Contents

Chapter 0

Intro

This is about the math for 3D positioning and rotating, and the computer code. There are examples and some real tricks, but the goal is for you to know enough to figure out all of those oddball move and spin problems for yourself. This is about 70% regular math.

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

0.1 Review of basics

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

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. But a 3D cow made for Unity should be facing forward, meaning the z axis runs from it's tail through the head; with +y running from feet to back.

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, or water if you have it, is wherever you program it to be.
- There's nothing special or bad or complicated about negative values. If you use (0,0,0) as the center of your game world, half will be negative, which is fine.
- There are no special out-of-bounds values. The edges of the board or the world are where-ever you program them to be.

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 units are whatever you want them to be, and don't have to stand for anything. For example:

- If you game takes place on a board, 1 unit = 1 square is fine. That way +1x is also +1 square to the right.
- For a game taking place somewhere real, 1 unit = 1 meter might be good. That way you can scale everything to actual measurements. If you prefer yards or inches (the game is about mice?), 1 unit = 1 of those is also fine.
- The 3D models you use aren't a good cue – they tend to have inconsistent scales. You might get a nice set of barnyard animals at 1 unit = 1 meter, then some buildings at 1 unit = 1 foot. They are easily rescaled, and almost always are.
- It's fine to not even know the scale, or even to have one. You can let the camera shows some area, and hand position things using the drag-tools. Everything is just where it is. If you find you need to know the numbers for something later, pasting them into code is fine. The computer won't care if cows are an oddball 0.741 units tall.
- Officially, 1 unit is 1 meter. But that only matters if you use the preset value for gravity (-0.98 on y). It's easy to change, and most people do.

Model origins. If you know 3D models, Unity handles origins in the normal way. Parents solve problems the same as usual. If you don't, here's a summary:

You can use the arrows to drag a Unity Cube where-ever you need, and that works fine. But suppose you place a Cube just touching a wall at x=10. You'll notice the Cube has x=9.5. The premade Unity shapes are centered, and the Cube happens to be 1x1x1, which means it goes half a unit in every direction from where you place it.

In technical terms, the Cube's origin is in its center. When you position it visually, that won't matter. But if the floor is at 0 and your code puts a

centered cube there using y=0, it will be half-way embedded.

It so happens that "centered" isn't the rule. If you import a cow, for example, there's a good chance its origin will be centered between the feet. Each object has its own origin, decided when it was created. The person making that cow found (0,0,0) in the blank screen. Then decided whether to build the cow all around it, or up from it, or maybe up and forward from (0,0,0) (which would make that cow's origin between its back hooves).

For now, it's enough to know that when we place a built-in Cube or Sphere somewhere, we're placing the center. But that if you grab other 3D models, they may intentionally not be centered.

Rotations: 3D rotations can get pretty complicated. The basic idea is you can spin on your x or y or z axis.

Imagine you have a cow on the ground, facing +z (which is considered forward.) Y-rotation turns the cow (remember y is up/down in Unity, so it's like a twirling merry-go-round pole horse.) X runs left-right, so an x-rotation rolls the cow forwards. Z runs forwards/backwards, along the length of the cow, so a z-rotation tips the cow sideways, like it's on a barbecue spit.

Objects also rotate around their origins. Unity shapes will spin nicely on all axes, since they have centered origins. But take the bottom-origin cow. It will spin fine on y, but an x or z rotation spins the cow around it's feet. If it's standing on the ground, a forward roll will put it completely underground before coming back up.

This is still not really important. One thing is knowing a funny-looking cow rotation simply means a non-centered origin (which you should be able to pick out if you watch to rotation for a few spins). Another is if we want a non-centered spin for a centered Cube, we merely need to change the origin (using a trick, below).

For y-rotations, Unity thinks 0 degrees is forward, along +z, and then it goes clockwise. So 90 degrees is facing right, 180 faces backwards (as you'd expect), and 270 faces left.

The rule for x&z spin direction is a left-handed coordinate system. You can look up some nice pictures of this. Here's a short version: wrap your left hand around the axis, thumb pointing positive. The curve of your fingers is the plus rotation direction. If you try it for y (like grabbing a lamp pole in front of you, thumb up) your fingers go clockwise.

For x you'll be grabbing a sideways line, thumb aimed right (like you're showing someone a briefcase). Your fingers will curl up and over. This means a +x rotation tilts down into the ground. To tilt backwards, making the cow look upwards, use -x.

z is the funniest. Grab a line coming out from you, thumb forward (an awkward underhand, for me). You fingers curl up and left – meaning +z rotations

tips you to the left.

If you know real trig, this entire system of degrees and clockwise might seem funny. It's done to be simpler for Game designers, and all normal Unity API's use it. Unity's has built-in sin, cosin ... and it's all correct trigonometry. If you decide to mix trig and Unity-rotations, which you rarely do, you'll have to convert back and forth.

Problems with real 3D models

We'll only use Cubes and Spheres, which are fine for testing. But a real game will need some cows brought in, and they can have all sorts of problems.

3D modeling programs tend to use flipped axes from Unity3D: z=up and x&y as the ground. You can use a cow made that way, but Unity thinks the cow's back (which is the +z part) counts as forward. It will be on it's tail, feet facing us. We could fix it by hand-spinning. But the y-axis is still running the wrong way – if we try to turn the cow, it will barbecue-spin instead. For a moving cow, we need to fix the wrong axis.

Modeling programs also like to use 1 unit = 1 centimeter. An excellent, cheap cow-maker you hire will tend to give you a 220-unit long cow. You can hand-fix the scale; but you have the same problem if your code tries to tweak the size. Fun fact: most 3D models are see-through from the inside. That super-huge cow might go around your entire game, and you won't even be able to see it before scaling it down.

The parent trick

Most real imported objects will need an adjustment to their size, rotation or origin. Even if something is perfect, you'll probably need another version of it with things shifted around. You can adjust these all at once using a parent. This is also how 3D modeling programs do it.

Suppose you have a wrong-size, wrong-facing, wrong-origin cow. To fix it, create an empty object, maybe named cow1. Leave it normal scale and no rotation, and place it in any spot you find helpful. Make the cow a child of it. Take that child cow and move, resize and spin it to how you want it. Then never change the child cow again.

Now the parent, cow1, acts like the cow you wanted. It drags around the real cow, keeping the adjustments you made. It appears to have the correct origin, and scale, and axes. The adjustments are hidden away in how the real child cow connects to cow1.

You can even do this to make other versions. A new empty, cow2, can have another copy of that cow inside, but with a different origin scale and axes (maybe for a menu-icon cow, which needs to be small and right-facing).

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;
```

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*. Each pair adds, and the result also has three parts. More specific: add x to x, y to y and z to z. 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)
```

It's useful, and we'll use it lots, but it's only a shortcut. `C=A+B;` is the same as `C.x=A.x+B.x; C.y=A.y+B.y; C.z=A.z+B.z;`.

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 that 3 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;`

Unity doesn't define pairwise multiplication: `A*B` isn't allowed. If you need it, which you rarely will, you can write it out in three lines. It also doesn't define scalar addition, such as `A+2`, but you also rarely need it.

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*0.5f` is an easy way to make (0.5, 0.5, 0.5).

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 for creating points. `Vector3.up*5` is a nice-looking way to write (0,5,0). Or we can combine them (scalars and addition) to make any point:

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

That's the same as `new Vector3(0,9,-4);`, except longer. But maybe it's easier to read it as "4 backwards and 9 up".

1.1 Points and Offsets

The first trick is knowing an x,y,z can be either a point, or an offset, depending on how we think about it. A point is the easy one – a spot on the map. Offsets

are arrows, which we add to points. An offset of (0,5,0) means "5 above some other spot".

`Vector3.right` and it's friends could be used as either, but are usually offsets, which is why they have those names. A typical use would be `transform.position+Vector3.right`, meaning "one space right, from me".

Suppose we're building a game board. We'll start by letting the user set the lower-left corner. It's a simple point:

```
public Vector3 cornerLL; // user will enter
```

The board squares will run sideways and forward from there. To help place them we'll make two offsets:

```
Vector3 sqSdways=new Vector3(1.2f, 0, 0); // arrow for 1 square over
Vector3 sqFwd=new Vector3(0, 0, 1.2f); // arrow for 1 square forward
```

These are offsets because of how we plan to use them. `sqSdways` isn't a spot on the map – the board may be nowhere near the point (0,0,1.2). But it's great as a single-square sideways arrow.

Using our vector math, we can compute positions of some sample squares. We start at the corner, and walk squares over:

```
s10 = cornerLL + sqSdways;
s30 = cornerLL + sqSdways*3;
```

The second one is so cool, how it clearly says "three squares sideways from the corner". It also shows a rule: an arrow times a scalar is going the same direction, but a different distance.

Adding two offsets is like following two different end-to-end arrows. This line says to start at the corner and walk 3 sideways and 2 forwards:

```
s32 = cornerLL + sqSdways*3 + sqFwd*2;
```

It puts us exactly at the corner of that particular square.

For more fun, assume we're placing 3D square models, which have their origin centered. Instead of walking to corners, we need to walk to centers – an extra 1/2 a square sideways and forwards. We can do that painlessly with another offset vector:

```
Vector3 cornerToCenter = sqSdways/2 + sqFwd/2;
```

The end result is a short diagonal arrow. Dividing by 2 is using a scaler – it's the same as `*0.5f`, but I though it looked nicer this way.

To place our squares we add that extra offset:

```
sq00 = cornerLL + cornerToCenter;
 ...
sq32 = cornerLL + cornerToCenter + sqSdways*3 + sqFwd*2;
```

```
              o
              | fwd*2
      sdway*3 |
      -- -- --
cToc /
   LL
```

1.1.1 More calculating offsets

A board lined-up on x and z isn't really showing the advantage of offsets very well. We know sideways is just +x and forward is +z. But with a tilted board, vector math is our best friend.

Instead of only one corner, let's have someone hand-enter three corners. Depending, the board could be tilted any which way (assume they make both sides the same length, and at right angles):

```
// lower-left, upper-left, lower-right
public Vector3 cornerLL; // lower-left corner from before
public Vector3 cornerUL, cornerLR;
```

```
UL
 \    diagonal
  \  game board
   \           ---LR
    \      --/
     LL--/
```

Since we know the board is 8 squares across, each square is 1/8th of the way. Instead of entering the sideways and forward square offsets, we can compute them:

```
Vector3 sqSdways = (cornerLR-cornerLL)/8;
Vector3 sqFwd = (cornerUL-cornerLL)/8; // same
```

The new thing here is that subtracting 2 points creates an arrow. The right corner minus the left corner computes an arrow along the whole bottom.

1/8th of that arrow is our sideways 1-square arrow. Pretty slick.

The really neat thing is how finding the square centers is the same as before:

```
// sample placement (same math as before):
cornerToCenter = sqSdways/2 + sqFwd/2;

sq32 = cornerLL + cornerToCenter + sqSdways*3 + sqFwd*2;
```

One more neat arrow trick: we can compute the missing upper-right corner, using the other three. The plan is: get the arrow going up along the left side; then add that to the lower-right corner:

```
Vector3 bottomToTop = cornerUL-cornerLL; // arrow up left-side

Vector3 cornerUR = cornerLR+bottomToTop;
```

Another plan would be to start at the upper left, and march 8 squares over:

```
Vector3 cornerUR = cornerUL+sqSdways*8;
```

One last thing about this entire game board example: if we knew how to rotate an arrow, we could have them enter only the two bottom corners. We'd find the arrow between them, rotate it 90 degrees, and use that to get the other two corners.

1.2 `transform.position` + offset

Let's assume we're a Cube, with a script on us. Our location, which is a point, is in `transform.position`. We can use that, plus offsets, to place things around us.

Assume that besides us, we have three cubes: red, green and blue. No scripts, just Cubes waiting for us to reach out and position them. This simple code places them at various spots around us:

```
public Transform redCube, greenCube, blueCube;
// assume hand-linked to real cubes through Inspector

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

  Vector3 greenOff = new Vector3(4, 1, 0.5f);
  greenCube.position = transform.position + greenOff;

  // blue piggy-back off green:
  blueCube.position = transform.position + greenOff*1.5f;

}
```

The point+offset math is nothing new. But now we're really placing something there.

If you want to see this really work, put a Rigidbody on us, add a bouncy material, make a floor, a ramp ... and let us knock around. The colored cubes will track us perfectly. Even when we spin, they stay locked in.

For a variant, let's say someone enters one long arrow, coming from us, where we want the colored cubes evenly spaced. We can do that with scalars:

```
public cubeLine;
// ex: (2,8,0) stacks the cubes up, leaning right, with gaps

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

The scalar determines where they go on the line.

A cool idea that gives us: what if we take just one cube and slide the scalar from 0 to 1 (in the code)? It will move along the line:

```
public Vector3 cubeLine;
float pct = 0.0f; // we'll make this go from 0 to 1
public Transform greenCube;

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

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

Stepping back, writing code to shoot out along any line seems like it might involve lots of math. But thinking of it as offsets and scalars makes it easy.

To jazz that up, we can use the trick of computing an arrow between any two points. Instead of entering `cubeLine`, let's say the red cube is the other end. Where ever we are, we want the green cube to shoot from us to the red one.

