // user enters these two

  Vector3 acrossArrow = lowerRight-lowerLeft;

  // spin the acrossArrow to get an upSide arrow:
  Quaternion spin90back = Quaternion.Euler(0,-90,0);
  Vector3 upSideArrow = spin90back*acrossArrow;
```

Now `lowerLeft + upSideArrow` gives us the upper-left corner.

A neat variation of that is making a pentagon. A little math shows the outside bend at each corner is 72 degrees. I'll start at the lower-left and go counter-clockwise:

```
    4
 5     3
  1   2
```

I'll assume we have 5 little cubes to drop at each corner. The code will use the same arrow, twisting it to walk along each new edge:

```
Vector3 corner=Vector3.zero; // corner #1
Vector3 arrow = Vector3.right*2; // the arrow from 1 -> 2
Quaternion cornerSpin = Quaternion.Euler(0,-72,0);
c1.position = corner;
corner+=arrow; // 1->2
c2.position = corner;
```

```
// now spin, move and place the rest of the corners:
  arrow = cornerSpin*arrow;
  corner+=arrow; // 2->3
c3.position=corner;
  arrow = cornerSpin*arrow;
  corner+=arrow; // 3->4
c4.position=corner;
  arrow = cornerSpin*arrow;
  corner+=arrow; // 4->5
c5.position=corner;
```

It's a little boring, which is the point. The rotating an arrow trick lets us do a boring turtle-graphics-style walk, using 1 arrow and 1 rotation. Each time we use the rotation on the arrow, it spins another 72 degrees counter-clockwise.

A fun trick is gradually rotating an arrow. It's the same as rotating ourself, except with an extra arrow coming out of us. This spins an arrow from forward, clockwise to backwards, repeating. To see if it worked we'll put a red cube at the tip:

```
public Transform redCube;
float degrees=0;
Vector3 baseArrow = Vector3.forward*2;

void Update() {
  // spin goes from 0 to 180, over and over:
  Quaternion spin = Quaternion.Euler(0, degrees, 0); // y-spin
  degrees+=3; if(degrees>180) degrees=0;

  Vector3 arrow = spin*baseArrow;

  redCube.position = transform.position + arrow;
}
```

`spin*baseArrow` is the fun part. It's an arrow forward, clock-spun 3 degrees, 6 degrees and so on. The tip traces out the right half of a circle, forward to back.

`spin` is really a free-floating rotation "slowly turn 180 degrees right". If we change `baseArrow` to `Vector3.right;` it will spin around the back half. Changing it to `new Vector3(3,0,-1)` would spin over whichever weird half-circle started there.

Some notes on the rules for using these:

- You can't flip the order. `spin*v` rotates `v`, but `v*spin` is an error.

- The star(`*`) isn't a multiply. For example, in rotation (10,45,90) times point (3,4,8) we're definitely not taking 10 times 3, 45 times 4 and 90 times 8.

  We're really running a function with those two inputs, doing lots of ugly angle math.

- Regular precedence rules apply. `v1+spin*v2` spins `v2` first, then adds `v1`. Using `spin*(v1+v2)` adds the arrows first, then spins the result.

- There isn't a `q1+v1`. We only needed 1 symbol and we picked `*`.

5.1.1 Converting a rotation into its arrow

In the last chapter `LookRotation` converted an arrow into a rotation. Sometimes it's nice to convert a rotation into an arrow: the (000) rotation is the forwards arrow, and so on.

The trick is to start with the forward arrow. Using it is like adding the rotation to 0. `Quaternion.Euler(0,45,0)` is a NorthEast rotation. Quaternion.Euler(0,45,0)*Vector3.forward is an arrow aimed NorthEast.
For any rotation, `q*Vector3.forward` is an arrow pointed the way `q` would aim.

Suppose we need an arrow aimed forwards and right 30 degrees. Normally we'd just write it as (x,0,z), but I don't know the ratio for 30 degrees. Instead we'll create that rotation and use it to make the arrow:

```
Quaternion qy30 = Quaternion.Euler(0,30,0);
Vector3 v30 = qy30*Vector3.forward;
// v30 is a 30-degree tilted forward arrow
```

Hopefully this is easy to read. We don't need to imagine the arrow then add that spin. Hopefully we'll see `Vector3.forward` and realize we can think of the spin's facing, and that's the arrow we'll get.

Unity actually uses this trick to turn your rotation into your forwards arrow. `transform.forward` is really `transform.rotation*Vector3.forward`.

5.1.2 More arrow rotation

For an exercise, suppose we want the red cube to start on our right side, then spin left, up and over making a 1/2-circle above us; then repeat.
Plan #1 is this: create a right arrow as the starting spot. As the picture shows, rotating that arrow around z will bring it up and to the left side:

```
    Top View
        | +z
  end   |
   o----+----O start
        |  +x
        |
```

The code for spinning a rightward arrow around z:

```
  Vector3 baseArrow = Vector3.right*3;
  Quaternion spin = Quaternion.Euler(0, 0, deg0to180);
  Vector3 arrow = spin*baseArrow;

  redCube.position = transform.position+arrow;
```

That doesn't feel too bad. `baseArrow` is exactly where red starts, and spinning it around z, like a propeller seems obvious enough.

Plan #2 is to make it using pure Euler rotations. Make a rotation that faces right, then leans back up and over to face left. Use `Vector3.forward` to turn it into an arrow:

```
  Quaternion spin = Quaternion.Euler(deg0to180, 90, 0);
  Vector3 arrow = spin*Vector3.forward;
```

If you can look at `Quaternion.Euler(degs0to180, 90, 0)` and quickly see it as a right arrow, tilting back on x, then this way is better.

Suppose we want to 1/2-circle the red cube from our *personal* right to left. We'll start with the arrow `transform.right*2`. The end is attached to us, spinning around the long line through our body – around our personal z-axis.

This is the cool part: that line is `transform.forward`, and we can make a rotation around it using `AngleAxis`. Here's the code:

```
// our right arrow, spinning around our +z:
Quaternion spin = Quaternion.AngleAxis(degs0to180, transform.forward);
Vector3 arrow = spin*(transform.right*2);

redCube.position = transform.position + arrow; // ball circles R to L
```

That was a slippery one. The trick is to think of `spin` as a free-floating rotation. `AngleAxis(0,transform.forward)` means no spin, which is no change. Changing 0 to 10 means to take whatever arrow and spin it 10 degrees diagonally around our forward arrow, and so on.

As I wrote in the intro, sometimes these are trial and error.

5.1.3 Cones, weirdness

Spinning an arrow around itself does nothing. For example spinning Vector3.up around the y-axis is always Vector3.up again. This code does that – the red cube at the tip won't move:

```
// spinning the up arrow around y does nothing:
degrees+=2;
Vector3 arrow = Quaternion.Euler(0, degrees, 0) * Vector3.up;
redCube.position = transform.position + arrow;
```

It works, it spins the arrow, But the tip doesn't go anywhere. It doesn't even have a secret spin. The input arrow is (0,1,0) and the output arrow is (0,1,0).

This next code also does nothing. It spins our personal forward arrow around itself. The red cube sits there, 3.5 units in front of us:

```
// forward arrow:
Vector3 baseArrow = transform.forward*3.5f;
// a spin around our forward arrow:
Quaternion spin = Quaternion.AngleAxis(degs0to180, transform.forward);
// this is always the same arrow we started with:
Vector3 arrow = spin*baseArrow;

redCube.position = transform.position + arrow; // never moves
```

Hopefully both of these are totally obvious. If you swing a stick, the tip moves. If you twirl it in place, the tip doesn't go anywhere.

There's nothing wrong with spins that don't move you. It's not an error. If you have lots of changing arrows and ways they spin, it makes complete sense that they sometimes line up for a do-nothing spin.

Spinning funny-angled arrows is like twirling an umbrella and tracking the end of a spoke. If it's all the way open, the spoke-ends make a big circle. As you close it, they make smaller and smaller circles. The spokes haven't gotten any shorter. We're still spinning the entire 2 feet worth of spoke, but the angle gives us tiny circles.

Here's code for an almost-closed umbrella spin. It takes a long almost-up arrow and spins it around the y-axis:

```
Vector3 baseArrow = new Vector3(1,8,0); // 8 up, 1 right
Quaternion spin=Quaternion.Euler(0,degs0to360,0); // spin around y
Vector3 arrow = spin*baseArrow;

redCube.position = transform.position + arrow; // radius 1 circle
```

The red cube is always at y=8, tracing out a small radius 1 circle. It acts like an almost-folded umbrella spoke.

A long arrow at a small angle spinning in a cone is fine. But all the matters is how the tip moves. The area swept out by the long part doesn't really matter.

Math-wise, arrows can be broken into the part along the spin arrow, and the part going straight away from it. That second part is what moves. For example (1,8,0) around y is like climbing up 8 and staying there; and spinning just (1,0,0) to make a small perfect circle.

To sum up: the best spins are when the spin axis and the arrow are at 90 degrees. Other angles are fine, but you may have to work harder to visualize how the tip moves.

5.1.4 Summary

- Use the forward arrow to turn a rotation into its arrow. If `q` is a rotation, `q*Vector3.forward` is the rotation's direction arrow.
- For everything else, don't think of which way a rotation points. Think of how it changes you. Instead of "face right" it would mean "turn right".
- The rule that z won't change your direction is only for rotating 3D models facing forward. For rotations spinning arrows, rotating them on z is perfectly fine and useful.
- Arrows that stick out at 90 degrees from the spin axis are easiest to visualize. But that doesn't have to happen. Non-perpendicular arrows will spin in a cone, which is legal and sometimes useful.
- Any type of rotation works equally well. Quaternion.Euler is good. But `AngleAxis` is often very useful. Especially AngleAxis around someone's personal arrow to spin in their personal space.
- We can now go back and forth from direction arrows to rotations: `q=LookRotation(v)` and `v=q*Vector3.forward` to go back.

5.1.5 Ball cone-shooting example

For a 3D game about fire-hoses, we'd like to shoot little blue balls forward in a small random cone. If we put a wall ahead of us, each ball could hit anywhere within a small circle.

The plan is to start with the forward arrow, which stands for perfect aim. On our aiming circle we'll pick a 0-360 degree direction to miss in. Then we'll pick 0-10 degrees for how far we're off. 0 is the center – we don't miss at all. 10 means we hit all the way at the edge.

That plan should work, but we need another plan to do it:

Version #1 that won't work: make a Vector3 with (x,y,10) where x and y are random -1 to +1. That will give a random mostly forward arrow that can tilt any direction. But the target area will be square. People will notice when we shoot enough, so it's no good.

Version #2 that won't work: Create an Euler rotation that can aim that way. That won't work because we can't make one. We can roll 0-360 on z, but tilting 0-10 on y will only ever go right/left. Or we can compass spin 0-10 on y, but then z-spin will only roll us in place.

That stupid yxz rule is getting us. `Quaternion.Euler(x,y,z)` can't spin y then z like we need.

Version #3: Tilt the forward arrow in 2 steps. First angle it right 0-10 degrees. Second, spin that arrow 0-360 around z. The arrow will trace out a small cone, which is exactly what we want.

The code, in a few parts:

```
Vector3 dirArrow = Vector3.forward;
// small rightward cock:
float missAngle=Random.Range(0.0f, 10.0f);
dirArrow = Quaternion.Euler(0, missAngle, 0) * dirArrow;
```

In our mind `dirAngle` is the actual direction the ball will go. Currently it's cocked 0-10 degrees right. Next we spin it around z. The tip will trace out a small circle:

```
Quaternion zRand360 = Quaternion.Euler(0, 0, Random.Range(0,360)) ;
dirArrow = zRand360 * dirArrow;
```

`dirArrow` is now aimed anywhere within a 10-degree forwards cone. To prove it works, we'll shoot the red cube using it:

```
// ball is 3 units along the aim arrow:
ball.position = transform.position + dirArrow*3;
// speed is in direction of aim arrow:
ball.GetComponent<Rigidbody>().velocity=dirArrow*10;
```

Finally, that always shoots forward. Suppose we want to shoot based on the way we're aiming. That seems like it might be hard, but it's not. We can rotate the arrow by our rotation:

```
Vector dirArrow = transform.rotation*dirArrow;
// aims forwards from us, in a 10 degree cone
```

Later we'll see a better way to think of this trick. But it's basically taking an almost-forward arrow, spinning it by us, to get an almost-our-forward arrow.

5.2 Combining rotations

We can also apply one rotation to another. It uses the star symbol in the same way. You write it like multiplication, but it's really running rotation math. `q1*q2` combines `q1` and `q2` into one big rotation.
`transform.rotation = q1*q2;` is completely legal.

But we run into the local/global axis problem we had with Euler angles. Suppose we start rotated `q1` and add a 30 degree z spin. Is that on our current z-axis, or the global z axis? The result will be different.

The good part is, we can pick global or local axis. The bad part is, we do it in a semi-confusing way.

5.2.1 Applying local rotations

When you multiply rotations, the first one is like the start, and the second one spins it more. It uses the local axis of the first. That's the entire rule: "second spin starts with the local axes of the first". It works just great, but it feels really odd at first.

To see it work, let's make 2 simple rotations: a lookAt, and a 360 degree spin around y:

```
// simple look rotation to red cube:
Vector3 toRed=redCube.position-transform.position;
Quaternion redLook = Quaternion.LookRotation(toRed);

// standard y-spin
deg+=2; if(degs>360) degs-=360;
Quaternion ySpin = Quaternion.Euler(0,degs,0);
```

Those two rotations are nothing special. `transform.rotation=redLook;` aims us at red, and `transform.rotation=ySpin;` gives us a clockwise flat compass spin.

But combining them gives something we couldn't do before. It makes us spin in a tilted circle, aimed at red (to see it, red needs to be away from us, and above or below):

```
// titled y-spin aligned with red:
transform.rotation = redLook*ySpin;
```

Think of it as aiming at red, then purposely adding `ySpin` after that. It's only natural it would spin on the tilted y.

We can use 2 combined rotations to solve the fire-hose cone problem. Spin 0-360 on z, then tilt 0-10 degrees to our current right:

```
Quaternion z360 = Quaternion.Euler(0,0,Random.Range(0,360));
float yAmt=Random.Range(0.0f, 10.0f);
Quaternion y0to10=Quaternion.Euler(0,yAmt,0); // small y tilt

Quaternion qAim=z360*y0to10; // <-final rotation
Vector3 aim = qAim*Vector3.forward; // convert it into an arrow
```

Visualize `z360*y0to10` as a cow spinning twice. First `z360` rolls it sideways. It's facing forward, but its right side is spun around to just anywhere. Next it spins 0 to 10 to its *personal* ride side, which for real could be up, left or any direction.

Breaking it into 2 rotation lets us force the order to be z then local y. We had no way to do that before.

In `Quaternion.Euler(-45,90,0)` we know the rule is 90 y first, then -45 on the local x. The word local means the same thing there as it does here. We can break (-45,90,0) into (0,90,0) * (-45,0,0) and it means real y90, local x45:

```
Quaternion ySpin90 = Quaternion.Euler(0,90,0);
Quaternion xSpin45 = Quaternion.Euler(-45,0,0); // tilting up

Quaternion y90thenx45 = ySpin90*xSpin45; // same as (-45,90,0)

transform.rotation = y90thenx45; // testing
```

`ySpin90*xSpin45` is just a long way to write `Euler(-45,90.0)`. We wouldn't write it out for real, since writing `Euler(-45,90,0)` is easier in every way.

For a longer example: as we all know, the Mark-II plasma cannon is mounted on a circular base. To aim, the whole base tips straight back. The scary-looking barrel swings side-to-side like the hand on a tilted clock.

In math terms, it uses an xy order – global x, local y. We can't make that with `Euler`. We'll have to use the combine rotations trick:

```
public float xTilt, ySpin; // degrees. Slide-change in Inspector

void Update() {
  Quaternion qTrack = Quaternion.Euler(0,ySpin,0);
  Quaternion qLift = Quaternion.Euler(xTilt,0,0);

  plasmBarrel.rotation = qLift*qTrack; // <-lift, which is x, goes first
}
```

If you try this, you can totally see how y is local. At first it's a normal side-to-side. But if you tilt back on x, `ySpin` is in a titled arc. If you've ever seen a real Mark-II plasma cannon, you'll recognize the distinctive rotation.

This next one is mostly an excuse to use three in a row. We want the cow to lie on its left side, with its back aimed at the red cube. Here's the plan: aim its face at the red cube; spin it 90 degrees on y – now the left side is aimed there; finally lie down left– now its back is aimed there, with the cow lying on its left side.

In code we make all 3 rotations and combine them in order:

```
Quaternion faceAimRed;
Vector3 toRed = redCube.position - transform.position;
faceAimRot = Quaternion.LookRotation(toRed);
Quaternion spinRight=Quaternion.Euler(0,90,0);
Quaternion fallLeft=Quaternion.Euler(0,0,90);

transform.rotation = faceAimRed*spinRight*fallLeft;
```

I think having three helps see the local rule. First we aim at red. Then spin right based on that. It's important we spin on local y so the left side is aiming exactly at red. Then we flop down left, based on how we're standing now.

Since y beats z, we could have combined them as `Euler(0,90,90)`, but that would have been harder to read. Facing right, then falling feels like it should be 2 steps.

Here's a simpler semi-practical one. We'd like an object to z-spin on it's starting rotation. Without changing it's aim, it will roll in place. We'll save the original rotation, create an increasing z-spin, and add it as local:

```
Quaternion baseFace;
int zDegs=0;

void Start() { baseFace=transform.rotation; }

void Update() {
  zDegs+=2; if(zDegs>360) zDegs-=360;
  transform.rotation = baseFace*Quaternion.Euler(0,0,zDegs);
}
```

It feels funny since we're recomputing our entire rotation from scratch every time. We take our beginning rotation and apply a 2-degree z-spin. Next frame we do the same thing, but with a 4-degree z-spin.

This trick works to take any arrow and give it a fun roll-in-place.

General local rotation combining notes:

- A rotation by itself isn't local or global. It depends on how you use it. `transform.rotation=q;` is using `q` as global. But `transform.rotation = spin1*q;` uses `q` on `spin1`'s local axes. But then `q*spin1` uses `q` on gobal axis again.

- The "y is always flat" rule only applies for values inside `Euler(x,y,z)`. For example, `q*Quaternion.Euler(20,50,0)` starts in `q`'s local space, then does 50 y first, and 20 x last.

  We often use the combine rotation trick purposely so we can y-spin on some local axis.

- There's no `q1+q2`. There's only 1 rule for combining them, and we picked `*` for it.

- The trick to making the second (and third ...) rotation is to make it as if you were global. `Euler(0,0,90)` lies on your left side. Using it after some other rotation lies on your local left side.

  Part of why we like the combine rotations is how simple that makes things.

5.2.2 Applying global rotations

We can also take a starting rotation and apply to it another as a global. To do that, the starting rotation goes second, and the one to apply as global goes first: `globalSpin*startingFace`.

Let's start with making a global z-roll. z is normally a local spin, so this should look interesting. We'll aim at the red cube and spin on z globally, by putting it in front:

```
public Transform redCube; // look at me
int zDegs=0; // constantly increasing 0-360

void Update() {
  zDegs+=2; if(zDegs>360) zDegs-=360;
  Quaternion zRoll=Quaternion.Euler(0,0,zDegs);

  Vector3 toRed=redCube.position - transform.position;
  Quaternion qToRed=Quaternion.LookRotation(toRed);

  transform.rotation = zDegs*qToRed; // <-spin qToRed on global z
}
```

Since `zDegs*` was first, this will not be a nice local roll on z. The arrow to red will spin around the real z-axis. It will aim it away – far away – from the red cube, probably spinning in a weird little cone. Each full circle will bring it back to looking at red. It's kind of a garbage formula.

You may be thinking "hey wait – you just told me `q1*q2` starts with `q1` and adds `q2` as local. Now you're telling me it's `q2` adding `q1` as global. Which is it?"

Amazingly, it's both. We'll just save that bizarre fact for later.

Going back to horrible arrow spins, suppose I have a 3D game where a cannon aims at a target, but also spins in a circle. You have to tap when it's pointing at the target. It should look like the up/down gear is fine, but the compass gear is running amok. The changes to do that:

```
Quaternion y360=Quaternion.Euler(0, deg,0); // y-spin

Quaternion cannonAim=y360*qToRed; // apply y to aim, as global
```

This gives us the same junky rotation where the arrow to red spins around crezily, but now it makes sense. The blown gasket is merry-go-rounding the aim arrow.

This last one is tricky. We've got a flat spinner that tells us which way the wind blows – North, East and so on. On a signal, everything will tip 90 degrees in the wind direction. Everything tips the same way, since this is just general wind – it's not coming from the spinner.

After some thinking, the wind-blow rotation should be 90 degrees around the spinner's personal x-axis. If the spinner was aimed forward, that flops everything forward – it seems to check out. The math for "90 on spinner's x" is: `Quaternion.AngleAxis(90, spinner.right);`.

Surprisingly, we want that as a global rotation. That's because we're using that axis exactly as it is. Local means we want each object to adjust the wind's x-axis based on their own rotation. That's clearly wrong. With a global wind-flop, the final code is:

```
Quaternion windSpin= Quaternion.AngleAxis(90, spinner.right);
// NOTE: +90 on x is forward&down

foreach(Transform blowMeOver in StuffToBlow)
  // push each item over by 90 in wind direction:
  blowMeOver.rotation = windSpin*blowMeOver.rotation;
```

5.3 Misc rotation combinations

We're back to the bizarre fact that for rotations, `A*B` is either A with B added locally, or B with A added globally. Both things are true. It's one of those crazy math facts.

When you're creating one, it's no problem at all. Suppose you have a plan to start with A, add B on A's locals, then twist it all using global C. That's `C*A*B` (A with B after and C before).

When you're trying to read a compound rotation, it's still not so bad. Think of having two ways to read it, and only one needs to make sense.

Take `lookAtRed*zRoll`. That can be looking at red first, then doing a z-roll on the local axis. Local z-rolls don't change how we aim, so that feels pretty good. Or it's z-roll first: we start aimed North, rolling in place. Then `lookAtRed`, as a global, twists us to look at red, taking the z-roll with it.

Consider the left-lying back-facing cow: `toRed*turnRight*lieOnLeft`. In this we should start with `turnRight*lieOnLeft`. Those take a 000 rotation cow and set it up – it's looking East, lying on its side. That's easy to visualize. Then we apply `toRed` as a global, spinning the whole thing in-place to face the red cube.

That's a useful pattern. Whenever a look-at comes first, think of it as a global. It applies last. Whatever you made with the rest, it gets aimed at red.

From before we know `z360*y0to10` makes a rotation aimed in a small forward cone. One way to read it: 360 spin on z, which does nothing yet. Then the random 0-10 local y-tilt, which could go in any direction of the cone. Or read it as: random 0-10 degree tilt right, followed by a global z-spin which traces out the cone.

The last step of the old cone example was making it face the way we do. That's another example of the "look-at goes first" pattern. `transform.forward*z360*y0to10` is a global `transform.forward` aiming the forward cone.

For fun, we can break `Quaternion.Euler(45,20,180)` into all 3 parts. The official order is always yxz, so we can write it as `y20*x45*z180`. We're allowed to think of that as x first, local z, then global y. Or any order as long as it follows the local/global based on how it was written.

Summary, notes:

- Whenever you create a combined rotation, you have to think about global vs. local. If you have a 90 degree spin on y, is it on the real y, `ySpin*q` or the local y `q*ySpin`?
- The order of a multiply matters! This one almost slipped by. We "know" the order of multiply never matters, but that's only the rule for normal algebra. For rotations, the order matters.
- You can sometimes fix things by flipping the order. Suppose `A*B*C` works wrong, but you're sure A, B, and C are the right rotations. Try moving A to the back, or flipping B and C around.
- You can create a different-order-Euler by doing it one axis at a time, in the order you want. If, for whatever reason, you need a rotation on z, then x, then y, you can force it with `qZ*qX*qY`.
- When you read a rotation, think of it both ways. Read `A*B` as A first, B local; or B first, A global. Usually one way makes more sense.

- z-rolls as the last thing are fun ways to play around without changing the aim direction. LookAt's usually go first, aiming at whatever the rest made.

- To use someone else's local, apply it as a global. If cow1 wants to spin right, it can use (0,90,0) as a local. But if you want to spin around cow1's y-axis, you need `Quaternion.AngleAxis(cow1.up, 90)` as a global.

- The associative property works. In other words, parens don't matter. You can write `A*(B*C)` to make it easier to read. It won't change anything.

- Doing it in parts is the same doing it altogether:

  ```
  Quaternion forwardAimCone = z360*y0to10;
  Quaternion meAimCone = transform.rotation*forwardAimCone;
  ```

- There's no way to make B and C both local on A. The only way is a chain. You have to decide which comes first. ABC means C uses AB's local coords. ACB means B uses AC's local coords.

Chapter 6

Moving

We already know the math to use points and offsets to gradually move things. Unity has some standard built-in commands that do the same thing.

We know how to move simple rotations – spinning around y. There are more standard quaternion commands that let us gradually move rotations in more ways.

6.1 Speed per second

If we want nice movement we need a better way of saying how fast we go. "Add 0.05 each update, and we're going whatever speed that works out to" is unreliable. It turns out the best way is to set speed-per-second, adding the correct fraction a little at a time.

The math seems easy enough – there are 60 updates/second, so we divide by 60. But sometimes there are 30/second. Even worse, that could change mid-program, or even mid second.

So what we do is forget about frame-rate and updates per second. Instead we time how long each update takes. If an update takes 1/10th of a second, we add 1/10th of the movement. If the next update takes 1/60th, we add 1/60th.

Since we have no way of knowing how long the current update will take, we use the time passed since the last one. At the start of each update, Unity computes that and saves it in `Time.deltaTime`. A typical value is 0.167 (which is 1/60th).

This code adds 3.5 to x every second:

```
public float x;

void Update() {
  x += 3.5f * Time.deltaTime;
}
```

This moves us forward at 3.5 units/second. In our minds `transform.forward*3.5f;` is our speed for second, and `*Time.deltaTime` is the trick to add it a little at a time:

```
void Update() {
  transform.position += transform.forward*3.5f*Time.deltaTime;
}
```

This next code moves the red cube towards us at whatever speed-per-second we enter:

```
public float speed; // units per second
public Transform redCube;

void Update() {
  Vector3 toUs = transform.position - redCube.position;
  toUs = toUs.normalized; // length 1
  toUs = toUs*speed; // our movement arrow over 1 second

  // standard "add correct fraction of the move arrow":
  redCube.position += toUs*Time.deltaTime;
}
```

An interesting thing – when it arrives it will buzz back and forth. It overshoots, then come back towards us and overshoots again.

6.2 MoveTowards

Unity has a helpful function that does the same thing as that last example. As a bonus, it won't overshoot. The form is `MoveTowards(startPoint, endPoint, moveAmount);`. For example, this moves the red ball towards us by 2 units:

```
redBall.position = Vector3.MoveTowards(
    redBall.position, transform.position, 2);
```

Like any normal function, it doesn't actually move anything. It computes the position. But the form we use is almost always `A=MoveTowards(A,target,amt);`, which moves A. The distance is in real units, not adjusted for anything. To move at 2 units/second we need the deltaTime trick:

```
// 2 units/second (this would be in Update):
redBall.position = Vector3.MoveTowards(
    redBall.position, transform.position, 2*Time.deltaTime);
```

The most fun thing above MoveTowards is how we can easily write it using the vector math we already know. It's a fun review:

```
Vector3 MoveTowards(Vector3 A, Vector3 B, float dist) {
  Vector3 AtoB = B-A;

  // if we would overshoot, stop on B:
  if(AtoB.magnitude<dist) return B;

  // standard unit vectors to go a set distance:
  Vector3 unitArrow = AtoB.normalized;
  return A+unitArrow*dist;
}
```

There's nothing wrong with using MoveTowards as a shortcut. It's a function, with a nice name, for a common calculation. But if we need it to work a little differently, or are fighting with it, we can always go back to the basics.

MoveTowards movement can seem robotic, but only if we use the same speed all the time. We can easily change it. This sets the speed to 1/2 the distance to the target, per second, but at least 1:

```
  // speed is 1/2 the distance to the target, but at least 1:
  float speed = ((goHere.position - transform.position).magnitude)/2;
  if(speed<1) speed=1;

  // same line as before:
  redCube.position=Vector3.MoveTowards(
    redCube.position, transform.position, speed*Time.deltaTime);
```

Or we could start at speed 0, increasing by 3 per second, to a maximum of 8. Whoever sets new targets would should also reset speed to 0:

```
  speed += 3.0f*Time.deltaTime; // speed goes up 3/second
  if(speed>8) speed=8;

  // same line as before:
  redCube.position=Vector3.MoveTowards(
    redCube.position, transform.position, speed*Time.deltaTime);
```

MoveTowards is good when:

- We want to go is a specific point. We often just want to move in a direction., which is easier with simple vectors.
- Speed is in units/second. Sometimes we want percent-based movement (like we did in the early chapters).
- We don't want to overshoot. Sometimes the target is an aiming guide, and we want to go through it and past. MoveTowards always stops.

- We want to change the target mid-move. MoveTowards handles that nicely – it move it towards the new target from where-ever it is now.

There are lots of ways to move. MoveTowards does a nice job handling one way: "speed per second to a point".

6.2.1 Lerp

`Lerp` stands for linear interpolation, which is the math way to say "a spot between 2 points based on a smooth 0-1". `C=Vector3.Lerp(A,B,0.25f);` sets C to a spot 25% between A and B, going along the line between them.

You may have noticed that we could do that back in the first chapter. Lerp is very simple. Written out:

```
Vector3 Lerp(Vector3 start, Vector3 end, float pct) {
  if(pct<0) pct=0; if(pct>1) pct=1; // pct must be 0-1
  Vector3 arrow=end-start;
  return start+arrow*pct;
}
```

A practical Lerp example is picking a random spot on a line. We give the end points and a random 0-1:

```
// Random.value creates a random 0 to 1
Vector3 pos = Vector3.Lerp(p1, p2, Random.value);
```

Lerp is a nice 1-line way to say things like "10% of the way from A to B".

Lerp isn't really a movement command, but it's sometimes used that way. This will move the green cube from us to the red one over 2.5 seconds:

```
float pct0to1=0;

void Update() {
  // this takes 2.5 seconds to go from 0 to 1:
  pct0to1 += 1 / 2.5f * Time.deltaTime;
  if(pct0to1>1) pct0to1=0;

greenCube.position =
   Vector3.Lerp(transform.position, redCube.position, pct0to1);
}
```

Notice how it's up to us to gradually move `pct0to1`. We can call the same MoveTowards over and over, and it will work. But Lerp only works if we increase the last input.

There's a common Lerp hack to get a zingy fast-then-slow movement. This zips a cherry, from whereever it is now, to a fixed point. Notice how cherry is the starting spot, and the thing we move:

```
void Update() {
  cherry.position = Vector3.Lerp(cherry.position, endPoint, 0.05f);
}
```

It says to move yourself 5% of the way each time, which is less as you get closer. It looks very zippy. 5% doesn't seem like much, but it adds up. You can't control the speed or the time it takes. This trick is really only useful for quick visual effects.

6.3 Gradually rotating

So far, we know how to gradually rotate by changing the numbers inside of a `Quaternion.Euler(x,y,z)`. There are more useful ways to spin gradually. We can spin arrows into other arrows, spin quaternions into other quaternions, or take fractions of quaternions.

6.3.1 Spinning an arrow

We can rotate an arrow without needing to make a quaternion or use degrees. Give it the final arrow and gradually spin to match. Like most arrow math, you have to imagine both arrows coming from the same point.

The basic code is like a MoveTowards. The third input is the speed, in radians per second(yikes!), and we'll ignore the 4th input for now:

```
  // gradually spins arrow until it's the same direction as endArrow:
  arrow = Vector3.RotateTowards(arrow, endArrow, 0.02f, 0);
```

The really cool thing is how it computes the best way to rotate. Depending on how endArrow aims, `arrow` might spin right, up, or diagonal. It makes a straight spin in whichever direction is best.

Here's an example showing it in use. The 'A' key puts the ball 4 units in front of us. That could be anywhere. Then it automatically spins that arrow to 4 units straight up:

```
public Transform redCube;
Vector3 arrow; // spins from forward*4 to (0,4,0)

void Update() {
  // 'A' key resets and begins new move:
  if(Input.GetKeyDown(KeyCode.A)) arrow=transform.forward*4;

  arrow=Vector3.RotateTowards(
      arrow, Vector3.up*4, 1.5f*Time.deltaTime, 0);
  // NOTE: 1.5 radians is about 90 degrees
  redCube.position = transform.position+arrow;
}
```

If you press ‘A’, you’ll see the red cube is on a curved path. It’s at the tip of a rotating arrow. If it doesn’t have far to move, the curve is small, but it’s still there.

The 4th input is how fast the arrow changes length. If the target arrow is longer or shorter than you are, you can shrink and grow to match it. 0 means not to change length. That seems pretty cool, but it’s a pain since they don’t auto-synch. You’ll probably finish the rotation, then grow in place until you’re long enough. Or vice-versa.

6.3.2 Moving a rotation

Rotations have a similar move-towards-like function. You give it the current rotation, the target rotation, and how much to move (in degrees, this time.) Here’s a simple example of spinning us towards a target:

```
Quaternion qWant = Quaternion.Euler(-90,0,0); // straight up

void Update() {
  transform.rotation = Quaternion.RotateTowards(
      transform.rotation, qWant, 60*Time.deltaTime);
}
```

This spins us from our current facing to aiming straight up, at 60 degrees/second. Try it with starting in various odd directions. It rotates us the closest way, along a nice, straight curve.

It also does something lines don’t – it matches the z-roll. If you’re on your back, this will roll you head-up. The degrees per second counts z. If you’re aimed mostly the correct way but have the exact opposite z-roll, this will take several seconds as if rolls you over.

Here’s a practical example. We want to look at the red cube by smoothly spinning. We get the rotation to it and smoothly spin ourself to match it:

```
void Update() {
  Vector3 qWant=Quaternion.LookRotation(
      redCube.position-transform.position);

  transform.rotation=Quaternion.RotateTowards(
      transform.rotation, qWant, 30*Time.deltaTime);
}
```

LookAt would make us always face red. This will have us track it, with the possibility it could “out run” us for a while.

This next one is really cute. Pretend our main game is walking around, spinning only on y. We move by changing the float `yFacing`. But sometimes we get knocked down. When that happens we activate our rigidbody, which lets it fall and roll around.

To get up, we turn our rigidbody back off and use RotateTowards to get up:

```
float yFacing; // our normal y-spin
bool isKnockedDown=false;

void getUp() { // call this every frame we’re knocked down:
  // compute the standing up rotation:
  Vector3 qWant=Quaternion.Euler(0,yFacing,0);

  // spin us towards the standing rotation:
  transform.rotation=Quaternion.RotateTowards(
      transform.rotation, qWant, 90*Time.deltaTime);

  check if done:
  if(transform.up.y>0.99f) isKnockedDown=false;
}
```

The last line is a trick to check for standing. transform.up is (0,1,0) when we’re perfectly straight. If we’re the least bit crooked in any way, the y-value will be less than 1.

It looks a little fake, like we have some sort of gyro guidance system to straighten up. But it’s also really cool how it gets right back up.

6.3.3 Rotation lerp

Rotations have their own Lerp. It works like the one we’ve seen for vectors. This finds the rotation 1/2-way between q1 and q2:

```
transform.rotation = Quaternion.Lerp(q1, q2, 0.5f);
// rotation halfway between
```

For example, this faces us 1/2-way between the red and green cube. If finds the rotation to each and averages them:

```
Quaternion qToRed=
  Quaternion.LookRotation(redCube.position-transform.position);
Quaternion qToGreen=
  Quaternion.LookRotation(greenCube.position-transform.position);

Quaternion qHalf=Quaternion.Lerp(redCube,greenCube,0.5f);
transform.rotation = qHalf;
```

That example isn't super useful, since we can already look between 2 things by finding the point in-between them.

Lerp lets us cut an angle in half, which we couldn't do before. The secret is to start with Quaternion.identity. We're averaging ourself with 000:

```
// qHalf is 1/2 of q:
Quaternion qHalf=Quaternion.Lerp(Quaternion.identity, q, 0.5f);

// this is like q/4:
Quaternion qHalf=Quaternion.Lerp(Quaternion.identity, q, 0.25f);
```

We can also use the Lerp hack to make a fast-then-slow rotation:

```
void Update() {
  Vector3 toRed = redCube.position - transform.position;
  Quaternion aimRed = Quaternion.LookRotation(toRed);

  transform.rotation=
      Quaternion.Lerp(transform.rotation, qUp, 0.03f);
  }
}
```

Notice how it's the same form: we're the starting point, and we also move, and we always move 3% of the way.

This can be nice for transitions. Instead of having the camera's angle snap somewhere, this makes it very quickly aim.

6.4 Misc, summary

Vector3.MoveTowards is the main, easiest way to move points in straight lines. Playing with the 3rd `speed` input can make it look pretty nice. Vector3.Lerp is for math. For when you think "I wonder where 30% of the way between A and B is"?

But both are just shortcuts for things we can already do.

Quaternion.RotateTowards is new, and very useful. It gives a perfect spin to any direction, in degrees per second. handles most things – gives you a smooth, constant shortest-way spin, in degrees/second.

Vector3.RotateTowards is about the same thing. Rotations and directions are similar, after all, and you can turn one into the other.

Quaternion.Lerp is useful for math – taking half a rotation. We have no other way to do things like that.

Both Lerps (points and rotations) have an unclamped version, which means the percent can be 1.1 or -0.5. Rotations with this can be useful:

```
// 1.5 times the rotation:
q=Quaternion.LerpUnclamped(Quaternion.identity, q, 1.5);
// the opposite of the rotation:
q=Quaternion.LerpUnclamped(Quaternion.identity, q, -1);
```

Quaternion Lerp has an alternate version, `Quaternion.Slerp` (spherical lerp). They both do the same thing. Supposedly Lerp is faster, but slightly less accurate. I've never seen a difference.

Chapter 7

Misc Math

This section is about miscellaneous things we can do with rotations and arrows, and other various functions you don't see very often.

7.0.1 Finding angles

`Vector3.Angle(v1,v2)` tells you the angle in degrees between two arrows. It treats them as if they were both coming from the same spot, finding the angle of the V they make. It doesn't care about the direction. If one arrow is 30 degrees from another, it could be right, up, diagonal – anywhere on a 30-degree cone.

It's always positive. The largest value is 180, for 2 arrows in opposite directions.

Angle and RotateTowards and AngleAxis are related in a fun way. It turns out that for any 2 arrows coming out of the same spot, there's always one perfect axis at 90 degrees to them both. You could use it with AngleAxis to turn one arrow into the other. The amount you would need to spin is what Vector3.Angle computes.

That's actually how RotateTowards works. It looks at the two arrows, computes the 90-axis they share, and finds the Angle. It then uses AngleAxis. RotateTowards looks so smooth because it really is a simple spin around 1 diagonal axis.

A common use of Angle is checking whether something is in our "vision cone." Find the angle between your forward arrow and an arrow to the target. If the angle is too big, we can't see it:

```
Vector3 toBall = ball.position-transform.position;
float angToBall=Vector3.Angle(transform.forward, toBall);

if(angToBall<30) print("I can see it");
```

It's a little fake. For real a creature may not see in a perfect circle – they might see further sideways than up. But this is often close enough.

My favorite mistake is using the positions instead of the arrows:

```
// this code makes no sense:
float angToBall = Vector3.Angle(transform.position, ball.position);
if(angToBall<30) print("a person at 000 can see both at once");
```

What happens is fun (but not actually important). It treats the positions as arrows, each coming from 000. If finds the angle between them as if you were standing at 000.

We can get the "flat" compass angle by using the old trick of zero-ing out y. This is the trick to not count standing on hills or valleys:

```
Vector3 toBall = ball.position - transform.position;
toBall.y=0;
Vector3 myForward=transform.forward;
myForward.y=0;

float yAngOnly=Vector3.Angle(toBall, myForward);
```

Naturally, this also won't tell you the direction. 90 degrees could be 90 degrees left or right.

`Quaternion.Angle` is the same as `Vector3.Angle` except it also counts the z-roll. If two cows are aimed the same way, but one is on its side, Quaternion.Angle says 90. If the direction is 30 degrees apart and so is the z-roll, the angle is 42.2. If you're running Quaternion.RotateTowards, this is how long it will take. But you usually want Vector3.Angle.

Suppose you had a game where someone has to line up two logs. If you only need them facing the same way, check if `Vector3.Angle(log1.forward, log2.forward)` is small. If you also want to check if they're spun the same way (both mossy side up?) use the quaternion version to compare the exact rotations: `Quaternion.Angle(log1.rotation, log2.rotation)`.

7.0.2 Cross Product

Cross Product finds the axis between two arrows. This is the what Angle and RotateTowards use. You rarely need it – other functions will compute it if they need it.

Another way to think of cross product is as if 2 arrows are spokes on a wheel. Cross product finds the axle. The internet also has lots of pictures of how a cross product looks.

In Unity, `Vector3 cp = Vector3.Cross(v1, v2);` gives it. It's a Vector3, which counts as a direction.

One semi-common use is finding the +/- direction of a rotation. It turns out that an axis can go either way, depending on the direction. The cross product of forward and right is y, but it's either up or down depending on the order:

```
v = Vector3.Cross(Vector3.forward, Vector3.right); // (0,1,0)
v = Vector3.Cross(Vector3.right, Vector3.forward); // (0,-1,0)
```

That makes it so, by the left-hand rule, it's always a positive rotation from the first to the second (that's a bit sneaky, but it's real trig).

This can be used to find the +/- on a compass spin:

```
float degs = Vector3.Angle(v1, v2); // 0 to 180, left or right
// if the cross product goes down, v1 to v2 is counter-clockwise:
if(Vector3.Cross(v1,v2).y<0) degs*=-1;
// degs is now -180 to 180
```

Cross product seems like a strange name for the axle arrow. If called that because the equation to make it multiples (which gives a product) the xyz's in a cross-wise pattern.

7.0.3 Normals

A **normal** is a an arrow that tells you which way a surface is facing. It comes straight out of it, with length one.

For examples, the normal of the floor is `Vector3.up`, and the normal of the right-side wall is an arrow pointing left. For a ball, the normal of any spot is an arrow from the middle, through it.

A normal is always coming from some specific surface. Suppose you have a pyramid – a base and four sides. The base's normal is an arrow pointing down. The side's normals are all pointing up at a diagonal.

There are various ways of getting normals. A raycast can get the normal of any collider it hits. This next one aims down. If it hits a tilted ramp, it will give us a tilted normal (otherwise the floor is just (0,1,0)):

```
RaycastHit RH; // stores raycast data
if(Physics.Raycast(transform.position, Vector3.down, out RH)) {
  Vector3 norm=RH.normal;
  ...
```

Getting the normal of official Unity terrain is a bit of a mess. We can ask for it at any point, but we need to convert the real world xz into a 0-1 percent. This looks up the normal of a Terrain where you're standing:

```
public Terrain ground;

TerrainData gd=ground.terrainData;
Vector3 myPos=transform.position, gPos=ground.transform.position;
// convert my position into 0-1 ground x and y:
float gx=(myPos.x-gPos.x)/gd.size.x;
float gy=(myPos.z-gPos.z)/gd.size.z;
// lookup:
Vector3 gNorm=gd.GetInterpolatedNormal(gx, gy);
```

As usual, it will tend to be (0,1,0). But it will tilt left on the left side of a hill, for example. It can be used to find the uphill and downhill directions.

If you have 3 corners of any flat surface, you can use the cross-product to get the normal:

```
v1


                v2
    v0

Vector3 normal = Vector3.Cross(v1-v0, v2-v0);
```

7.0.4 Reflect

This is getting a little obscure – I've never used it. The `Reflect` command bounces an arrow off of a surface. You pretend the tip of the arrow is against the wall, and Reflect tells you which way it bounces off. It uses the wall's normal to know how to bounce.

This slightly complex code shoots a laser, using a raycast, and bounces off anything it hits named `"wall"`:

```
Vector3 laserDir;

RaycastHit RH = new RaycastHit();
bool gotaHit=false;
if(Physics.Raycast(transform.position, laserDir, out RH)) {
    if(RH.transform.name!="wall") gotaHit=true;
    else {
      // bounce off wall:
      Vector3 hitPos = RH.point;
      Vector3 wallNorm = RH.normal;
      Vector3 dir2=Vector3.Reflect(laserDir, wallNorm);
      print("hit wall, bouncing");
      // now shoot from where we bounced:
      if(Physics.Raycast(hitPos, dir2, out RH)) {
        if(RH.transform.name!="wall") gotaHit=true;
    }
  }
  if(gotaHit) print("We hit "+RH.transform.name);
}
```

7.0.5 Opposite of an angle

`Quaternion.Inverse(q);` is the proper way to write `-q`. Put another way, suppose you want to subtract q2 from q1. Write it like: `q1*Quaternion.Inverse(q2)`.

Of course, it's up to you to figure out whether to subtract it as a local or a global. You may need to use `Quaternion.Inverse(q2)*q1` it you need to subtract it as a global.

You don't need this very often since we can naturally make the opposite of most rotations anyway:

- `FromToRotation(A,B)` is opposite `FromToRotation(B,A)`.
- `Quaternion.Euler(0,-y,0)` is clearly the opposite of `Quaternion.Euler(0,y,0)`.

  But that's not true for xyz. In `Quaternion.Euler(x,y,z)` you can't just flip every xyz. To test this:

  ```
  float x,y,z; // use any values
  Quaternion q1 = Quaternion.Euler(x,y,z);
  Quaternion q2 = Quaternion.Euler(-x,-y,-z);

  print((q1*q2).eulerAngles); // not (000). These are not opposite
  Quaternion q1Inv=Quaternion.Inverse(q1);
  print((q1*q1Inv).eulerAngles); // 000
  print(q1Inv.eulerAngles); // these will be some funny numbers
  ```

- `Quaternion.LerpUnclamped(q, Quaternion.identity, 2.0);` is a longer way to flip any rotation. There's no reason to use it now that we know about Inverse. The 2.0 takes it from `q`, all the way down to 000, then past to `-q`.

We can use Inverse to make Quaternion.LookRotation(q1,q2) (the one we have now only works for vectors). We can make it by "subtracting" q1 from q2:

```
// sample rotations:
Quaternion q1=Quaternion.Euler(30,-60,10);
Quaternion q2=Quaternion.Euler(20,-45,22);

// q2-q1, as a global rotation:
Quaternion q1to2global=Quaternion.Inverse(q1)*q2;
// test using q1 + q12:
```

```
print((q1to2global*q1).eulerAngles); // (20,-45,22)