The only change is computing `cubeLine` as the red cube's position minus ours:

```
// shoot green cube from us to the red one:
float pct = 0.0f;
public Transform redCube, greenCube;

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;
}
```

This next one is simpler. Multiplying by a negative scalar flips an arrow. We'd like the green cube to hide behind us (from the red cube). We'll can take that same arrow from us to red, and take it times -0.1:

```
// green cube hiding behind us from red cube:
  Vector3 toRed = redCube.position - transform.position;

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

1.2.1 Camera vector positioning

Instead of placing a green cube nearby, we can use the same tricks to place the camera. It will appear to be tracking us.

This code puts the camera above us, and a little ahead:

```
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;
}
```

The camera would need to be aimed downwards (a 90 degree spin on x). If we have the rigidbody bouncey set-up from before, the camera-tracking looks pretty cool. But if you go to Scene view and watch, it's no different than when we made the cubes track us.

We can give it a camera zoom by scaling the offset. We'll add a `zoomPct` variable and have up/down arrow keys change it:

```
public float zoomPct=1; // 1=full length
// zoomAmt -- to camera arrow multiplied by this

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

  // arrow keys zoom in/out:
```

```
  if(Input.getKey(KeyCode.UpArrow) zoomPct-=0.01f;
  if(Input.getKey(KeyCode.DownArrow) zoomPct+=0.01f;
}
```

Real zooms don't go straight to our center, like this does. They have a little offset – like going to a spot above our shoulder or something. That's easy to make by using two arrows, and scaling only the second one:

```
  |
  | offset#2, zooms in/out
  /
 / offset#1, always this long
Player
```

The code is pretty simple: add offset#1, then offset#2 times the zoom percent:

```
  Vector3 toCam1 = Vector3.up*8+Vector3.forward*3;
  Vector3 toCam2 = Vector3.up*10;

  theCamera.position = transform.position + toCam1 + toCam2*zoom;
```

Mathwise, it's just a point plus two offsets, which we're done before. We just never had a reason to scale only one of the arrows.

1.2.2 Moving with vectors

Vectors are a nice way of storing "this is how much I move each frame". They say which direction, and how fast. We like to say the arrow is how much we move each second. Each update we'll move a fraction of that.

This moves the green cube along whatever vector the user enters:

```
public Vector3 greenMvArrow;
public Transform greenCube;

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

I like how the += feels like what it does. `x+=0.1f;` moves `x` a little. `greenCube.position += someVector3;` does the same thing, except in 3D space.

Of course, we could move ourself, using the same method, with `transform.position += mvArrow/60;`

We we moving the green cube before, using a more complicated method. We needed to know both end points (us and the red cube) and the percent. It always stayed on the line, even if one end moved. Since we always knew what

percent of the way we were, it was easy to stop when we got there, and go back to the start.

This method is simpler, for when we're happy to just go in a direction.

One note: I'm assuming Update runs 60 times/second. If you know about `Time.deltaTime` (or want to look up that trick), we'd definitely use that instead of 60.

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. If 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;
```

The code is a little longer, but we can adjust the 0.5 to whatever we need. Average is really just a shortcut for this, when we're sure we'll only want 50% of the way and won't need to adjust it later.

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 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, `(A+B)/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 you have `sq.position=A+B+C;` and it's working wrong. Check whether A counts as a point, and B and C are offsets. That's the only way the math makes sense. If something with `B*2` is working funny, check "does B count as an arrow?" and "am I wanting to double how long it is?"

Finally, two fun errors. It's easy to accidentally use an offset by itself, forgetting to to add the starting point:

```
greenCube.position = toRed*pct; // <- oops
// Forgot to start at our position
```

This is like moving the line to start at (000). It's the same length and direction, but in the wrong place. It tends to be really confusing since 000 usually isn't any special spot in your game. Depending where you are, the line can be nearly correct or completely off the screen.

If you get a line that appears to shift in funny way, check that you added the starting point.

It's also easy to flip the arrow direction by mistake. It's the end point minus the start point. If you see `transform.position - redCube.position`, that's as arrow going *to* you, *from* the red cube. Nothing wrong with that, but if you follow it from you, it goes away instead of towards.

Chapter 2

Local and Global coordinates

Local coordinates are a geometry trick for handling math from someone's point of view.

A common example is wanting to know if a tree is to our left. All we need to do is put it on our personal xyz grid and check whether x is negative.

We don't have a personal xyz grid, but it's easy to pretend we do. It's known as our local coordinates. In them, 000 is always where we are, +z is always ahead of us, and so on. Standard math can translate back-and-forth to the real xyz. Lots of things that are hard to do using the real xyz are easy using our personal grid. We translate back when done.

2.1 Local/global translate tools

The first way most people see local axes are the drag arrows in edit mode.

To see them, get in Scene view, select any object and pick the Translate tool (on top: it's the Crossed arrows, between the Hand the the Circle arrows). The red, green and blue arrows show the x, y and z axes (the colors are always in that order – red is always x, and so on).

The next group of buttons, to the right, should say Pivot and Global. Global means to use the real x,y,z. Clicking it toggles to Local, then back to Global.

If you spin the object, then toggle, you'll see the difference in the arrows. Global is always the same – the real arrows. Local depends on your spin: blue (+z) always comes out of your front, and red (+x) is always to your right. If you tip yourself, green (+y) is your personal up and not the real one.

But these are just a handy way to move – lots of people like to slide things the way they're facing. For real, the system only stores the one real set of x, y,

z's. You can watch the Inspector numbers to check this.

This Local/Global feature is actually standard on any 3D program.

2.2 Using your local axis in code

In code, Unity precomputes the 3 local-axis arrows for everything. For you, they're `transform.right`, `transform.forward` and `transform.up`.

To compare, remember `Vector3.right` is the global x. It's always a boring (1,0,0). `transform.right` is your personal x.

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;
}
```

Those local arrows are length 1, so `*3` scales it to length 3 like normal. Also like normal, we can combine them. This puts the red cube 2 in front and 4 left:

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

Using `right*-4` for left is the normal way. It feels like setting x=-4. And by now everyone knows the flipping an offset trick.

This makes us move the way we're facing. Nothing special, but fun:

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

Every object has it's own, which you look up using their transforms. This moves the red and green cubes along their forwards:

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

You're allowed to mix&match, but there's rarely a good reason. This makes us move in the direction the red cube is facing:

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

We won't jump to the red cube. It's only acting like a remote control, telling us which direction we go. Which is just silly.

This positions the green cube from us, based on the blue and red cubes' facings. It's legal, but the final position is nonsense:

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

Those two vectors probably aren't even at right angles, and they have no relation to us or the green cube. We're just following two random arrows.

But here's a fun one where it makes sense to use different arrows. A chain of cubes where each is in front of the other:

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

The pattern "me + my forward*aNumber" makes sense. If we want to spin the cubes to make a zig-zag, that's our business.

An example where we move something along a line won't show anything new, but it's fun. This slides the green cube alongside of us:

```
public Transform greenCube;

float dist=-3; // will go from -3 to 5

void Update() {
  Vector3 toGreen = transform.right + transform.forward*dist;
  greenCube.position = transform.position + toGreen;

  // moves dist from -3 to 5:
  dist+=0.02f;
  if(dist>5) dist=-3;
}
```

If you like the percent-based version, we can also do it that way. Nothing new, but it's fun:

```
float pct=0; // from 0 to 1

void Update() {
  Vector3 moveLine = transform.forward*8; // the entire line
  Vector3 toStart = transform.right + transform.forward*-3;

  greenCube.position = transform.position + toStart + moveLine*pct;

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

It's another one where we have one normal offset, then another scaled.

2.3 Intro to local space theory

The official way to think of your local space is by using normal (x,y,z) vector3's. Two spaces right and one forward is (2,0,1) and we just know it's local space.

When we're done, we translate into real space. Here's a fun function to do that:

```
Vector3 myLocalToReal(Translate tt, Vector3 localOffset) {
  Vector3 pos = tt.position;

  // combine our three arrows, scaled by what they said:
  pos += tt.right*localOffset.x;
  pos += tt.up*localOffset.y;
  pos += tt.forward*localOffset.z;

  return pos;
}
```

It does the same math we were doing before. The difference is how easy it makes it to use just (2,0,1) for local:

```
  Vector3 v = new Vector3(2,0,1); // counts as local
  // put it (2,0,1) in my local:
  greenCube.position = myLocalToReal(transform, v);
```

In general, we like to be able to "think" in local space, using as many vector3's for local x,y,z as we need. `v.x+=1;` moves one more space to our right, and so on.

When we're completely finished, one final standard step converts to real.

2.4 Other Unity commands and local coords

Unity's built-in Translate command uses local coordinates. This moves us forward (our forward):

```
transform.Translate(0,0,1);
```

`transform.Translate(-0.1f, 0, 1);` is forward and drifts a little left.

It's easier than the `transform.forward` method, but more limited – we can't use it to place items.

When you're a rigidbody, Unity gives a command for either way:

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

They thought `Relative` sounded better than local. The Translate command has more synonyms for local/global, in an optional second parameter:

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

Sometimes the real xyz grid is called World Space, and the one based on us is our Local Space.

To sum up: you'll often seen two versions of a coordinate-using command. The exact words may change, but one is global, the other local.

And if you know vector math and offsets, it's often easier to use that.

2.5 Childing, `localPosition`

When an object has a parent, Unity displays it's position in the parent's local space. You may have noticed this: take a Cube just 1 unit in front of another, and drag to make it a child. It's Position numbers change to (0,0,1).

It's a very funny semi-glitch. They could add an extra slot for things with parents, or at least relabel it. But everyone (who uses 3D software) knows the rule about it now being local.

A fun way to use this: make one thing a child of another and enter 000 for Position. It snaps directly to the object – 0 units away from it, in all directions.

In code there's an extra variable for that, named `localPosition`. This moves us forward, the way our parent is facing:

```
  // pretend we are a child of something
  transform.localPosition += Vector3.forward;
```

It's sneaky, at first. Imagine it runs for a while and gets to plain old (0,0,5). The system knows that's an offset, from your parent, the way it's facing. If your parent faces diagonal, you move diagonal.

Here's the "slide a ball from -3 to 5" example, written if the red cube is a child. All of the local position math is gone, since the system does it:

```
public Transform redCube; // assume this is a child of us
float zz=-3; // moves from -3 to 5

void Update() {
  redCube.localPosition = new Vector3(1, 0, zz);
          ^^^^^^^^^^^^^ this makes it work

  // same code as before, moving zz
  zz+=0.02f;
  if(zz>5) zz=-3;
}
```

You're suppose to see `localPosition` and immediately understand x=1 means one to our left, and `zz` means our forwards.

Overall, the idea is that you made it a child for a reason. Expressing children's positions in the parent's local space quickly becomes natural, and the easiest way to get most things done.

2.6 Looking at the numbers

This section is very optional, if you aren't happy unless you see numbers.

Suppose we're tiled 45 degrees on y:

- Our forward will be (0.707, 0, 0.707). You might recognize those as the sin and cos of 45 degrees.
- `transform.up` will be just (0,1,0). Since we didn't tip or lean, our up is still the real up. Nothing wrong with that.
- `transform.forward + transform.right` makes an upside-down V. It ends on the x-axis. But that's obviously correct: for us tilted 45 degrees, the real x shoots right and forward.

This simple program shows values for your local arrows. It uses the trick of copying into Inspector variables every frame:

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

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

With no rotation, you'll see they're the same as the real ones. Spinning 180 degrees flips some signs (forward will be (0,0,-1)). 90 degrees moves them around: forward becomes +x.

If you know sin/cos of 30 degrees by heart, spinning 30 degrees will make (0.5, 0, 0.86).

The manual says `transform.forward` is in *world* space. What?? That's really saying it's ready to use. We think (0,0,2) in local space, then we write `transform.forward*2` to get the world space translation.

2.7 Errors

Mixing local and world axes in the same math usually gives junk, for example: `transform.right*3 + Vector3.right*2`. It's not illegal – 3 to my right, then +2 on x. But it's almost never useful - usually a sign you made a mistake.

It's easy to use `Vector3.up` instead of `transform.up` and not notice the problem. If you never tilt or lean, they're always the same. Sometimes I write `Vector3.up` to purposely get funny results if I tip, since I should never tip.

If something expects local coordinates, you can't use `transform.forward` and friends. For example:

```
red.localPosition = new Vector3(0,0,1); // 1 forward
red.localPosition = transform.forward; // yikes
```

The entire point of using localPosition is to let the system convert for us. `transform.forward` is us converting by hand. It turns (0,0,1) into different numbers, then the system converts them again.

The end result is a weird double-rotation.

The same problem with AddForce. These both push us forward. In the first one, we account for our rotation. In the second, it does:

```
// two same ways to push us on our forward:
rb.AddForce(transform.forward*6);
rb.AddRelativeForce(new Vector3(0,0,6));
```

But `rb.AddRelativeForce(transform.forward);` double-converts.

These become easier with practice.

2.8 Bonus 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.

Pretend we have some tiny rigidbody balls as prefabs, ready to be used as shots. This positions them at the wing tips:

```
public Transform ballPrefab;

void Update() {
  // space 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;
```

Nothing new here, math-wise. I almost used negative rightWing for the left, except that would put it -5 x and *behind* us by 2.

The speeds are both 10 forward, then -2x and +2x to angle them inward. The math part is still nothing new:

```
    // the rest of firing 2 wing-blasts:
    bRight.GetComponent<Rigidbody>().velocity =
      transform.forward*10 + transform.right*-2;

    bLeft.GetComponent<Rigidbody>().velocity =
      transform.forward*10 + transform.right*2;
  }
}
```

Only the +2 and -2's are flipped. Really, it's just two boring lines coming from our wingtips. Except setting them as the `velocity` makes the balls automatically shoot that way.

One note: rigidbody's start with gravity turned on. That's fine, but for a spacelike feel, uncheck that box.

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

As a shortcut, Unity lets us measure distance either way: length of an arrow, or distance between two points. `Vector3.Distance(A,B)` or `C.magnitude` (no parens, just because.) Examples:

```
float dist = Vector3.Distance(transform.position, marker.position);
print("You are "+dist+" away from the marker");

// same thing: get arrow to marker, then measure it:
Vector3 toMarker=marker.position-transform.position; // arrow from us to marker
dist = toMarker.magnitude; // length of arrow = distance
```

The 1-line call to `Distance` looks nicer since it's a shortcut. Inside, it's subtracting the points and running `magnitude`.

We don't have to use these to measure between objects. They work on any points or arrows, even ones we just make. You might remember this from the pythagorean theorem (a right triangle with sides 3 and 4 has hypotenuse 5):

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

`A`'s not really an arrow – we didn't subtract 2 points to get it. Maybe we intend to use it as one .... But either way `magnitude` gives the length as if it were an arrow.

Here are some fun facts about distance and magnitude:

- Distance is always one positive number. Negative distance make 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 order in `Distance` doesn't matter. `Vector3.Distance(A,B)` is the same as `Vector3.Distance(B,A);`
- `magnitude` is for offsets. If you use it with a point, like `(transform.position).magnitude`, it gives the distance from (0,0,0), which is rarely useful. Plus, a clearer way is `Vector3.Distance(transform.position, Vector3.zero)`.
- You can get flat xz distance (distance on a map, not counting hills) by changing the arrow's y to 0:

  ```
  Vector3 toMarker = marker.position-transform.position;
  toMarker.y=0; // now toMarker is a flat arrow
  float dist = toMarker.magnitude;
  ```

- Just so you know, **magnitude** is the official mathematical term for the length of an arrow.
- Also just so you know, not having ()-parens after `A.magnitude` is the official rule. It's a real function call, but using `C#`'s getter trick to leave them out.

3.1.1 Direction

Often we don't need an arrow going all the way to the target. We need one pointing to the target, which we often call a direction arrow. To simplify, direction arrows are usually length 1. For example, `transform.forward` is our forward direction arrow.

The important thing is that direction arrows only tell us which way to go. The length is unimportant. For example (2,1,0) and (4,2,0) are the same direction. If you wanted to write that direction (twice as far x as y) the official way,

you should use trig to make it length 1, but don't have to.

Raycasts are a nice example of using direction arrows. They take a position and a direction and walk that way until they hit something. This shoots an imaginary ray north from us:

```
Vector3 dir=new Vector3(0,0,10);
if(Physics.Raycast(transform.position, dir))
  print("forward is blocked");
```

Raycasts think the second input is a direction. (0,0,10) is the forward, +z direction. It would say we were blocked if anything was 1 in front of us, or 10 or any distance. We could have used (0,0,1) for the direction and it would run the same.

If we wanted to check for obstacles sideways and up, `dir` could be (2,1,0) or (4,2,0).

For a comparison, `Debug.DrawRay` takes an actual offset. It starts at the point you give it, then adds the offset and draws exactly that arrow. In this case, the 10 really means 10:

```
Vector3 dir=new Vector3(0,0,10);
Debug.DrawRay(transform.position, dir); // draws length 10 forward arrow
dir=new Vector3(0,0,0.333f);
Debug.DrawRay(transform.position, dir); // draws very short forward arrow
```

Having Raycast and DrawRay work differently is for sure confusing. But it does a nice job of showing the terms: offset means we care about the whole arrow and where the tip ends, and direction means we don't.

(Funny story: in the manual DrawRay says it takes a direction, but that's a typo. It's an offset.)

3.1.2 Normalized direction

You can often use a direction arrow of any length, but there are some tricks you can do with an arrow of exactly length 1. The math term for that is a normalized direction, or sometimes a unit vector (which is shorthand for "a 1-unit long vector.")

There's a really slick trick to turn an arrow into length 1: divide by its length. Here's an example getting a length 1 direction to a marker:

```
Vector3 toMarker = marker.position - transform.position;
float dist = toMarker.magnitude;
Vector dirToMarker = toMarker/dist; // length 1 arrow to marker
```

This trick works for any arrow – it can be pointing backwards, or have length less than 1 (it will grow) – and it still works.

Unity provides two shortcut functions for that. One of them makes *you* be length one, and another makes a length one version of you. Examples:

```
A = new Vector3(3,4,0);
A.Normalize(); // A is now (0.6, 0.8, 0), which happens to be length 1

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

A=A.normalized; // same as A.Normalize();
```

Normalizing is the ten dollar math term for getting a length 1 version of an arrow. But it's nothing special besides dividing by the length. For example, (1,0,0) is normalized, which is a fancy way of saying it's already length 1.

Some normalizing notes:

- Normalize something that's already length 1 doesn't change it. It doesn't do any harm, either – it's safe to normalize something just in case.
- Most people use the official `normalize` command. But if you already know the length, `A=A/len;` works fine and is faster.
- 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. Unity just gives you (0,0,0), which isn't correct, but it's the best it can do.

3.2 Normalized direction + length

With the theory out of the way, we're ready to do tricks by breaking an offset into length one direction and magnitude. We start with this:

```
Vector3 toB = B-A;
float len = toB.magnitude;
toB = toB.normalized;
```

Now `toB` is a length 1 direction arrow towards B, and `len` is how far.

The simplest trick is that `A+toB` is *one* unit from A towards B. `A+toB*3.5f` is exactly 3.5 units from A towards B. We can pick the exact distance in `A+toB*dist`. We can slide `dist` from 0 to the total, `len`, to walk a real distance from A to B.

Here are few simple examples. This puts a "shield" two units away from us, facing the marker:

```
// get length 1 arrow to marker, all in one line:
Vector3 toMarker = (marker.position-transform.position).normalized;
```

```
shield.position=transform.position+toMarker*2;
shield.LookAt(marker); // not needed, but fun
```

You might remember from before we could use a fraction of a vector to do something similar. The improvement here is we can give an actual distance.

Our old cube-hiding was behind us 1/10th of the distance to what it was hiding from. That made it closer or further, depending. Not what we wanted. Distance math lets it always hide 3 units behind:

```
Vector3 toMarker = (marker.position-transform.position).normalized;
// hiding from marker, behind us and 3 away:
hidingCube.position=transform.position - toMarker*3;
```

This next example slides a ball from us to a marker. We did that before, but now it moves at a constant rate (before, it took about 2 seconds, no matter how close or far we were):

```
public Transform ball;
float ballDist=0; // this is the actual distance from us, in units

void Update() {
  // direction and distance to marker:
  Vector3 toMarker = marker.position-transform.position;
  float len=toMarker.magnitude;
  toMarker.Normalize();

  ball.position=transform.position+toMarker*ballDist;

  // slide ballDist from 0 to total len, at 2 units/sec:
  ballDist+=0.01f; // this many meters per frame
  if(ballDist>len) ballDist=0;
}
```

This way looks better for most things – moving twice as far finally takes twice as long. The percent method is nice for displays – faster movement gives a hint that the target is further away.

The unit vector trick is also great for throwing a rigidbody at something. Before, we could shoot a ball at 10 units/second in our forward direction; but not towards a target. Now we can.

This fires a ball at a marker, always with a speed of 5:

```
if(Input.GetKeyDown(KeyCode.Space)) { // space key fires
  // start the ball a little ahead of us:
  Transform ball = Instantiate(ballPrefab);
```

```
  ball.position = transform.position + transform.forward*2

  Vector3 toMarker=(marker.position-ball.position).normalized;
  Vector3 vel = toMarker*5; // unit vector times speed
  ball.GetComponent<Rigidbody>().velocity = vel;
}
```

Notice how it finds the total arrow from the *ball* to the marker, not from the player. Otherwise the angle might be off. We don't bother computing the distance to the marker, since we didn't need it.

3.3 Looking at the numbers

Sometimes you look at the numbers for distance, and they seem funny. If you never do, skip this.

A surprising thing is, 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. Going up by one added just 0.05 to the distance. If you flip it around, this makes sense: imagine driving to a town 10 miles east and 1 mile north. That's 11 miles if we have to drive that way. But we know a diagonal straight-shot road will be a good deal – it will be just a little longer than 10 miles.

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. The answer had to be at least 5, and the 3 and 4 didn't add much.

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. So it's like an arrow (8,3,1). The distance will be 8 plus a little more.

For real, distances are computed using the Pythagorean theorem: $x^2 + y^2 = d^2$. That's why (3,4,0) has length 5.

Normalized (length one) vectors have the same funny-looking math as `transform.forward`. A unit diagonal arrow really is (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.

But, to repeat myself, you don't need to know these numbers. If you know trig or want to learn, it's fun to look at them. If you notice (0.6, 0.8, 0) and think "wait, aren't they suppose to be length 1?" now you know they don't add to one – they pythagorean square add to 1.

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.

For testing, we'll want a good reference object – something where we can easily tell which way it's aiming, and see top/bottom at a glance. If you have a 3D cow or gun pointing along +z, that will be fine. Otherwise I like to make a testing object:

Make a sphere. Then add a cube above it for a hat – child the cube to your sphere and adjust local position to (0,1,0). Then add a forward arrow – child another cube, scale it along z and place it in front (+z):

```
 |  H <- cube hat
+y  O NNNNNNN <- long cube nose
  +z ->
Side view
```

We'll put out rotation code on this, or a cow or gun, to see rotations better.

A decent way to think about a rotation is in two parts: which way it points, and how it rolls. First we aim the nose anywhere, then we roll the hat around.

Or, same idea, point somewhere with your thumb stuck straight out. Then roll your arm so your thumb spins.

That accounts for every possibly way a 3D object can spin.

4.1 Quaternions

Rotations are *not* stored as x, y, z degrees. The Inspector for rotation is a lie. It's actually using math to convert to and from the real way rotations are stored, which we never see.

The real way everyone stores rotations is something called a quaternion. Real programs for robots or 3D space – anything with 3D rotations – have been using them for years. Unity just copied the standard way.

This small example saves our starting rotation and resets it when we press "a" (after you've hand-spun it):

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

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

void Update() {
  if(Input.GetKeyDown("a"))
    transform.rotation = savedStartFacing;  // copy back the saved rotation
}
```

Two things are interesting here. `Quaternion` is the type that stands for rotation. We can declare rotation variables like `Quaternion q;`.

The other is that our rotation, `transform.rotation`, is also a quaternion. It's not really x, y, z degrees. It never was. The Inspector is doing lots of work to show us converted values.

`transform.rotation = savedStartFacing;` is copying one quaternion into another. Quaternions are rotations, so it's copying a rotation.

Here's another example that switches my rotation with the red cube's. It uses the standard swap, with a temporary rotation variable:

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

We can't do much more than this now, since we can't create our own rotations yet.

4.2 Ways to make a rotation

In practice, we like to make rotations in different ways. Setting the x, y, z's for degrees is fine for some things. Other times we want to aim at a green cube, letting the computer figure the angles. Sometimes we want to spin around a diagonal line. Occasionally we want an offset – we're facing A and want the change to face B.

A nice thing about quaternions is they have functions to do all of those. We can set rotations using whatever function seems best for the job, and, later on, mix&match them.

4.2.1 Y, local X, local Z rotations

Using x, y, z degrees is probably the most familiar. It works like aiming an airplane: heading, climb/dive and roll.

Heading is a simple spin on y. It faces us in any compass direction. Next we tilt up or down, along a line drawn through the wings. Together those two things can point us in any direction.

But wait – the line through our wings, based on our y-spin, is our local x axis! It turns out the obvious, best way – the one we naturally use without thinking about it – is global y, then local x. Huh. That's also how Unity reads them.

Finally, we get to roll. It won't change which way we face. It's a spin around the line from tail to nose, which is our local z axis.

All together, x,y,z rotations are global y, local x (based on y,) then finally local z (based on y and x).

What that means is, suppose we want to know which way (20, 90, 284) points. We know z won't matter. 90 y means it's facing east, for sure. Then 20 x means it's tipped a little downward (left-handed coordinates. -20 is up).

When-ever you need to hand-set rotations to face somewhere, spin y to the compass direction, then x for the vertical angle. Then z just for fun.

Euler angles in code

Setting rotations with y, x and z degrees is officially called using Euler angles. In code you create them using the `Quaternion.Euler` function.

The numbers have the same meaning as in the Inspector. This points us east and 20 degrees down:

```
transform.rotation = Quaternion.Euler(20, 90, 0);
```

To make a cow face straight up, we can leave y at 0, then crank x back to -90:

```
public Transform theCow; // link to a cow
  theCow.rotation = Quaternion.Euler(-90, 0, 0);
```

The hardest part is remembering that you can't use just `transform.rotation = new Vector3(-90,0,0);`. Because rotations aren't really x,y,z angles.

We know an x-rotation by itself rolls us forwards. Technically it's making us aim up and down, but it looks like a roll. This would roll a cow forwards, tail over nose, by changing x. It's pretty simple:

```
public float xSpin=0;
```

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

Harder, suppose we want the cow facing right, which is y=90, and spinning like it's on a barbecue spit.

x is always a forward roll, tail-over-nose. `Quaternion.Euler(xSpin, 90, 0);` would have us facing right, rolling right.

But not a problem, if we stop thinking about the direction, and think about the type of roll. We want a tilt-roll, and that's z. Here's our sideways barbecue cow:

```
public float zSpin=0;

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

Getting these right takes a little practice.

A fun thing, we know y and x aim us. We can make code that uses y and x to actually aim us (using the ASWD keys):

```
public yy=0, xx=0; // heading and up/down
// z will always be 0, since it never affects how we're aimed

void Update() {
  // AD keys spin (y):
  if(Input.GetKey("a")) yy-=1;
  if(Input.GetKey("d")) yy+=1;
  // WS keys raise/lower (x):
  if(Input.GetKey("s")) xx+=1; // s is down, w is up. They look..
  if(Input.GetKey("w")) xx-=1; // ..backwards since +x is down

  transform.rotation = Quaternion.Euler(xx, yy, 0); // aim us
}
```

Simple aiming is best done like this – keep your own copies of the y and x spins. It works pretty well if you can't aim too high or too low.

There's one built-in shortcut for the Euler method. `Quaternion.identity` is short for `Quaternion.Euler(0,0,0)`. This will snap up to facing due-north:

```
transform.rotation = Quaternion.identity; // reset rotation to all 0's
```

4.2.2 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);
  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;
}
```

The arrow counts as a direction – the length doesn't matter. I just thought 10 up and 1 over was easier to read.

Another maybe real use, suppose we're spinning around only y. That's spinning around an up arrow, so we can write it as an AngleAxis. `AngleAxis(degrees, Vector3.up)` is the same as `Quaternion.Euler(0,degrees,0)`, but maybe it looks nicer since it has less numbers.

4.2.3 Look in a direction

This method of setting a rotation doesn't use any degrees at all. We give it a direction arrow and tell it to face that way. The command is `LookRotation`, and the input is one direction arrow.

An simple example, this makes a north-east facing arrow and uses it to aim us that way:

```
Vector3 v = new Vector3(1,0,1); // north east
transform.rotation = Quaternion.LookRotation(v);
```

Obviously this is just 45 degrees. A better use might be an arrow going 3 forward and 1 right, where we can't think of the angle in our head:

```
// look in whatever direction 3 forward and 1 right would be:
Vector3 v = new Vector3(1,0,3);
transform.rotation = Quaternion.LookRotation(v);
```

To aim at a point, like at some other object, we get to re-use our arrow-to-something-else math. This makes us face the red cube:

```
  Vector3 toCube = redCube.position - transform.position;
  transform.rotation = Quaternion.LookRotation(toCube);
```

This is so common – making ourself look at a spot, that there's a shortcut for it named `LookAt`. It takes a point, then computes the line from us to it and all the rest:

```
// I look at the red cube:
transform.LookAt(redCube.position);

// same as:
//   transform.rotation = Quaternion.LookDirection(
//       redCube.position - transform.position);

// the red cube looks at me:
redCube.LookAt(transform.position);
```

All of our old math works with these. If we want to look a little above the red cube, we can do it:

```
Vector3 aboveRedCube = redCube.position + Vector3.up*1.2f;
transform.LookAt(aboveRedCube);
```

A common look-at trick is getting a "flat" spin. Suppose something is on a hill and we want a car to face it by only spinning on y, not tipping back.

We can find the arrow, which slants up, but then flatten it out by setting y to 0:

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

Here's the same trick using LookAt. It makes a fake bunny position, level with us:

```
Vector3 levelBunnyPos = bunny.position;
levelBunnyPos.y=transform.position.y;
transform.LookAt(levelBunnyPos);
```

4.2.4 FromToRotation

This one is a real oddball, which won't be useful until we know how to combine rotations and more math like that. It takes two directions and gives the rotation which would take you from one to the other.

This finds the angle between a northwest arrow and a northeast arrow:

```
NW     NE
 \     /
  \   /
   \ /
90 degrees between them

Vector3 northWest = new Vector3(-2,0,2);
Vector3 northEast = new Vector3(2,0,2);
Quaternion qq = Quaternion.FromToRotation(northWest, northEast);
```

Of course, the result is a 90 degree clockwise rotation on y. But we computed it in a cool way.

It's more interesting when we have arrows to objects. This computes "if you were looking at the red cube, what extra rotation would face you to the blue one":

```
Vector3 toRed = redCube.position - transform.position;
Vector3 toBlue = blueCube.position - transform.position;
Quaternion qq = Quaternion.FromToRotation(toRed, toBlue);
```

The use for these is when we learn how to add rotations, which we can't do yet.

4.2.5 LookRotation's extra roll

`LookRotation` just points us somewhere, and, as we know, that leaves z free to roll. Normally it just leaves it at zero – "head up", But there's an optional input for z-spin.

The rule is pretty funny. Instead of giving a 0-360, you give it a direction and it tries to face your head (your local +y) that way.

Normally use uses `Vector3.up`. This aims at exactly the marker, and also rolls you onto your right side:

```
transform.LookAt(marker.position, Vector3.right);
```

The head direction is like a suggestion. It probably won't be able to face your head exactly that way. It spins on z to whichever gets it the most in that direction.

4.3 Details and math

This last section is unimportant details. A little about how quaternions really work. And then the problems with using xyz rotations. There aren't any useful tricks here But it might make you feel a little better about them.

4.3.1 Real Quaternion values

If you look inside a quaternion, there's an x, y, z and w. You might think those are degrees, but they're not. If you know trig you might think they're radians – still ice cold. They're totally different variables that happened to use x, y and z. If fact, quaternions weren't even invented to be rotations. We used them for other math, then got rid of them when calculus was invented. That's right – people using quaternions thought "calculus is so much easier than these."

If you have an irresistible urge to look at the actual numbers in a quaternion, they're simple 4-float structs. Code to look at them:

```
void Start() {
  q=Quaternion.Euler(0,0,50);
  Debug.Log(q.x +", "+ q.y +", "+ q.z +", "+ q.w);
  // (0.4, 0, 0, 0.9)
}
```

So 50 degrees on z turns into two numbers that make no sense, in slots that make no sense (x and w for z? Huh?) They're not even trig values. You can try it with a few more rotations and they make even less sense.

So, the values in a quaternion are public, but never look at them.

Unity makes them `public` because quaternions are real math, and there are quaternion formulas out there which directly change those numbers. But, really, Euler, LookDirection, AngleAxis and FromToRotation are all you ever need.

4.3.2 dot-`eulerAngles`

If you need your program to look at the yxz degrees of a quaternion, you can get them with the `eulerAngles` function:

```
  Debug.Log( transform.rotation.eulerAngles );
  // shows x, y, z; like in the Inspector
  Debug.Log( transform.rotation.eulerAngles.y );
  // shows just the y-rotation in degrees
```

Remember that the raw x,y,z's, like `transform.rotation.x`, are those non-sense 0.4 quaternion numbers. The `eulerAngle` function does the translating

into degrees.

You can see it's really translating to and from, since the numbers change. This sets using -20 for x, but converting back-and-forth translates it to 340:

```
void Start() {
  // make a rotation then check how it worked:
  Quaternion a1 = Quaternion.Euler(-20,45,0);
  Debug.Log( a1.eulerAngles ); // (340.0, 45.0, 0.0)
}
```

Because the numbers can change, `eulerAngles` is really only useful for testing.

4.3.3 Multiple ways to write an angle

A crazy thing about xyz degrees is there are 2 legitimate ways to write every angle. We can add and subtract 360, like -20 is the same as 340. But forget that. There's one other way involving an up-and-over on x.

For any rotation, you can make it by spinning the opposite way on y, then angling up and over on x, then flipping 180 degrees on z. As an example (0,0,0) is the same rotation as (180,180,180).

Try it with your finger – spin 180 to point to you, tilt up and over 180. That faces you away, palm up. Then roll 180 to be palm-down again.

Another example is (-45,90,0), which is facing right and 45 up. That can also be (-135, -90, 180): face left, but up-and-over 135 degrees, then flip upside-up on z.

It seems like we could make a rule that x has to be between -90 and +90. But that's no good, since we could legitimately get (-100,0,0) by leaning more and more backwards.

That's also (-80, 180, 180), which seems like worse numbers.

This is another reason why you can put a xyz degrees into a quaternion and get different values out.

4.3.4 How the Inspector handles rotation

The Inspector does one more odd thing while it runs your code. It saves the starting Inspector values you entered for rotation, and shows you numbers close to them.

Suppose, before starting, you enter -180 into the y-rotation Inspector slot. That's the same as 180 but using -180 is fine. If you run and your code reads

`transform.rotation.eulerAngles.y` it will see the adjusted value of 180. But the Inspector still shows -180, which is nice.

If you spin one more degree, your code sees 181 and you see -179. Still sort of nice.

The weird part is, suppose your code sets it to 90. The Inspector still thinks you like seeing y-spins close to -180. It will show you -270. It will always adjust the numbers, using mostly +/-360, to show -360 to 0 in that slot.

If you restart with 900 hand-entered there (which is also the same as 180) the numbers it shows you are adjusted

This has no effect on how the program runs, or even what it prints. Only what you see in that side window. If has no effect at all on built games, since they don't have that panel.

4.3.5 Moving with euler xyz problems

One of the confusing things about quaternions is it seems like xy rotations with degrees work just fine, so why bring in these other things. This section is about why xy degrees are terrible.

We usually imagine rotations as a globe. Every spin is the same as a particular spot on it. x and y are latitude and longitude. The two common math things we want involve two points, which was imagine as being near the equator. We want to find the angle between them, and draw that line.

The angle seems easy – compute the x and y difference and use the pythagorean theorem. For example, maybe point A is 60 degrees sideways from B and 20 degrees up. The diagonal is probably about 67. Sure, all three lines in the triangle are curved a little, but not much and maybe it cancels.

Tracing the arrow seems just as simple: proportionally increase both angles. It they hit +60 and +20 at the same time, our arrow should follow the diagonal straight from A to B.

It turns out both of those sort-of work only near the equator. Otherwise there are big problems:

We have to account for shrinking y. If we drag our nice 60,20 triangle further up the globe, 20 x is just as far, but 60 y gets smaller and smaller (just like a real globe – longitudes get closer together, in smaller rings).

The angle method also doesn't account for over-the-top. Suppose you have two points opposite and higher up (75,0) and (75,180). The shortest line is straight over the north pole. That's 30 degrees on x (75+15 to get to the pole, the 15 more to reach 70 on the other side).

Everything is actually a little bit like that. Take points in Canada at the same latitude, just 30 apart on y, but the same x. The shortest path *isn't* a globe spin. It arcs up a little more (imagine there's a second equator, with those

two points on it. That's the shortest line).

Here's an example where the "smoothly change x and y degrees" movement idea is completely wrong. Start 5 degrees in front of the north pole, (85,0) and go to the east side of the equator, (0,90). That should be a line curving down and east, just a little backwards.

But halfway through the two angles will be (42.5, 45). That's way too far forward, and totally not between those two points. The whole thing is a weird forward then back curve.

It turns out that smoothly moving degrees on a globe doesn't make the shortest line. It's always a funny curve, that gets worse and worse as we leave the equator.

All together, xy angle math is good enough near the equator, and turns to junk as you get near the poles. 3D animators work with angles a lot, and know this. For legs, they know not to use a standing globe. Instead, tip it sideways so the normal leg swing counts as being along the equator.

If your game has something that mostly spins, tipping just a little, xy degrees is fine. Angle it (using the child trick) so y is the main way you spin.

If your game is about an actual artillery piece, which for real has trouble using its 2 gears to track things mostly over it, xy degrees are perfect.

But for anything where you can aim anywhere – even just anywhere on the top half of a globe – and want correct math and smooth motion, quaternions are so much better. They don't have poles, or any other spots where spins act funny.

4.3.6 Quaternion setting commands

This section has nothing to do with quaternions or angles. As we know, computers often like to give 2 versions of commands – one that computes the answer, and another that makes you into the answer. We've seen this with `p.normalized` vs. `p.Normalize();`.

The commands from above compute an angle. That's the most useful version. These alternate commands compute and set. For example these two do the same thing:

```
q = Quaternion.AngleAxis(5, Vector3.up);
q.ToAngleAxis(5, Vector3.up); // shortcut
```

Forget about what AngleAxis does. The first one computes a rotation. Then we put it in `q`. The second one computes it straight into `q`. That's the only difference.

But quick review: `AngleAxis(Vector3.up,5)` makes a 5-degree spin on y.

Here are the rest, just so you can say you've seen them:

```
q.SetLookRotation(A);
// same as q=Quaternion.LookRotation(A);

q.SetFromToRotation(A,B);
// same as q=Quaternion.FromToRotation(A,B);

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

The last one, `q.eulerAngles=`, is confusing. To fit the pattern it should have been `q.SetEuler(0,90,10);`. But it really is a function call. `eulerAngles` is using the C# get/set trick, to be a function in disguise.

And don't let the `Vector3` in it confuse you. It counts as the xyz degrees.

These last three are super-shortcuts. You're allowed to directly assign to your local forward, up and right. It spins you that way.

```
transform.forward = toRedCube;
// same as:
transform.rotation = LookRotation(toRedCube);
```

It makes your forward arrow line up with that arrow. It's main use is if you see LookRotation, and transform.rotation=, and say "whoa, too much!". But all it does is secretly run them for you.

The next two are real oddballs. `transform.right=toRedBall;` gives you a rotation so your right side faces that direction. It really computes LookRotation and adds an extra 90 degree sideways spin.

`transform.up=toRedBall;` is the same. It's like a LookAt using your +y axis.

Most of time I see these used is when someone didn't know +z was forward. They had a cow facing +y, didn't know how to fix it, and used `transform.up=` to aim it.

Chapter 5

Combining rotations

Instead of thinking of a rotation as an absolute facing, you can think of it as a change. In other words, `Quaternion.Euler(0,90,0)` could mean facing east, or it could mean a quarter circle spin from where-ever you are 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 `v1` by `q`. It's a little like how `"cat"+"fish"` uses `+`. We're not actually multiplying the rotation numbers together, but mathematicians think `q1*q2` is a natural way to say "combine these rotations".

This is some of the hardest stuff, but if you have the basic idea, you can usually trial and error and figure out what you need after some testing.

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

vf = y90*vf;
```

The trick is to think of `y90` as a free-floating 90-degree spin. Applying that to a right arrow turns it into a backwards arrow.