// q2-q1, on q1's local axes:
Quaternion q1to2Local=q2*Quaternion.Inverse(q1);
// test using q1+q12:
print((q1*q1to2Local).eulerAngles); // (20,-45,22)
```

7.0.6 Square magnitude

There's a built-in function that gives you the *square* of the length of a line. If `v` has length 3, `v.sqrMagnitude` is 9.

That seems insanely pointless. Why would you need your length squared? And if you did, wouldn't it just be easier to write `len*len`?

It's just a trick to speed up the math, and does nothing else useful. Here's the explanation:

The formula to find the length of an arrow is `Mathf.Sqrt(x*x+y*y+z*z)`. In other words, when you run `v.magnitude` the first step gets 9, then the second step square roots it to get 3.

When you use sqrMagnitude, you're saying to save time by only running step 1, and you'll take it from there.

Here's how that can save time: Suppose you want to find the nearest enemy, from you. Find everyone's `sqrMagnitude` and use the smallest. We don't care about the actual distance, so we may as well compare the one that's faster to compute.

The other use: suppose you want to find everything 3 away from you. Take everyone's `sqrMagnitude` and check for 9 or less.

The important thing is, `sqrMagnitude` does nothing useful – it can sometimes be a smidge faster, but makes you think harder to write the program.

7.1 Trig you should almost never use

In school we learned that angles and rotations use trig. They do, but it's almost all built-in to the system. There's rarely a need to use it in 3D.

7.1.1 Radians

Real trig functions, like sin, cosine . . . use radians instead of degrees. They're different in three ways:

- 1 radian is about 57 degrees.
  For real, there are 2PI radians in a circle, which is a repeating decimal: 6.283185 . . . . So 90 degrees is 1.57079 . . . radians. Ug.

- 0 radians is facing along +x (instead of +z).
- Radians go counter-clockwise. 0 is right, 1.57 is forward, 3.14 is left.

What this means is if you have a trig angle of 2, that's 114 degrees, but you're not done. It also starts on +x and goes backwards. In Unity math it's really -24 degrees.

Unity has a built-in `Mathf.Rad2Deg`, but it's just 57. It doesn't take care of the +90 counter-clockwise part. Likewise `Mathf.Deg2Rad` is just the number 1/57.

If you need to convert a trig angle in radians to a unity angle in degrees, or back, it's:

```
degs=-rads*Mathf.Rad2Deg+90; // radians to unity degrees

rads=-degs*Mathf.Deg2Rad+Mathf.PI/2; // unity degrees to radians
```

7.1.2 atan2

In trig, arc-tangent turns a slope into the angle (in radians). But it won't work for 90 degrees and gives the answer+180 for anything aiming left.

A computer trick fixes that. Instead of computing the slope, we give it x and y. This would give us the angle we're facing:

```
Vector3 fwd=transform.forward;
float ang=Mathf.Atan2(fwd.z, fwd.x);
```

That gives `ang` in radians, so we're not done. Ugg.

atan2 is useful in a few special circumstances, but for most problems it's simpler to use other commands.

7.1.3 dot product

Dot product tells you the angle between two arrows, sort of. They have to be length 1 (you can normalize them,) and it gives the cosine of the angle, which can be converted to radians. It also can't tell left from right, the same as Vector3.Angle.

Vector3.Angle is just better.

7.1.4 Rotation matrixes

Another standard computer way of representing rotations is a 4x4 grid, called a Rotation Matrix. In Unity, a `Matrix4x4`.

Graphics cards use these, so you might see them in Shaders. Some of the older GUI items use them. The camera has one.

But quaternions are better for most things. There's no reason to use a rotation matrix unless you're required to.

Chapter 8

Using local space

There are quite a few problems which are only difficult because we could be anywhere, aimed in any direction. If we couldn't rotate, they'd be a lot easier. If we could never move away from 000 they'd be even easier than that.

The Local Space trick says that you can do that. Whenever something is difficult because of your spin, you can pretend you don't have one. While you're at it, pretend you're always at 000. Solve the problem that way, and you can easily turn it into the real answer.

8.0.1 Local space and children

Here's a warm-up using parent/children to place objects. First a quick review:

When you make something a child, Unity starts displaying position using its parent's local space. (000) means you're on top of your parent and (2,0,1) means you're 2 right and 1 forward, using your parent's arrows. When we un-child, we don't move anywhere, but the position numbers snap – Unity recalculates your position in global space.

Now onto the example. Suppose we have a diagonal road and want lampposts and mailboxes along the sides. It you've ever tried something like that using the real x&z arrows, it's a mess.

Here's our plan: create an empty gameObject named `roadLocal`. Put it in the middle of the road, rotated to aim along it. Then temporarily make every lamppost and mailbox a child. Adjust them, and un-child when done. That's the whole trick.

It works because of local coords. Since they're children, sliding mailboxes on x&z goes perfectly along the road. If one mailbox is at (3,0,8) the one on the other side can flip x to be at (-3,0,8).

Basically, we figured out a way to make the road count as if it started at 000 and went perfectly along +z. We used the parent/child trick, which made a perfectly slanted local space that allowed us to pretend.

8.1 Local to global math

Suppose we want to pick a random spot in front of us. We want it to be somewhere in a square area with local z from 3 to 5 and local x between -2 and +2. We could solve that with `transform.forward` and so on, but I'm sick of that.

The plan will be to pretend we're at 000, with no spin. We'll determine the spot, and then adjust to account for where we really are:

```
// pick the spot as if we were at 000, no spin:
Vector3 localSpot;
localSpot.y=0;
localSpot.z=Random.Range(3.0f, 5.0f); // 3-5 in front of us
localSpot.x=Random.Range(-2.0f, 2.0f); // +/-2 right and left

// when done, adjust for our real position and spin:
Vector3 spot = transform.position + transform.rotation*localSpot;
```

The first part is boring, which is good. There isn't a single extra term snuck in to account for spin. It's all from chapter 1. That's why this trick is so great. Doing things as if you were at 000 with no spin tends to be pretty easy.

The last line shifts the spot to account for our position and spin. With very little effort, `spot` is now somewhere in a square, aligned with us, in front of us.

That line is the standard conversion from "pretending at 000"-land to the real coordinate system. We'll see the same line every time we use the trick.

In this next example, we'd like to, once again, spin the red cube from our right side, up and over to our left, then snap back and repeat. First we make it as if we were at 000, no spin. That's easy – spin (4,0,0) around the real z:

```
// simple rotation around z:
degs+=2; if(degs>180) degs=180;
Quaternion zSpin = Quaternion.Euler(0,0,degs);
Vector3 redPos = zSpin*(Vector3.right*4);

redPos = transform.position + transform.rotation*redPos;
```

The first part has no weird diagonal lines and no `transform.right`'s. It's not completely simple, but it's not too bad. When done, the standard line converts it. `redPos` is now moving right to left over us, based on our spin.

In this last one, I want the red cube to shoot straight out from me, starting a little bit up and ahead of me. It should go out 4.6 units from where it started, then repeat:

```
float redDist=0; // 0 to 4.6
```

```
void Update() {
  redDist+=0.6f*Time.deltaTime; // it moves at 0.6 per second
  Vector3 redPos=new Vector3(0,1,1.5f); // starting position
  redPos.z+=redDist; // move forward

  redCube.position = transform.position + transform.rotation*redPos;
}
```

Once again, the code to move the cube is downright boring. `redPos.z+=redDist;` moves us forward. Yawn. Pretending we're always at 000 with no spin takes all the fun out of it.

As usual, the fact that we're not really at 000 with no spin is fully taken care of in the last line. It looks exactly like the last line in every other Local Space code.

8.2 Bringing into local

The other type of Local Space problems are when we want to look at other objects. For example "is that tree to my left"? If we didn't have a rotation, that would be easy – just compare x's. If we were also always at 000 it would be even easier – +x is to my right, -x is to my left.

Once again, the Local Space trick says we can do that.

The conversion lines go in front. They get the xyz's of where the object looks, to us. This would find where a tree is:

```
Vector3 treeLocal = Quaternion.Inverse(transform.rotation)*
    (tree.position-transform.position);

if(treeLocal.x>0) print("tree is to my right");
else if(treeLocal.x<0) print("tree is left of me");
if(treeLocal.z<0) print("tree is behind me");
```

That first line looks even odder, but it's also a standard conversion. We'll see it every time we need to examine another object as if we were at 000 with no spin.

The same as the other version, the math is pretty boring. We're at 000, looking forward. If the `treeLocal` is (2,0,-1) then the tree is 2 spaces to our left and 1 space behind us.

If we needed to check several items, maybe 2 trees, we can save some time by pre-computing the inverse. The math will look about the same:

```
qi = Quaternion.Inverse(transform.rotation); // pre-compute this
Vector3 t1Local = qi*(tree1.position-transform.position);
Vector3 t2Local = qi*(tree2.position-transform.position);
```

```
// check both trees:
if(t1Local.z>0 || t2Local.z>0)
  print("at least one tree is ahead of us");
```

What if we want to know if we're ahead of or behind a truck? That means we care about the truck's rotation, and not ours. We can do the trick as if the truck was at 000 with no rotation:

```
Vector3 meInTruckCoords = Quaternion.Inverse(truck.rotation)*
    (transform.position-truck.position);

// check my position in truck land:
if(meInTruckCoords.z>0) {
  print("in front of truck");
  if(meInTruckCoords.z>5) print("pretty far ahead");
  else if(meInTruckCoords.x>-2 && meInTruckCoords.x<2)
    print("it's going to hit you!!");
}
```

The math in the conversion line was a little tricky. But, as usual, the rest is super easy. If our z is truck-coordinates is positive, we're in front of it. If our truck-coordinates x is between -2 and 2, we're in the path.

8.3 Round trip local space

The most complicated use of local space is when you need to bring something in, adjust it, then bring it back to global. We'll use both equations, in the normal place they go.

Suppose a ball needs to be lined-up exactly on our z-axis. Sometimes it jiggles out-of-line and we need to force it back.

The plan is to bring it into our local space, change x and y to 0. Then compute it back to global.

The code:

```
// get local space ball coords:
qi = Quaternion.Inverse(transform.rotation);
Vector3 ballLocal = qi*(ball.position-transform.position);

// snap back to our centerline, which is easy:
ballLocal.x=0; ballLocal.y=0;

// now convert back to global and put the ball there:
Vector3 b2=transform.position+transform.rotation*ballLocal;
ball.position = b2;
```

The first and last part are the standard conversions. The middle is the work, and it's so easy it doesn't seem like we did anything. If you're at 000 facing along +z, how do you force something to your forward arrow the shortest way? Get rid of its x and y.

8.4 Local Space Theory

As you may have guessed "pretending you're at 000 with no rotation" isn't an official math thing.

For real, we're choosing to move around using our local axes, the same as always. When we're inside the trick and write (0,0,3), we're actually starting at our position and following our personal `transform.forward` for 3 units.

The secret of Local Space is how it feels like pretending we're at 000 with no rotation. When we decide to work in our local space, we mentally fade out the real xyz's. We do math as if our grid is the only one. We purposely ignore the real xyz's until we're all done. The secret is that in every way that matters, our Local Space is a perfect world with us at 000 and no spin.

Suppose we use `Quaternion.Inverse*(tree1.position-transform.position)` and get (-1,0,0). For real, us and the tree are probably way off somewhere, but close together. If we look at how we're rotated, the tree is exactly 1 unit along our left arrow. But once you know all of that, you can forget it. "The tree is at (-1,0,0) in our local space" is a quick and useful way to say all of that.

Later on, `pos.x+=2` really means to walk 2 spaces on our `transform.right`. The Local Space trick says to relax and stop trying to do two things at once. Walk around on Local as if it's a regular grid. Let x+2 just be x+2. Convert later.

The pair of equations – bringing trees in, and moving answers out – are technically local-to-global conversions.
`Quaternion.Inverse*(tree1.position-transform.position)` converts global to local. `transform.position+transform.rotation*A` converts local to global.

8.5 Misc

8.5.1 Converting arrows

We usually bring points into and out of local space. That's what the regular formulas do. But sometimes we need to do that with arrows. The formulas as a little different.

As an example, suppose the real world `windDirection` is (1,0,0). The wind is blowing East. If we need to use it in our local space, it will probably be in a different direction. Likewise, if we compute a local wind direction of (1,0,5), that's in our right and forwards arrows. We'll need to convert to global.

This would bring the wind into local space. It's the usual equation, without subtracting a position:

```
Quaternion qi=Quaternion.Inverse(transform.rotation);
Vector3 windLocal=qi*windDirection;
```

A stranger version for that, suppose we need a ball's forward arrow, in our local space. It's the same equation:

```
Vector3 ballLocalForward =
  Quaternion.Inverse(transform.rotation)*ball.forward;
```

Going the other way, suppose we calculate a velocity in our local space. Multiplying it by our rotation converts it to global:

```
Vector3 localVelocity = ...;

// convert to global:
Vector3 vel=transform.rotation*localVelocity;
ball.GetComponent<Rigidbody>().velocity=vel;
```

Notice it's the same equation as for points, but without adding a position.

8.5.2 Converting rotations

It's not as common, but sometimes we need someone else's rotation, in our global space. To visualize this, suppose we're facing right and a tree is facing forward. To us, the tree is facing left.

The equations are the same as for arrows:

```
// global to local rotation:
Quaternion treeLocalSpin=
    Quaternion.Inverse(transform.rotation)*tree.rotation;

// local to global rotation:
tree.rotation=transform.rotation*treeLocalSpin;
```

Since rotations and forward arrows are almost the same thing, I generally use only forward arrows.

8.6 Built-in shortcuts

Unity has 6 shortcuts for converting points and arrows to and from local space. If you know the equations, there's no reason to use these, but it's interesting to see the names.

These go from local to global:

- `transform.TransformDirection(v);` converts local arrows into global. It's just `transform.rotation*v`.
- `transform.TransformVector(v);` is the same, but also multiplies by your scale. More on that later.
- `transform.TransformPoint(v);` converts a local to global point. But also multiplies by the scale.
- There's no version that only converts local points to global.

The other three go from global to local. They're the opposites of the three above:

- `transform.InverseTransformDirection` converts global arrows into your local space. It's just your inverse rotation times the arrow.
- `InverseTransformVector` is the same, but divides by your scale (yes, divides – that makes the math work out).
- `InverseTransformPoint` converts a point from global to local, and also divides by your scale.

Multiplying and dividing by the scale is funny. Most things have a scale of (1,1,1), so it won't matter. If you changed the scale, those commands will give wrong results.

But Unity adjusts children's local positions based on the parent's scale. If a child's x is 3 and the parent has x-scale 0.5, the the child is really only at x=1.5. The special commands using scale are meant for child objects, I think.

8.7 Summary, notes

If you need to do a few simple movements, `pos+transform.right*2` is just fine. But suppose you need to do more than that. Compare:

```
// working in global:
Vector3 pos=transform.position;
if( ... ) pos+=transform.forward*3;
if( ... ) pos+=transform.right*1.6f+transform.forward;

// using the local trick:
Vector3 pos=Vector3.zero;
if( ... ) pos.z+=3;
if( ... ) { pos.x+=1.6f; pos.z+=1; }
pos = transform.position + transform.rotation*pos;
```

Thinking in local space, even if we didn't have to, has 1 extra line at the end, but makes the rest of it easier to read.

It seems funny that `transform.forward` is in global space. When we use it we're thinking in local space – moving 1 along our z. But then we immediately turn it into the global coordinates. In local space, "1 ahead of me" is simply (0,0,1). We'd convert it to global later.

The trick is only for when whatever it is we're doing depends on our rotation. Suppose we want to find the nearest object. We don't need this trick since our rotation doesn't affect distance. But say we want the nearest where sideways and backwards count as more than forwards (we hate turning and backing up). Now we want Local Space.

The form is always:

```
o compute local coords of anything we need

o do math in local coords

o convert anything we need to use "for real" into global coords
```

Everything is a variation of this. Sometimes we're merely checking something. We convert it in, do our checks in Local, and there's nothing to convert back. Other times we have nothing to bring in – we compute a position in Local and bring it out.

If we need to "fix" a position, we convert it in, adjust it in local, then convert it back. But we might bring in several objects, use them to compute some other spot, and convert only that back for use.

Deep down, the trick is as simple as it sounds "can I solve this if I was at 000 with no spin, then account for my spin later?"

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.