It works with any kind of spin and any arrow. This finds an arrow to the red cube, then spins it 20 degrees:

```
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;
```

The ball will be the same distance from us as the red cube – we only rotated the arrow. It will be at the same height, since that's how y-spins work.

Way back in the game board example I said we only needed two corners on the bottom, not even lined up, and we could compute the other two. To do that we'll take the arrow across the bottom and spin it backwards 90 degrees to make the arrow along the side:

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

  Vector3 acrossArrow = lowerRight-lowerLeft;

  // the new part to spin the board edge arrow:
  Quaternion spin90back = Quaternion.Euler(0,-90,0);
  Vector3 forwardArrow = spin90back*acrossArrow;
```

The computed `forwardArrow` is the same length as the arrow along the bottom, running at 90 degrees to it. Adding it to each of the bottom corners will give us the top two.

A fun trick is hitting any arrow with a changing rotation. The arrow will spin around, the tip tracing a circle. This spins a red cube in a half-circle around us:

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

void Update() {
  Quaternion spin = Quaternion.Euler(0, degrees, 0); // y-spin
  Vector3 arrow = spin*(Vector3.forward*2);

  redCube.position = transform.position + arrow;

  degrees+=3; if(degrees>180) degrees=0;
}
```

It only goes 180 then snaps back on purpose, so we can see where it starts. Since we're angling a forward arrow, this traces out the right half of a circle.

If we used `Vector3.right*2`, we'd start there and spin over the bottom half.

Some notes on the rules for using these:

- You can't flip the order. `spin*v` rotates `v`, but `v*spin` is an error. That's the rule from real math.
- 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 picked multiply to mean "spin by" and there's nothing else we'd even use `+` to do.

5.1.1 Converting a rotation into its arrow

It's still nice to sometimes think of a rotation as the way it aims you, which is like an arrow. We can use our new math to get that arrow.

The trick is that no rotation, (000), is like a forward arrow. Any other rotation is how it would bend a forward arrow. Pretty sneaky, right?

`Quaternion.Euler(0,45,0)*Vector3.forward` bends the forward arrow 45 degrees, which is the way (0,45,0) counts as facing.

`q*Vector3.forward` converts rotation `q` into it's arrow.

For example, `Quaternion qq = Quatenion.Euler(-30,45,0);` is facing forward, right and a little up. `qq*Vector3.forward` is an arrow aimed forward, right and a little up.

Unity uses this trick to compute `transform.forward`. It's really our rotation times the forward arrow: `transform.rotation*Vector3.forward`.

Before, we knew how to use LookRotation to convert an arrow into a rotation. Now we know how to go the other way.

5.1.2 More arrow rotation

We can spin using any axis we want. Going around the x-axis gives an edge-on vertical half-circle:

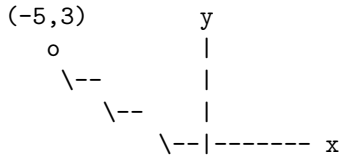
```
float degrees; // pretend this goes from 0 to 180 and snaps back

  // underground half of a circle around the x-axis:
  Quaternion spin = Quaternion.Euler(degrees, 0, 0);
  Vector3 arrow = spin*(Vector3.forward*2);
```

It's the bottom half because of the left-handed-rule: +x spins forward and down. If we changed to using a backwards arrow we'd spin over the top half.

Of course, we can code `degrees` to go backwards to spin the other way.

I've been cheating a little by using easy examples. It can be hard to visualize spinning just any arrow around any axis. For fun, let's spin (-5,3,0) a few ways.

First let's spin it around y:



The -5 will spin, making a radius 5 circle around y. The 3 won't change. The whole circle is at height 3. You can imagine the arrow tracing out a shallow upwards-facing cone.

If we roll it forwards, around the x-axis, the -5 stays the same. We'll trace out a radius 3 edge-on circle, always 5 to the left.

You can imagine the arrow, glued to x as it spins, tracing out a longer left-facing cone. But the part we see, the tip, is still just a circle.

Spinning around z is the easiest, since we're at right angles to it. To z, we count as a regular length 5.8 arrow with a funny starting spin. z-rotation will spin a circle right on the picture, centered right there on (000).

Sometimes a trick is to stop thinking of an arrow, and just imagine the point. Draw a line straight to the axis, at the nearest possible spot. It always traces out a circle with that radius.

A funny case is accidentally spinning an arrow around itself. It won't change at all. For example, this spins an up arrow around another up-arrow:

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

Pretty much this says to point straight up, then spin your finger in place. It's like we're tracing out a circle of radius 0.

It's not an error; just probably a mistake. The red cube will always be 1 unit above us.

So far we've been making rotations with the `Quaternion.Euler` method, but any way to make a rotation will work. Suppose we want to spin a ball around us, from our personal right to left. The spin should be around our forward arrow. AngleAxis was made for funny spins like this:

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

degrees+=2; if(degrees>180) degrees=0;
ball.position = transform.position + arrow; // ball circles us
```

Essentially, we're pointing along our forward, sticking out our thumb to the right, and spinning our hand.

A summary:

- Use the forward arrow to turn a rotation into it's arrow.
- For every other arrow, `q*arrow` works like an offset, spinning it more in that direction.
- There's nothing different about spinning around z. The "z doesn't matter" rule only happens for facings made with y,x,z.
- Spinning arrows that aren't going straight away from the axis is a little tricky. You have to sort of draw another line from the tip, straight to the axis, and spin that. Or else imagine the arrow spinning to make a cone.
- Any kind of rotation can be used. Especially `AngleAxis` to spin something around diagonal lines.

5.1.3 Ball cone-shooting example

Here's one longer example. I'd like to shoot a ball along +z, but randomly in a cone. My idea is to use two steps. First take the forward arrow and cock it right 0 to 10 degrees. Then spin that arrow around z by a random 0-360:

```
Vector3 dirArrow = Vector3.forward;
// small rightward cock:
dirArrow = Quaternion.Euler(0, Random.Range(0,11),0) * dirArrow;
```

Now `dirArrow` is flat, pointing forwards and a little bit right. This next part twirls it around z. Instead of slightly right, it will be slightly up, or left and down .... The possible directions make a narrow forwards cone:

```
// random 0-360 z-spin:
Quaternion zRand360 = Quaternion.Euler(0, 0, Random.Range(0,360)) ;
dirArrow = zRand360 * dirArrow;
```

We can use our usual tricks to place a ball at the end of the arrow, and give it a speed (if we make it a rigidbody) in that direction. We can shoot balls almost-forward.

Finally, suppose we want to shoot the balls the way we're facing. We can apply our rotation to `dirArrow` to spin it that way:

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

It's pretty much the trick of turning `Vector3.forward` into the arrow for some rotation. `dirArrow` is almost forward, so the result is almost our forward.

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. We take rotation `q1` and apply a 30 degree z rotation. Is it on our current z-axis, or the global z axis? Those can be very different.

The good part is, we can pick which one want. The sort of bad part is we pick by which order they go in.

5.2.1 Applying local rotations

When you multiply rotations, the first one is the starting rotation, and the second one is on the local axis of the first.

A simple example, we can take a LookRotation and give it a z-roll (the one on local z, that spins us without changing aim direction):

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

// standard z-roll. Pretend z goes from 0 to 360:
Quaternion zRoll = Quaternion.Euler(0,0,degrees);
```

These are nothing special. Notice how `zRoll` is a perfectly ordinary spin on z. It's not local or global – that depends on how we use it. This applies it as a local to the look rotation:

```
// combine: lookRotation with a z-roll:
transform.rotation = qLook*zRoll;
```

If we let it run, with the degrees on zRoll changing, we'd stayed with our nose aimed at red, z-rolling in place.

For a more math-y example, we know `Quaternion.Euler(45,90,0)` is a shortcut for a spin on global y, then local x. It's a good shortcut, but we can also write it the long way:

```
Quaternion ySpin90 = Quaternion.Euler(0,90,0);
Quaternion xSpin45 = Quaternion.Euler(45,0,0);

Quaternion y90thenx45 = ySpin90*xSpin45;

transform.rotation = y90thenx45;
```

That's just a bad way to write `Quaternion.Euler(-45,90,0)`. But we can flip it to have global x then local y.

As we all know, the Mark-II plasma cannon is mounted on a circular base with a sideways rod through it. The whole base tips straight back to aim up and down. The scarey-looking barrel swings side-to-side on the base like a clock-hand, swinging in a tilted curve.

In other words, it's an exact global-x, local-y. The code to aim:

```
float xTilt, ySpin;

void Update() {
  readUserInputs();
  // pretend AD keys move ySpin between -90 to 90 ...
  // ...and WS moves xTilt between 0 to -90 (straight up)

  Quaternion ySpin = Quaternion.Euler(0,ySpin,0);
  Quaternion xSpin = Quaternion.Euler(xSpin,0,0);

  plasmBarrel.rotation = xSpin*ySpin; // <-x is 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 to aim high, `ySpin` is a standing side-to-side arch.

If you've ever seen a real Mark-II plasma cannon, you'll recognize the distinctive rotation. You may also realize this way is much better at aiming above us than the yx way. But we just traded – it's horrible at aiming sideways. Degree and axis-based spins always have bad spots.

This next one uses 2 local rotations. We're making a game for kids called Left or Right. A 3D cow will always have a little ball ahead of it and a little bit left or right. On higher stages, the cow can roll on its side, or any which way.

For example, the cow could be lying on it's right side, aimed above the ball. The ball is to the cow's right.

The plan is to put the ball anywhere, then: 1) start with the rotation looking at the ball, 2) add a random 360 z spin. We're still looking at the ball, 3) add a +/-10 degree y-spin. Now the ball is perfectly to the cow's left or right.

The code:

```
  Vector3 toCube = cube.position - cow.position;
```

```
Quaternion lookCube = Quaternion.LookRotation(toCube);

float zz=Random.Range(0.0f, 360.0f);
Quaternion zRand360 = Quaternion.Euler(0,0,zz);

float yy=10; if(Random.value<0.5f) yy*=-1;
Quaternion yLeftRight=Quaternion.Euler(0,yy,0);
```

Those are the three simple angles. Now we combine them according to the plan:

```
cow.rotation = lookCube*zRand360*yLeftRight;
```

Visualizing it is similar to visualizing normal euler rotations: a starting rotation, then two locals; each using the current axis. Imagine the cow aimed, then spun on it's back or something. Be the cow. Then apply the final small left or right spin.

5.2.2 Applying global rotations

It's not as often that we have a starting rotation and want to apply another as a global, but we can do it.

The starting rotation goes second, and the one to apply as global goes first.

Suppose we want the red cube to have my rotation, spun by 90 degrees on the real y:

```
// pretend y90 is euler(0,90,0)
redCube.rotation = y90*transform.rotation;
```

In our minds, `transform.rotation` is the start, with `y90` added to it. Since we put it in front, it's global – a twist of our facing on the real y.

Adding a rotation as global is good when we have a plan that needs it. Here's the plan for the "almost in front cone" using only rotations: face forward and randomly 0-10 degrees right; then spin random 0-360 around z. Spinning around local z does nothing, but global z twirls us in a nice cone:

```
Quaternion qRandCone = qRandZ360 * qRightALittle;
```

The plan says `qRightALittle` is the start. We added `qRandZ360` in front to make it spin us around global z.

We can use the `*Vector3.forward` trick to turn it into an arrow. Or use it directly, for example, to aim us: `transform.rotation=qRandCone;`.

Just in case, here's the code making the rotations:

```
// forward arrow tilted right:
float tiltAmt=Random.Range(0,10.0f);
Quaternion qRightALittle = Quaternion.Euler(0, tiltAmt, 0);

// random any spin around z:
float zz=Random.Range(0, 360.0f);
Quaternion qZrand360=Quaternion.Euler(0,0,zz);
```

Next we want something for a game where a cannon spins in a circle and you have to tap when it's pointing at the target. The target can be higher than us, and we want to get the angle right.

The plan is to use LookRotation to get the correct up/down aim. Then spin around the real y-axis:

```
Quaternion qLook=Quaternion.LookRotation(target.position-transform.position);

// pretend degrees increase 0 to  360:
Quaternion y360=Quaternion.Euler(0, degrees,0);

Quaternion facing=y360*qLook;
```

The last line takes `qLook` as the main facing, then applies `y360` in front to get the global spin.

In our game you win if you tap within 5 degrees of the target, which would be 0-5 or 355-360, since 0 degrees is exactly correct.

5.3 Misc rotation combinations

With the "after=local, before=global" multiplication rule, you may have noticed a possible problem: `q1*q2` could be either. It's `q1` with `q2` applied local. It's also `q2` with `q1` applied global.

You should think of it whichever way makes the most sense to you.

For example, `lookSpin*zRoll` feels like lookSpin comes first. It's easy to visualize as a 3D model aiming, then spinning along it's local z.

But `ySpin*lookSpin` also feels like lookSpin comes first, adding ySpin as global. The 3D model is aiming, like before, but now we're giving it a merry-go-round global spin, which pulls the aim left or right.

`lookSpin*y90` is a fun one. Try thinking of `y90` first, which is an eastward facing. Then `lookSpin` is applied to it as a global. It will twist us to face the lookSpin direction, except we're already facing right. I kind of have an idea of the result, but hmmm ....

Let's try the way starting with `lookSpin`. We're facing the target and apply y90 on our local axis. That's easy – y90 on local is a right turn. All together it says "face the target and turn right". It's the equation for making your left

side face the target.

Here's one more puzzle: imagine rotation `x360` is changing to constantly roll forwards on x. And `ypm20` means "y plus/minus 20". It's a y-spin back and forth from -20 to 20. It's basically facing forward, with a sideways wiggle.

As they both move, what does `x360*ypm20` make?

I'll imagine `ypm20` as the start. It's like pointing two fingers forward in a V – it goes back&forth between them. Then we apply `x360` as a global. That rolls us directly forwards. We'll be somewhere inside that V. I can sort of visualize it, but don't love it.

The other way is `x360` first, with `ypm20` applied as a local. I can understand that – a small local y-spin is looking to your left and right. Applied to a cow, it's a forwards spin, with a left/right swing. Applied to the forwards arrow, it's a circle with a zig-zag (imagine tracking a dot on the cow's nose).

Maybe the first way makes more sense to you. But it's nice how there are two possible ways to visualize, and you only need to understand one.

Suppose someone's notes say that `qRat` is the base, in the rotation equation `qCow*qCat*qRat`. That means the others are applied as global. Go backwards from right to left. Apply `qCat` using global coordinates. If it's 20 on x, roll 20 forward on the real x. Then apply `qCow` as global to that.

It's not common, since not many problems are solved that way. But you can add a chain of global rotations by going right-to-left.

Sometimes you read it from the inside. For example `ySpin*lookSpin*zRoll`. This feels like `lookSpin`, with a local z-roll, then a global y-spin. It's the "wrong aim" example, with a fun extra tilt.

Summary, notes:

- When you see `q1*q2`, think about which way it makes the most sense: q1 with local q2 applied, or q2 with global q1 applied.
- Try to visualize what each rotation means. LookAt rotations are good starting spots, whatever position. Small local y-spins feel like "turn right or left". Likewise local z-spins feel like a fun "spin in place". Applying your rotation as global shifts it to your point-of-view.

  Think about whether a rotation is moving, and how much.
- Every rotation had a plan. In `q1*q2*q3` the person making it may have started with any of those three, applying the rest as global or local tweaks. It probably makes the most sense if you read it using their order.
- You can read these equations left-to-right and they're all locals. Or read them right-to-left and they all count as globals.

One more fun example using these. We want a model to z-roll around it's original facing. First we'll save the start rotation and create two different z-spins: global z, and our local z:

```
Quaternion mySavedQ;
Vector3 mySavedForward;
float degrees=0; // moves 0 to 360

void Start() {
  mySavedQ = transform.rotation;
  mySavedForward = transform.forward;
}

void Update() {
  degrees+=2; if(degrees>360) degrees=0;

  // global z-spin:
  Quaternion z360=Quaternion.Euler(0,0,degrees);

  // a different spin on our z:
  Quaternion zMe360=Quaternion.AngleAxis(degrees, mySavedForward);
```

Plan #1 is the most obvious one: start with our rotation and spin over our personal z. It's a world spin (which seems odd, but it is) so it goes in front:

```
transform.rotation = zMe360*mySavedQ;
```

In theory we can try reading this left-to-right instead. But spinning on `zMe360` before we're lined up on it is just a mess.

The other plan starts with our facing and applies a normal z-spin on our local z. In other words, it's a basic local z-roll. Easy to read:

```
transform.rotation = mySavedQ*z360;
```

For fun, we can sort of read it right-to-left. Start with us facing forward, rolling on z. Then global twist that into our correct rotation, which keeps the same z-roll.

I like this example since the laziest way is also the best. It feels like computing our real angleAxis z should simplify the rest, but it doesn't.

5.4 Z, global x, global y

Our math says we can read standard euler rotations backwards. Instead of y, local x, local z, we can read `y*x*z` as `z` with the other two applied as globals.

For run, let's try it:

Imagine facing forwards, spinning on z. Next global x angles us up or down. The z-roll gets carried along with us.

Finally we spin on the real `y`. The z-roll and the upward tilt are carried along as we swivel left/right.

It's bizarre that those two ways are the same result, but every other combination of global and local gives a different final rotation.

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. Since they make it so easy, we can try some fancier movement.

We know how to move simple rotations – spinning around y. Unity has some 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 bad. We should set our speeds using amount-per-second.

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

The simplest way to pro-rate movement is using the actual amount of time an update takes. We don't know how long this one will take, so we use the time since the previous.

At the start of each update, Unity computes how long it's been since the last one started, and puts it in `Time.deltaTime`. That's a good name, since delta means the amount of change.

This code adds 3.5 to x every second:

```
public float x;

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

It's pretty slick. If things run at a perfect 60fps, this is the same as dividing by 60. But even if updates have random times between them, this always adds the proportion of 3.5 based on how much time passed. Each second's worth of updates always adds to 3.5.

Whenever we want anything to change over time, we'll write down the amount per second, then multiply by `Time.deltaTime` when we use it.

An example, this moves us forward at 3.5 units/second:

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

In our minds, `transform.forward*3.5f` is the speed each second. `Time.deltaTime` computes the tiny fraction to add this update.

This lets us enter any speed/second, and moves the red cube towards us at that speed. It uses our old friend, the normalization trick:

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

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

  redCube.position += toUs*Time.deltaTime;
}
```

To use this trick properly, just make sure your speed is per second. Suppose you have `pct+=0.01f;`. That seems like just a guess at how much per frame. We'll throw that out. We really want it to go by 0.5/second.

The nicer math is `pct+=0.5f*Time.deltaTime;`.

Likewise, suppose you see `n1*Time.deltaTime` just anywhere in code. You know `n1` is something per second.

6.2 MoveTowards

Unity (and lots of other people) has a helpful function that takes a starting point, an ending point and a distance. It tells you where you'd be if you went that far along the line.

As a bonus, it won't overshoot. If A and B are 3 apart, and you tell it to move 4, it stops at B.

We can write it using stuff we already know:

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

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

  // standard normalization to go a distance:
  Vector3 unitArrow = wholeArrow.normalized;
  return A+unitArrow*dist;
}
```

It's called MoveTowards, but clearly doesn't move anything. Even so, it's pretty useful when we want to move something towards a point.

Here's a redo of moving the red cube to us, using the real built-in MoveTowards:

```
redBall.position = Vector3.MoveTowards(
    redBall.position, transform.position, speed*Time.deltaTime);
```

Notice how we finally have `Time.deltaTime`. MoveTowards takes the actual distance to move. Our `speed` is in per-seconds, and we need to convert it to the movement for this one step.

This use is the most common. We take our current position as the start. Next frame we're in a new spot, which is the start of the next MoveTowards.

This also lets us track a moving target. If we move around, or even teleport, the red cube starts moving towards there.

The movement can seem robotic, but only because we use the same speed all the time. A fun thing to do is change it. This makes us go faster when we're further away (and slower as we get close):

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

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

We could do the opposite and have it start at speed 0, but quickly increase (somewhere else would reset speed to 0 for each fresh target):

```
  speed += 3.0f*Time.deltaTime;
```

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

Notice the extra use of `Time.deltaTime` in the first line. We want to increase our speed by 3 each second. `Time.deltaTime` lets us add it gradually each frame.

A summary. MoveTowards is good when:

- The place you want to go is a specific point.

  We often just want to move in a direction. That's easy and we don't need MoveTowards for it.

- You want to move in units/second.

  The other way to move is percent-based: you'll get there in 2 seconds, at whatever speed that takes. This way takes whatever time it takes, based on the speed you use.

- You don't want to overshoot.

  Sometimes the target is only an aiming guide, and you want to go through it and past. We can do that easier the old way: compute our movement once at the start, and add it each frame.

Basically, there are lots of ways to move. MoveTowards does a nice job handling one common way.

6.2.1 Lerp

Another common way to move between two points is by using a 0-1 percent. There's a common built-in function that can help:

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

Forcing the percent to be between 0 and 1 is new, but otherwise it's merely basic offset math. But that's fine. A 2-line function with a nice name is a good deal. Lerp stands for *linear interpolation*, which is the math term for this.

A fun non-movement-based example, suppose we need cats to randomly appear on one edge of the screen. We could find the position like this:

```
Vector3 catPos = Vector3.Lerp(sideLow, sideHigh, Random.value);
// note: Random.value is a 0-1 shortcut
```

That's a nice, math-free way of saying "anywhere on a line between those two spots." It's easier to control the positioning using 0-1: fully random is easy to read, if we want a lower-half cat 0 to 0.5 is the obvious way, and so on.

Lerp for movement works the same as we did it earlier. We need to save the original starting point, and will move a percent from 0 to 1:

```
 saved
original    current
 start      position                target
   S ----------o-----------------------E
  0.0           ^ currently at         1.0
                 0.3 of the way
```

The hardest part is probably getting the speed working correctly. With percent movement we say how many seconds it takes. But then our math needs to flip it – if it takes 4 seconds, we need to go 0.25 each second.

A mini lerp-base move between two points:

```
void Update() {
  pct += (1.0f/seconds)*Time.deltaTime;
  redCube.position = Vector3.Lerp(start, end, pct);

  if(pct>=1.0f) // do whatever we do when we get there
}
```

As we already know, this moves faster or slower depending how far away. It also works funny if we move `end` midway through (our position suddenly shifts to be on the new line, the same percent of the way.)

Percent-based motion is nice for displays: a far-away coal mine has a faster moving arrow.

This also shows why Unity's standard Lerp limits pct to 0-1. Legally, a 1.1 lerp is 10% past the end, which is no problem. We've done it. But motion code don't usually want that. Unity's version lets us overshoot `pct` and stop exactly on the end.

There's another, completely different way people use Lerp. Sometimes you want a fun "shoot off and slow down" motion, like collecting a cherry and having it fly into your inventory.

The main think is we don't care about the speed or the time it takes. We only care about a fun motion. It works like this:

```
void Update() {
  cherry.position = Vector3.Lerp(cherry.position, endPoint, 0.05f);
```

Notice how the percent never changes, but the starting position does. It says to always move us 5% closer. At 60 times/frame, that adds up fast. It never gets to the end, but it gets within a rounding error pretty fast.

When you see a Lerp where the thing before the `=` matches the start, it's this fun version.

And, to repeat, this method is no good for exact numbers. If you need to go 10 units in 2 seconds, there's no math for that. The best you can do is play with the 0.05. And even if you throw in `Time.deltaTime`, this way runs slower on slower framerates.

It's just an easy way to get a special effect.

6.3 Gradually rotating

Our previous method of making a moving rotation had us change `degrees` and then re-compute the quaternion: `degrees+=2; q=Quaternion.Euler(0,degrees,0);`. It's nice to know we can use that if we have to. But there are lots of shortcuts.

There's also a new thing: we can take any two rotations and smoothly spin from one to the other.

6.3.1 Spinning an arrow

This command rotates an arrow. The new thing is we don't need to tell it how to rotate. Instead we give it the final arrow and it figures out the best way to spin to it. To visualize it, imagine both arrows coming from the same place.

The basic code looks like this:

```
arrow = Vector3.RotateTowards(arrow, endArrow, 0.02f, 0);
```

If we run this over and over, `arrow` will eventually turn into `endArrow`. The third input is the amount to spin. Sadly, it's in radians – 1 radian is about 60 degrees.

If `arrow` is 1 unit forward, and `endArrow` is 1 unit right, this does a simple 90 degree spin over y, which we could do before.

The cool part is the arrows can be in any crazy direction. It figures out the best way to spin. Imagine the arrows are two points on a globe – the spin will be the best way to fly between them.

Here's a longer real example. The 'A' key puts the ball 4 units in front of us, then it automatically spins to be 4 units straight up. If we turn ourselves in various interesting ways, we'll see it rotate along the nearest side:

```
public Transform ball;
Vector3 arrow;

void Update() {
  arrow=Vector3.RotateTowards(
```

```
      arrow, Vector3.up*4, 1.5f*Time.deltaTime, 0);

  ball.position = transform.position+arrow;

  // reset, for testing:
  if(Input.GetKeyDown(KeyCode.A)) arrow=transform.forward*4;
}
```

The speed of 1.5 is the radian issue. It translates to 90 degrees, which is the rotation per second.

A note: for arrows close together, this looks like MoveTowards. Put your fingers out in a V. MoveTowards goes straight from the tip of the first to the second. RotateTowards spins the finger. You end in the same place, but there's a little arc.

But aim two fingers nearly opposite. MoveTowards goes straight through the middle. RotateTowards make a big half-circle, going the shortest way.

If the ending arrow is a different length, it will also change the length as it spins. The 4th input is how fast.

A problem is they probably won't synch up. It might finish turning and grow the rest in place, or grow to full length quickly while only partly turned. If you need them to finish at the same time, you'll need to do the math yourself. But it usually looks fine if it's even close.

6.3.2 Moving a rotation

Rotations have a similar move-towards-like function. You give it the current rotation, the target 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 facing up, spinning at 60 degrees/second. Try it with us starting in various odd directions. Like the arrow `RotateTowards`, it rotates us the closest way.

But this also takes into account the z-roll. Roll into your back and run it – we'll spin straight up while also rolling right-side-up. For more fun, point yourself straight up but z-rolled. Running will spin you in place until you're straightened out.

The degrees per second counts the change in aiming and rolling. If you're not changing your facing much, but have to spin on z a lot, it will take longer than it seems.

Here's a practical example. We want to look at the red cube by smoothly spinning:

```
void Update() {
  Vector3 qWant=Quaternion.LookRotation(
      redCube.position-transform.position);

  transform.rotation=Quaternion.RotateTowards(
      transform.rotation, qWant, 30*Time.deltaTime);
}
```

This handles moving targets fine, since it always moves a little from our current facing. We'll rotate at a constant speed, but that won't look too bad.

Here's one more which I use sometimes. We have an object that normally only turns on y, but sometimes it gets knocked down. We want it to "get up":

```
float yFacing; // our normal y-spin

bool getUp() { // call this every frame until you get up
  // the standing up rotation:
  Vector3 qWant=Quaternion.Euler(0,yFacing,0);

  transform.rotation=Quaternion.RotateTowards(
      transform.rotation, qWant, 90*Time.deltaTime);

  // standing has our personal y as (0,1,0)
  // if y is almost 1, close enough:
  return transform.up.y>0.99f;
}
```

This isn't different than any other RotateTowards, but it looks pretty cool how we get back up to how we were facing.

6.3.3 Rotation lerp

Rotations have their own Lerp. Like the one we've seen for vectors, it takes the start rotation, the end, and a 0-1 percent.

The main use is to get in-between rotations or fractions of rotations. To get a rotation 1/2-way between `q1` and `q2`, use:

```
transform.rotation = Quaternion.Lerp(q1, q2, 0.5f);
// rotation halfway between
```

Suppose `q1` is a look-rotation to a cat, and `q2` aims you at a dog. The line above computes a rotation looking half-way between them.

If you need to cut an angle in half, a trick is to average it with no-rotation. This computes 1/2 of `q`:

```
Quaternion qHalf=Quaternion.Lerp(Quaternion.identity, q, 0.5f);
```

You might be wondering why we don't use `q/2` or `(q1+q2)*0.5f`. They aren't legal math, and even if they were, they wouldn't do rotation stuff. `Quaternion.Lerp` is the substitute.

Suppose you want to write `q2/4`. Use `Quaternion.Lerp(Quaternion.identity, q2, 0.25f);`. It's a a pain, but not too bad.

Like movement lerps, Lerp for quaternions can be used for takes-this-long spins. This spins you from start to end in 3 seconds:

```
  pct01+=(1/3.0f)*Time.deltaTime; // 0.333 each second
  transform.rotation=Quaternion.Lerp(startRotation, endRotation, pct0to1);
  if(pct01>=1) { // do done with rotation stuff
```

Quaternions can also use the Lerp fast-then-slow trick to make pretty spins. This gives a nice hurry-then-slowdown spin to make us look straight up. Notice how the percent is always 0.03, but it starts from our current spin:

```
Quaternion qUp; // pretend this was set to aiming up

void Update() {
    transform.rotation=
      Quaternion.Lerp(transform.rotation, qUp, 0.03f);
  }
}
```

This can be nice if you were going to snap to a rotation, and that would be fine. But you want something just a little nicer.

6.4 Misc, summary

Both Lerps (points and rotations) have an unclamped version, which means the percent can be 1.1 or -0.5. For points you hardly need it, since `p1-p2*0.5f+p2*1.1f;` is easier to write.

But for rotations the unclamped version is helpful. For example we can scale `q` by 1.5 or -0.1:

```
q=Quaternion.LerpUnclamped(Quaternion.identity, q, 1.5f);
// like q=q*1.5, if that were legal
q=Quaternion.LerpUnclamped(Quaternion.identity, q, -0.1f);
// like q=q*-0.1
```

A semi-realistic example, I want to turn to look at the ball, but with an overshoot past it. If I have to turn left, I'll go a little extra left; but if I need to tilt up, I'll go a little extra above it. The plan is to get the rotation to face the ball, then compute 110% of the me-to-there rotation:

```
Quaternion qToBall=Quaternion.LookRotation
    (ball.position-transform.position);

 // the overshoot rotation, starting from my current facing:
Quaternion qtbPast=Quaternion.LerpUnclamped
    (transform.rotation, qToBall, 1.1f);

transform.rotation=qtbPast;
```

To decode the Lerp: 1.0 for the percent gives the ending spot, which is looking directly at the ball. 1.1 is 10% past. Since the start of the lerp is how we're facing now, it computes the full turn, plus 10% going the same way (it took me a while to figure this out).

Quaternion Lerp has an alternate version, `Quaternion.Slerp` (spherical lerp). They both do the same thing. Lerp is actually a less accurate version that runs faster, especially for big spins. But I've never seen a difference or noticed wrong numbers for Lerp.

To summarize:

For points, MoveTowards will handle most things. The speed is in units, which is good for real motion. Changing the speed as it moves can make most effects you need.

Vector3.Lerp is harder to set-up and only useful for when you need a specific time, no matter how far. The me=Lerp(me,end,0.05) version can be fun for special effects.

And these are both short functions with simple math. You can always write it out yourself.

For rotations, Quaternion.RotateTowards handles most things – gives you a smooth, constant shortest-way spin, in degrees/second. Quaternion Lerp is very useful for math – taking half a rotation. Plus making a rare time-based spin.

Vector3.RotateTowards is for special purposes, and awkward to use. You usually move points instead of rotating arrows. And you usually rotate arrows using your rotation. But if all you have is the ending arrow, I guess it's fine.

Chapter 7

Misc Math

This section is about miscellaneous things we can do with rotations and arrows, and how they work.

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, and finds the angle of the V they make.

A nice thing is it doesn't matter if the arrows are "diagonal" to each other. It can measure the angle just as easily. If you use `RotateTowards` to spin a vector, this is measuring how far you have to go.

It always gives 0-180, and doesn't tell you which direction. If one arrow is 30 degrees from another, it could be clockwise, counter-clockwise, up, down or any diagonal. The second arrow could be anywhere on a 30-degree cone around the first.

That's a common use of Angle – 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");
```

A common error is using the positions of the two things, by mistake:

```
float angToBall = Vector3.Angle(transform.position, ball.position);
if(angToBall<30) print("a person at 000 can see both at once");
```

The inputs count as arrows, which it assumes come from the same spot. That wrong code checks the angle for someone standing at (0,0,0).

Sometimes you want the "flat" angle – like it would be on a map, not counting the up/down part of the angle. You can get that with the old flat arrow trick. Zero-out the y from both arrows:

```
Vector3 toBall = ball.position = transform.position;
toBall.y=0;
Vector3 myForward=transform.forward;
myForward.y=0;

float yAngOnly=Vector3.Angle(toBall, myForward);
```

This also won't tell you left vs. right.

Similar to `Vector3.Angle` is `Quaternion.Angle`. But `Quaternion.Angle(q1,q2)` works in the same funny way as `Quaternion.RotateTowards`. It counts the aim direction and also the z-roll difference. You usually want Vector3 angle instead.

Suppose you had a game where someone has to line up two logs. If you just 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?) check `Quaternion.Angle(log1.rotation, log2.rotation)`.

7.0.2 Cross Product

Cross Product is one of those math things you don't realize you sometimes need until you do.

Sometimes you have two arrows going into or coming out of the same spot and you need to spin one towards the other. For that, you need their combined axis-arrow, the arrow going at 90 degrees to them both. That's the cross-product (google will show you lots of pictures.)

In Unity, `Vector3 cp = Vector3.Cross(v1, v2);` will get their cross-product. It's really their axis-arrow, but people say cross product and just know that means an arrow.

Here's a fun cross-product fact: remember `FromToRotation` figures out how to spin one arrow into another? It uses `AngleAxis` to do it. It finds their cross-product, which is the axis. Then uses `Angle` to get the degrees.

Another fun fact: axes have a plus and minus direction. The cross-product follows the left-hand rule to decide which way to go. If the rotation from the first angle to the second is clockwise, the cross-product goes up, otherwise it goes down.

You can use this to find the direction of `Angle`. This assumes the two vectors are basically flat on xz. A downward cross-product means counter-clockwise:

```
float degs = Vector3.Angle(v1, v2);
// if the cross product goes down, v1 to v2 is counter-clockwise, so flip degrees:
if(Vector3.Cross(v1,v2).y<0) degs*=-1;
// degs is now a proper -180 to 180
```

Another fun cross-product fact: `Vector3.Cross(transform.forward, transform.right)` is `transform.up`. A main purposes of cross-product is using any two axis to find a third.

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. If you have a left-to-right ramp (like a /-slash,) its normal would be pointing up and to the left. Even curved surfaces have normals, but in a game most round things are made of small flat edges.

The difference between a normal and a forward arrow is that a normal has an actual flat surface that it points away from. Suppose you have a pyramid – a base and four sides. The entire thing might be facing up. But the base's normal is pointing down, and the side's normals are all pointing diagonal up.

Normals are another of those things which you'll know when you need it. Sometimes you're doing some math and need to know "which way is that wall facing." That's what the normal is for.

How do you find the normal? It depends. A common trick is to raycast (often straight down.) RaycastHit's tell you the normal of what they hit:

```
RaycastHit RH; // stores raycast data
if(Physics.Raycast(transform.position, Vector3.down, out RH)) {
  Vector3 norm=RH.normal;
  ...
```

If we're using official Unity terrain, we can ask the terrain for it's normal. It's a little messy since it wants the 0-1 percent (if the ground is 400 wide and you're at 100, you have to give it 0.25.)

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);
```

If you want the normal of something and can get three corners, the normal is the cross-product of two connecting edges. Suppose `v0` is a corner, and `v1` and `v2` are adjacent corners. The normal is:
`Vector3.Cross(v1-v0, v2-v0);`

7.0.4 Reflect

An example of using a normal is the `Reflect` command. It takes a line aimed at a wall, and figures out what direction it would bounce off.

Obviously, this depends on which way the wall is facing, which is the normal. So the inputs to Reflect are the arrow and the wall's normal. Here's a laser that can bounce off one wall (it raycasts, and if it hits a wall, reflects and raycasts again):

```
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);
}
```

If you're wondering, the way reflect works is by finding the angle between you and the normal, getting the cross product, and spinning around it by double the angle. It uses all of our fun new stuff.

7.0.5 Opposite of an angle

The same way `q*0.5f` won't give you half an angle, getting `-q` won't reverse it. The official way to flip an angle is `Quaternion.Inverse(q);`.

Many angles can be flipped without this. `Quaternion.Euler(0,-y,0)` is the opposite of positive `y`. You could also use `Quaternion.Inverse(Quaternion.Euler(0,y,0)`, but why?

`FromToRotation(A,B)` can be reversed with `FromToRotation(B,A)`.

You could even get the opposite of a ball's rotation that way:
`Quaternion.FromToRotation(ball.transform.rotation, Quaternion.identity);`

A no-limit Lerp can also compute an inverse: start at your angle and go 100% past 0: `qInv = Quaternion.LerpUnclamped(q, Quaternion.indentity, 2.0);`.

But inverse is a proper math term, and `Quaternion.Inverse(q)` might be easier to read as "the opposite" than having to decode the various longer ways.

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.

Suppose you want to find everything 3 away from you. It's faster to find everyone who's `sqrMagnitude` from you is 9 or less. If you want to find the nearest enemy, you may as well find the smallest `sqrMagnitude` from yourself.

The important thing is, `sqrMagnitude` does nothing useful – it just runs faster but makes you think harder to write the program.

7.1 Trig you should never use

Doing things with trig tends to take more testing to get it to work, and there's almost always an easier way involving built-ins. But just so you know, or if you need to use an equation with trig:

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. It's -24 degrees in Unity math.

Unity has a built-in `Mathf.Rad2Deg`, but it's just the number 57-point-whatever. It's not a full conversion. Likewise `Mathf.Deg2Rad` is just the number 1/57.

If you really 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
```

There's another method where you switch x and y in spots, with no math. It looks easy, but there are so many places to change I always mess it up.

7.1.2 atan2

A standard trig trick is turning a line's slope into its angle, using arc-tangent. If a line goes 3 right and 2 forward, arc-tangent will tell you it's at 33 degrees. But it crashes on 90 degrees and only works for half a circle (it wrongly tells you 3 left and 2 back is also 33 degrees).

The standard computer trig trick is a rewrite using x and y called `atan2`. Instead of the slope, you give it y and x. It gives the correct 0-360 angle with no crashes.

But it's in radians (and also from +x going clockwise). Bleh. Using Angle is almost always easier.

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 tells you 1 to -1. Going the same direction is 1, 90 degrees apart gives 0, and exactly opposite gives -1.

It's actually the cosine of the angle between them, so 45 degrees difference give 0.71. The same as `Angle` it can't tell left from right. -45 degrees difference is also 0.71.

Vector3.Angle is just better. The only reason to use dot product is if you happen to have some formula that needs it.

7.1.4 Rotation matrixes

A 4x4 grid of numbers, called a `Matrix4x4` in Unity, is an alternate way to represent a rotation. Graphics cards use these, so Shader programmers tend to know them.

Quaternions are a better way to handle rotations. There's no reason to use a rotation matrix unless you're forced to – like setting a value for a shader or using the old GUI system (the only way to spin was to set a rotation matrix.)

Just so you know, a 4x4 matrix really stores a rotation and a position and scale, all coded together. But we have those already, in easy to use form.

Chapter 8

Using local space

There are many problems that would be easy if we were always at (000) facing forward – for example "is this tree to my left?"

Working in local space is a technique, stolen from mathematics, that helps problems like those. This section is how to do it, and, more important, how we should be thinking.

8.1 Local Space Theory

After playing around with `transform.forward` and `transform.right*2`, and always needing to add `transform.position` you've probably invented the idea of local space for yourself.

Some of the things that become obvious as we do this math:

- Our right, up and forward make a perfect xyz grid. We may as well just call them xyz and write them the normal way, like (2,0,1).
- We won't have any serious equations that move on our x and also the global x. We can say we're working in our local space and then never have to say "our x" and "our y" ever again.

  We can say things like "In our local space, move +2 x, add `v1`, and divide `z` by 2". We don't need to explain how every single movement is using our 3 arrows.
- We're always going to start from our position. It's like our personal (000). There's no reason to write "local space (2,0,1) from us". The "from us" part is included in local space.

Mathematicians simplify it all the way. They think of local space as a fully working xyz coordinate system. It's as if a bunch of people all picked 000's and different xyz's. One is named Global, but it's not any better or different than

the others.

The whole trick is: when you have a problem, first decide which coordinate space makes it easier. If it's some local space, convert into that, do regular xyz math there, then convert back.

8.1.1 Local space and children

As we've seen, when you make something a child, Unity displays the position in the 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. Those two things are the definition of local space.

As a quick check, we can unchild ourself to see our global coords, then go back in to see local again.

We can use "children get to use parent's local space" for some neat tricks.

Suppose we have a diagonal road and need to place some lampposts and such along it. We want to place one 2.3 left of the road, and the other 2.3 right; and put the next set 4 forward from that, and so on. But the stupid diagonal road makes the math a mess.

Instead, we'll place them using the road's local space. Make a gameObject, place it in the center of the road, facing +z exactly along it. To help get it perfectly straight you could add a cube stretched along z.

Then temporarily make every lamppost a child. That makes them use the road's local space. If two mailboxes have the same x's, they're exactly across from each other. To put a lamppost 4 units further down, add 4 to z.

When we're done, drag them all out and disable the road-empty.

Notice how we never actually wanted them to be children. We invented "road local space" to make the problem easier, put our stuff into it, used it, then put everything back into global.

8.2 Local to global math

We already know how to walk around from us, using our personal xyz's. And it's kind of a pain. Thinking of it as our local space won't let us do anything new, but makes it easier.

Suppose we want to place some things around us, say at `p1` and `p2`. Our new thinking says we'll choose to work in our local space. That means we'll pretend we're 000 and the xyz coming from us is the real xyz, in perfect lines sideways, forwards and up.

Suppose `p1` should be a random amount in front of us: that's `p1=new Vector3(0,0,r);` (pretend we made `r`). Maybe `p2` goes the same amount

behind. We can use `p2=p1; p2.z*=-1;`. To go left, subtract from `x`

That's pretty boring math, which is the point. In this world, we count as being in the most boring possible spot, looking in the most boring possible direction.

We can do harder math – adding arrows, rotating them, percents. It will work the same as it did when we only had one xyz.

The last step is converting to global. It's always the same formula: spin by our rotation, then add our position. It's actually easier than all of those `forward`'s and `right`'s:

```
// convert locals to globals:
Vector3 g1 = transform.position + transform.rotation*p1;
Vector3 g2 = transform.position + transform.rotation*p2;
```

Adding our position is what we've always done. Multiplying by our rotation is new, but we've seen that math before – apply our spin to an arrow.

But we'll think of the whole equation as the formula for converting local space into global.

Another example, not using anything new: we want to put a ball into a random square in front of us. Squares are simpler in local coordinates:

```
Vector3 ball; // will be local space
ball.y=0;
ball.z=Random.Range(3.0f, 5.0f); // 3-5 in front of us
ball.x=Random.Range(-2.0f, 2.0f); // a little random left/right

// now convert to global and place it:
ball.position = transform.position + transform.rotation*ball;
```

To go back to the start of this chapter: we're placing the ball exactly as if we were are (000) facing forward. The first 4 lines are very clean. Then we bring it from local to global with the standard formula (which in theory anyone would recognize and understand what we were doing).

8.3 Bringing into local

The next harder problems deal with "where is the tree?" stuff. We want to work in local space, but the points we have are in global. We need to bring them into local.

The equation is the opposite of local-to-global: subtract our position and apply the opposite of our rotation.

This converts the tree into our local coordinates:

```
Vector3 tLocal = Quaternion.Inverse(transform.rotation)*
    (tree.position-transform.position);
```

It's the opposite of the first equation (remember Inverse is the shortest way to get the opposite rotation).

Here's the rest of the tree math. It's very simple, which is the point:

```
if(tLocal.x>0) print("tree is to my right");
else if(tLocal.x<0) print("tree is left of me");
```

For 2 trees we could pre-compute the inverse. This looks a little different, but is still the global-to-local conversion:

```
qi = Quaternion.Inverse(transform.rotation);
Vector3 t1Local = qi*(tree1.position-transform.position);
Vector3 t2Local = qi*(tree2.position-transform.position);

if(t1Local.z>0 || t2Local.z>0)
  print("a tree is ahead of us on the road");
```

With just a little practice, you can spot that equation and see it means we're bringing the trees into local space.

What if we want to know if we're ahead of or behind a truck? Not whether we can see it, but whether it can see us? We can bring ourself into the truck's local space:

```
Vector3 pt = Quaternion.Inverse(truck.rotation)*
    (transform.position-truck.position);
```

Getting that right was a little confusing – use the truck's rotation, and subtracting it's position from us. But it's just plugging 2 things into the equation.

The rest is easy:

```
if(pt.z>0) {
  print("in front of truck");
  // check for narrow strip directly in front of the truck:
  if(pt.z<5 && pt.x>-2 && pt.x<2)
    print("it's going to hit you!!");
}
```

8.4 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.

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, and change x and y to 0. That snaps it back along our line. Then we bring it back to global.

The code:

```
// get local space ball coords:
qi = Quaternion.Inverse(transform.rotation);
Vector3 bLoc = qi*(ball.position-transform.position);

// snap back to our centerline, which is easy in our local:
bLoc.x=0; bLoc.y=0;

// now convert back to global and put the ball there:
Vector3 b2=transform.position+transform.rotation*bLoc;
ball.position = b2;
```

That really will keep the ball on whatever diagonal line we happen to be facing (even with up and down tilts).

Mentally the first and last part is simply "use local space". The actual work is that little bit in the middle.

8.5 Misc problems

The basics of bringing a point into, out of, or in-and-out of local space will solve most problems like that. But sometimes there are some messy parts to worry about.

8.5.1 Mixing local/global

Mixing local coordinates and globals gives junk results. In theory it's easy to avoid – remember which points are global, which are local, and don't mix them. But it happens.

Supose `windDirection` is a global and we do this:

```
// t1 is tree1 in local space:
Vector3 t1=Quaternion.Inverse(tree1.position-transform.position;

Vector3 leafPos=t1+windDirection*3; // <- mixing local and global
```

Wen we convert back, it's going to be the wrong way. We need to use the local wind direction.

Even more rare is using more than one local space. For a demolition derby game we might use our local space and a truck's to see who's ramming who where.

Our local and truck-local are different and can't be mixed.

8.5.2 Local math from "us"

In our local space, we always count as (000) facing (0,0,1). That's easy to forget when we do math from us.

Suppose we want an arrow from us to the tree. It seems like it should be `t1-transform.position`. But we count as (000). The real arrow is just `t1` minus (000).

It's another example of local space making "relative to us" math easier. It seems weird how `t1` is a position and an arrow, but it depends on how you use it.

The other confusing part is how our forward is always (0,0,1).
Using `transform.forward` is a mistake in your local space.

Like we're been doing, to move something in our forward, add to `z`. That's why we're using local space.

8.5.3 Converting arrows

To bring an arrow into local space, use only your spin. Otherwise it's the same.

Here's a proper use of `windDirection`:

```
Quaternion qi=Quaternion.Inverse(transform.rotation);

Vector3 ballLocal=qi*(ball.position-transform.position);

Vector3 windLocal=qi*windDirection;
```

The middle line is to compare – it's global to local for a point. For the wind, we skip the subtraction and only spin.

An common use is converting a forward arrow. This brings the ball's forward into our local space:

```
Vector3 ballLocalForward = qi*ball.forward;
```

Here's an example going the other way. We figure out a velocity in local space, then bring it into global to use it:

```
Vector3 velLocal;
// set it here, using local coords since it's easier

// now turn it back into global and apply it:
Vector3 vel=transform.rotation*velLocal;
ball.GetComponent<Rigidbody>().velocity=vel;
```

If we never heard of local space we might still do it this way.
`transform.rotation*velLocal` is a standard "spin an arrow by us". But "convert local to global" is easier when you get used to it.

8.5.4 Converting rotations

Rotations use the same rule as arrows – only multiply by the spin:

```
// global to local rotation:
Quaternion ballLocalSpin=
    Quaternion.Inverse(transform.rotation)*ball.rotation;

// local to global rotation:
ball.rotation=transform.rotation*ballLocalSpin;
```

Local spins confuse me. I'd rather use local forward arrows.

But think of it this way: you've got a problem where you want to use local space, and also need to compare your spin with my spin. It's already hard. Bringing the other rotation into local will make it less hard.

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 6.

But we may as well see them:

- `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 (spins and also adds your position). But also multiplies by the scale.
- There's no version that only converts local points to global.

Multiplying by your scale is usually no effect, since most scales are (1,1,1).

The reason it's there is because children's local positions scale by the parent. A child with position (1.5, 0, 0) is 3 to your right if your x-scale is 2. As you change your scale, the child moves but it's numbers stay the same (try it).

I'm not sure if that's good or bad. But if you use childrens' positions, multiplying by the parent's scale seems right.

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.

Mostly you'll see `TransformPoint` and `InverseTransformPoint` used converting to local and back, with the scale left at (1,1,1).

8.7 Summary

The point of local space is being able to say "I wonder if this is easy in local space?"

Imagine how you'd solve it at (000) and no spin. If that way is easy, convert to local space. Usually it's way, way easier – like you were doing diagonal line math, and now you're comparing `x`'s.

If the problem involves rotations or forward arrows, yikes! But you can convert those into local space too, and the result is still probably going to be easier.

Using a few `transform.forward`'s and `right`'s is fine. Technically you're doing all global space math. If you use them a lot and things get messy, consider switching to shorthand, using only (2,0,1)-style numbers. Then convert later.

It so happens distance math is fine without it – for example finding the nearest enemy. Most angle-to math is also fine, even is-this-in-my-view-cone problems.

Chapter 9

Misc examples

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

9.1 Controlling your y-spin

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

9.2 Getting your y-spin

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

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

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

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

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

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

Another way to get 0-360 y-spin is to take the forward z-arrow of the rotation, flatten it, and find the angle between it and +z. Since the `Angle` function can't tell left from right, it's only correct if the angle is on the right side. If the arrow points left (x is negative) we flip it:

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

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

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

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

But reading `y` works fine.

9.3 Align with ground

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

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

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

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

```
// straight up w/random spin:
tree.rotation=Quaternion.Euler(0, Random.Range(0,360), 0);

// find tilt needed to align with ground:
Quaternion groundTilt=Quaternion.FromToRotation(Vector3.up, norm);
// apply as global rotation to the tree:
tree.rotation = groundTilt*tree.rotation;
```

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

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

```
Quaternion baseTreeSpin;

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

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

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

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

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

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

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

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

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

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

9.4 Orbit camera

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

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

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

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

The code:

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

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

  // standard y,x euler spinning arrow:
  Quaternion qCam = Quaternion.Euler(camXspin, camYspin, 0);
  theCam.position = transform.position+qCam*toCam;

  theCam.LookAt(transform.position);
}
```

A neat thing about using a backwards arrow is how +x finally tilts up instead of down like it normally does.

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

This would go before we use the `qCam` arrow, to spin it with the player:

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

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

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

We'll compute the camera position as normal, then use MoveTowards to go there:

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

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

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

9.5 Changing length of arrows

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

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

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

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

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

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

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

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

9.6 Drawing a line between

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

```
 duck              _____
 eyes             /     \   ball
Oo---------------|---o   |
I                 \     /
L                  -----
```

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

```
Vector3 toBall= ...

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

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

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

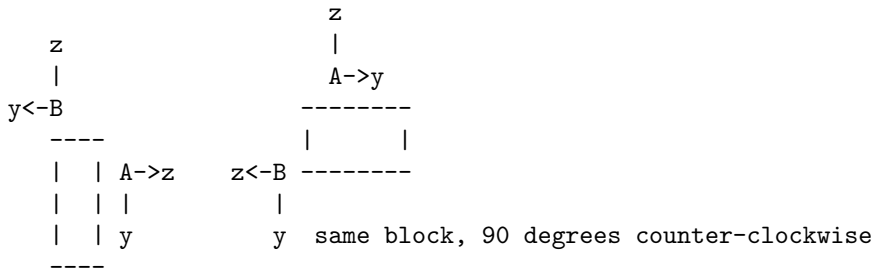
```
// arrow to camera is used for UP:
Vector3 toCam = myCam.transform.position-line.position;
line.LookAt(ball.position, toCam); // 2nd input controls z-spin
```

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

9.7 Connect two blocks

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

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



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

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

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

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

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

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

It took me a few tries to get right:

```
public Transform cube1, cube2;

void Start() {
  Transform mp1A=cube1.Find("A");
  Transform mp2B=cube2.Find("B");
  Vector3 mpAPos = mp1A.position; // start from here

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

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

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

This took a ton of testing. The first testing step was to try to get block2's rotation correct, without trying to move it yet.

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

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

```
public Transform cube1, cube2;

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

  // temporary flip so B is parent of cube2:
  mpB.parent=null; cube2.parent=mpB;
```

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

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

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